40 Important Question Openai-Sdk

ABDUL REHMAN GIAIC AND PIAIC STUDENT

1. What's the main advantage of **custom tool behavior functions** ?
2. Which method is used to execute an **agent asynchronously** ?
3. What's the purpose of **RunContextWrapper** ?
4. What does **Runner.run_sync()** do ?
5. What does **extra= "forbid"** do in a **Pydantic model config** ?
6. What's the main advantage of **strictschemas over non-strict** ?
7. What happens when you combine **StopAtTools with multiple tool names** ?
8. What is the purpose of **context** in the OpenAI Agents SDK ?
9. What's the key consideration when choosing between different **tool_use_behavior** modes ?
10. What information does **handoff_description** provide ?
11. What does **ToolsToFinalOutputResult.is_final_output** indicate ?
12. What's the difference between **hosted tools and function tools** in terms of **tool_use_behavior** ?
13. How do you **convert an agent into a tool for other agents** ?
14. What method **returns all tools available to an agent** ?
15. What is the **first parameter of every function tool** ?
16. What is the purpose of the **get_system_prompt()** method ?
17. What's the difference between **InputGuardrail and OutputGuardrail** ?
18. What is the primary purpose of the **instructions parameter in an Agent** ?

19. What does the **reset_tool_choice parameter control** ?
20. What happens if a **function tool raises an exception** ?
21. What does the **clone()** method do ?
22. What is **ToolsToFinalOutputFunction** in the OpenAI Agents SDK ?
23. What is the return type of **Runner.run()** ?
24. What happens if a custom tool behavior function **returnsis_final_output=False** ?
25. When would you use StopAtToolsinstead of **stop_on_first_tool** ?
26. What is the default value for **tool_use_behavior** in an Agent ?
27. Whats the key difference b/w **run_llm_again** and **stop_on_first_tool** in terms of performance ?
28. Which **Pydantic v2 decorator** is used for **field validation** ?
29. What is the purpose of **model_settings** in an Agent ?
30. How do you enable **non-strict mode** for **flexible schemas** ?
31. What does the **handoff_description** parameter do ?
32. How do you **implementschema** evolution while maintaining backward compatibility ?
33. Can **dynamic instructions** be **async functions** ?
34. What happens when **tool_use_behavior** isset to **'stop_on_first_tool'** ?
35. What's the difference between **mutable and immutable context patterns** ?
36. What causes the error **'additionalProperties should not be set for object types'** ?
37. When should you use **non-strictschemas over strictschemas** ?
38. In a custom **tool behavior function**, what parameters does it receive ?
39. What does **Runner.run_streamed()** return ?
40. How do you create **dynamic instructions** that change based on context ?

1. What's the main advantage of custom tool behavior functions ?

Custom Tool Behavior Functions ka Buniyadi Faida Kya Hai?

Definition ()

Custom tool behavior function aik aisa function hota hai jise aap khud banate hain taa ke woh kisi khaas **tool** (maslan, database, API service, file system, ya koi bhi external utility) ke saath *interact* karne ke tareeqe ko define kare. Iska buniyadi maqsad tool ke default behavior ko badalna, usay mazeed flexibility dena, ya uske operation mein additional logic shamil karna hota hai.

Explanation ()

Imagine karein ke aapke paas aik general-purpose tool hai jo kayi jagah istemal ho sakta hai. Lekin, har jagah us tool se aapki zarurat thodi mukhtalif hai. Yahan par **custom tool behavior functions** kaam aate hain.

`Custom tool behavior functions` ka **buniyadi faida** hai:

"Enhanced Flexibility and Control over Tool Interaction Logic"

(Tool ke sath Rabte ke Logic par Behtar Lachak aur Control)

Iska matlab hai ke aap tool ke default tareeqa-e-kaar par poora control hasil kar lete hain. Is faide ko mazeed samajhne ke liye, chand points par ghaur karein:

1. **Tailored Logic (Apni Marzi ka Logic):**
   - Aap tool se kis tarah data bhejna chahte hain, ya data receive karne ke baad usay kaise process karna chahte hain, ye sab aap apni marzi ke mutabiq set kar sakte hain.
   - Maslan, agar aapka tool kisi API se data fetch karta hai, to aap `custom behavior function` mein data fetch karne se pehle authentication tokens refresh kar sakte hain, ya fetch karne ke baad data ko kisi khaas format mein parse kar sakte hain.
2. **Centralized Error Handling (Markazi Ghalti ki Shinakht):**
   - Tools se interact karte waqt ghaltiyan (errors) aana aam baat hai. **Custom behavior functions** aapko yeh mauqa dete hain ke aap tamam tool-related errors ko aik hi jagah par handle karein.
   - Is se code saaf suthra rehta hai aur debugging (ghaltiyan theek karna) asaan ho jata hai. Aap re-tries implement kar sakte hain, ya specific error codes par mukhtalif actions le sakte hain.
3. **Cross-Cutting Concerns (Mushtarka Masail):**
   - Kuch kaam aise hote hain jo har tool interaction ke sath zaroori hote hain, jaise logging (data record karna), performance monitoring (karkardagi ki nigrani), ya caching (data ko temporary save karna).

- **Custom behavior functions** in "cross-cutting concerns" ko tool logic mein directly bake karne ki ijazat dete hain, jis se aapka core application logic saaf rehta hai. Aapko har baar API call karne se pehle manually log nahin karna padega, balkay custom behavior function khud kar lega.

4. **Abstraction (Amaliyat ki Poshidagi):**
   - Ye tool ke andar ki pechida working (complicated working) ko application ke baqi hisson se chupate hain. Application ko sirf ye pata hota hai ke usay "ye kaam karwana hai," usey is baat se matlab nahin hota ke "ye kaam kaise ho raha hai."
   - Agar tool ka internal mechanism badal bhi jaye, to aapko sirf `custom behavior function` mein change karna hoga, application ke baqi hisse mutasir nahin honge.
5. **Testability (Qabili-e-Azmaish):**
   - Jab tool interaction logic centralized hota hai, to usko test karna asaan ho jata hai. Aap tool ke actual implementation ke bajaye, mock objects (naqli objects) ke sath `custom behavior function` ko test kar sakte hain.

Mukhtasaran, **`custom tool behavior functions`** aapko apne application aur external tools ke darmiyan aik "smart intermediary" (aqalmand darmiyani) banane ki ijazat dete hain, jo aapki khaas zaruriyat ke mutabiq tool ke operations ko adjust kar sakta hai.

Example Code ()

Aaiye aik misal dekhte hain jahan hum aik simple `HTTP client` tool ke liye `custom` **`tool behavior function`** banate hain taa ke har request se pehle `API key` add ho sake aur errors ko handle kiya ja sake.

```
import requests # Aik simple HTTP client library
import json
import time
from typing import Dict, Any

# --- Simple HTTP Client Tool ( فرض کریں یہ آپ کا ٹول ہے ) ---
class HttpClient:
    def __init__(self, base_url: str):
        self.base_url = base_url

    def get(self, endpoint: str, headers: Dict[str, str] = None) -> Dict[str, Any]:
        url = f"{self.base_url}/{endpoint}"
        print(f"\n[Tool]: Fetching from {url}")
        response = requests.get(url, headers=headers)
        response.raise_for_status() # Raise an HTTPError for bad responses (4xx or 5xx)
        return response.json()

    def post(self, endpoint: str, data: Dict[str, Any], headers: Dict[str, str] = None) ->
Dict[str, Any]:
        url = f"{self.base_url}/{endpoint}"
        print(f"\n[Tool]: Posting to {url} with data: {data}")
        response = requests.post(url, json=data, headers=headers)
        response.raise_for_status()
        return response.json()

# --- Custom Tool Behavior Function ---
# ( یہ فنکشن ٹول کے رویے کو کسٹمائز کرے گا )
def my_custom_http_behavior(tool_instance: HttpClient, method_name: str, *args: Any, **kwargs:
Any) -> Any:
    """
    HTTP client ke liye custom behavior function.
    Is mein API key injection aur centralized error handling shamil hai.
```

```
    """
    api_key = "my_secret_api_key_123" # Fake API Key

    # Har request se pehle headers mein API key add karein
    if 'headers' not in kwargs:
        kwargs['headers'] = {}
    kwargs['headers']['X-API-Key'] = api_key
    print(f"[Custom Behavior]: Adding X-API-Key: {api_key}")

    try:
        # Tool ka asal method call karein (get ya post)
        method = getattr(tool_instance, method_name)
        result = method(*args, **kwargs)
        print(f"[Custom Behavior]: Request successful for {method_name}.")
        return result
    except requests.exceptions.HTTPError as e:
        print(f"[Custom Behavior ERROR]: HTTP Error occurred: {e.response.status_code} -
{e.response.text}")
        if e.response.status_code == 401:
            print("[Custom Behavior ERROR]: Unauthorized! Check API Key.")
            # Yahan aap API key refresh karne ka logic bhi dal sakte hain
        raise # Error ko aage propagate karein
    except requests.exceptions.ConnectionError as e:
        print(f"[Custom Behavior ERROR]: Connection Error occurred: {e}")
        # Yahan re-try logic ya network check logic aa sakta hai
        raise
    except Exception as e:
        print(f"[Custom Behavior ERROR]: An unexpected error occurred: {e}")
        raise

# --- Application Logic (Main Code) ---
if __name__ == "__main__":
    # Apne tool ko initialize karein
    api_tool = HttpClient(base_url="https://jsonplaceholder.typicode.com") # Testing ke liye fake
API

    print("--- Getting a post (Successful Case) ---")
    try:
        # Custom behavior function ke zariye tool ka istemal karein
        # 'my_custom_http_behavior' ab HttpClient.get() method ko control karega
        post_data = my_custom_http_behavior(api_tool, 'get', 'posts/1')
        print(f"Received Post Data: {json.dumps(post_data, indent=2)}")
    except Exception as e:
        print(f"Application caught error: {e}")

    print("\n--- Getting a non-existent post (Error Case) ---")
    try:
        # Galat endpoint, jisse 404 error aana chahiye
        non_existent_data = my_custom_http_behavior(api_tool, 'get', 'posts/99999')
        print(f"Received Non-existent Data: {json.dumps(non_existent_data, indent=2)}")
    except Exception as e:
        print(f"Application caught error for non-existent post: {e}")

    print("\n--- Posting new data (Successful Case) ---")
    try:
        new_post = {"title": "foo", "body": "bar", "userId": 1}
        response_post = my_custom_http_behavior(api_tool, 'post', 'posts', data=new_post)
        print(f"New Post Response: {json.dumps(response_post, indent=2)}")
    except Exception as e:
        print(f"Application caught error for new post: {e}")
```

Code Example ki Wazahat ():

1. **`HttpClient` Tool:** Yeh aik simple Python class hai jo HTTP GET aur POST requests bhejti hai. Ye hamara "tool" hai.
2. **`my_custom_http_behavior` Function:** Yeh hamara **custom tool behavior function** hai.
    - Ye `tool_instance` (yaani `HttpClient` ka object) aur `method_name` (yaani `get` ya `post`) receive karta hai.
    - **Faida (Benefit):** Is function ke andar, hum har request se pehle `X-API-Key` header add kar rahe hain, jo ke authentication ke liye zaroori ho sakta hai. Ye logic tool ke asal `get` ya `post` methods mein nahin hai, balkay bahar se inject kiya gaya hai.
    - **Faida (Benefit):** Is mein centralized `try-except` block bhi hai jo mukhtalif qism ki `requests` library ki ghaltiyon ko handle karta hai (jaise `HTTPError`, `ConnectionError`). Agar `401 Unauthorized` jaisi ghalti aaye to yeh aik specific message deta hai.
    - `getattr(tool_instance, method_name)(*args, **kwargs)` line asal mein tool ke method ko dynamically call karti hai, saare arguments aur keyword arguments ke sath.
3. **Application Logic:** `if __name__ == "__main__":` block mein, hum `api_tool` ko initialize karte hain. Phir, jab hum `api_tool.get()` ya `api_tool.post()` ko call karna chahte hain, to hum direct call karne ke bajaye, unhein `my_custom_http_behavior` ke andar wrap karte hain.

Is tarah, application ko tool se interact karne ke liye sirf `my_custom_http_behavior` ko call karna hota hai, aur woh function khud-ba-khud API key inject karta hai aur errors ko handle karta hai. Application ke core logic ko in details se paak rakha gaya hai.

Summary ()

Custom tool behavior functions ka buniyadi faida unki flexibility aur control hai jo woh external tool interactions par faraham karte hain.

Yeh aapko tool ke default functionality par aik additional layer banane ki ijazat dete hain, jahan aap:

- Specific business logic shamil kar sakte hain.
- Centralized tareeqe se errors ko handle kar sakte hain.
- Common concerns jaise logging ya authentication ko automate kar sakte hain.
- Tool ki andaruni pechidegiyon ko application code se chhupa sakte hain.
- Apne code ki testability ko behtar bana sakte hain.

Natijaatan, aapka code zyada saaf, maintainable, aur adaptive ho jata hai, kyunki aap tool ke istemal ke tareeqe ko apni application ki khaas zaruriyat ke mutabiq dhal sakte hain. Yeh aapko mukhtalif contexts mein aik hi tool ko alag alag tareeqon se istemal karne ki taaqat deta hai, baghair tool ke core code mein tabdeeli kiye.

2. Which method is used to execute an Agent Asynchronously ?

Agent Asynchronously Execute Karne Ke Liye Kaunsa Method Istemal Hota Hai?

Definition ()

Aik agent ko **Asynchronously execute** karne ka matlab hai usay is tarah chalana ke woh apna kaam karte hue doosre tasks ko bhi saath-saath chalne de, baghair poore program ko roke. Is se application ki responsiveness aur efficiency behtar hoti hai, khaas kar ke jab agent ko lambe arse tak chalne wale operations (jaise network requests, database queries, ya LLM calls) karne hon.

Explanation ()

Jab aap aik agent ko asynchronously execute karte hain, to iska buniyadi maqsad yeh hai:

"Application Ki Overall Responsiveness Aur Efficiency Ko Barqarar Rakhte Hue, Agent Ko Background Mein Lambe Operations Perform Karne Ki Ijazat Dena"

(Allowing the Agent to Perform Long-Running Operations in the Background While Maintaining the Application's Overall Responsiveness and Efficiency)

Is maqsad ko mazeed tafseel se samjhte hain:

1. **Non-Blocking Operations (Non-Blocking Amaliyat):**
   - **Faida:** Asynchronous execution ka sabse bada faida yeh hai ke yeh **non-blocking** hota hai. Jab agent koi aisa kaam karta hai jis mein waqt lagta hai (maslan, kisi API se data fetch karna), to synchronous execution mein poora program us data ka intezar karta hai. Asynchronous execution mein, agent data ka intezar karte hue doosre tasks ko control de deta hai, jis se application freeze nahin hoti.
   - **Misal:** Agar aapka web server aik agent ko chalata hai to agar agent synchronous ho to jab tak agent ka kaam poora nahin hota server doosri requests ko process nahin kar sakta. Asynchronous agent server ko doosri requests handle karne ki ijazat deta hai.
2. **Concurrency (Aik Sath Kaam Karna):**
   - **Faida:** Asynchronous methods concurrency (aik se zyada kaam aik hi waqt par handle karna) allow karte hain. Aik hi thread mein aap kai agents ya agent ke mukhtalif hisson ko concurrently chala sakte hain, jo I/O-bound operations ke liye bohat faidamand hai.
   - **Misal:** Agent aik hi waqt par kai APIs se data fetch kar sakta hai, ya kai emails bhej sakta hai, baghair har operation ke liye alag thread banaye.
3. **Resource Utilization (Resources ka Behtar Istemal):**
   - **Faida:** Threads ke muqable mein, asynchronous operations (khaas kar `asyncio` jaisi libraries mein) system resources (CPU, memory) ka behtar istemal karte hain. Har asynchronous task (coroutine) ke liye alag thread banane ki zarurat nahin hoti, jis se overhead kam hota hai.
   - **Misal:** Aik single event loop hazaron concurrent I/O operations ko manage kar sakta hai, jabke utni hi threads chalana system ko bojh talay daba sakta hai.

4. **Scalability (Paimanai Salahiyat):**
    - **Faida:** Asynchronous agents zyada scalable hote hain. Woh aik hi server par zyada concurrent requests ya tasks ko handle kar sakte hain, kyunki woh blocking operations ke dauran idle nahin rehte.

Kis Method Ka Istemal Hota Hai?

OpenAI Agents SDK (aur aam taur par Python mein asynchronous programming) mein agent ko asynchronously execute karne ke liye jo buniyadi method istemal hota hai, woh hai:

`async def agent.run()`

Ya koi bhi aisa method jiske naam ke shuru mein **`async`** laga ho aur jo **`await`** keyword ke sath call kiya jata ho.

- `async` keyword function ko coroutine banata hai.
- `await` keyword coroutine ko pause karta hai jab tak koi I/O operation mukammal na ho jaye, is dauran control event loop ko wapis de diya jata hai taake woh doosre ready tasks ko chala sake.

Agar aap ko ek synchronous context se asynchronous agent ko run karna hai, to aap `asyncio.run()` ka istemal karte hain apne main `async` function ko chalane ke liye.

Example Code ()

Chaliye aik `AsyncAgent` ki misal dekhte hain jo **`async def run()`** method istemal karta hai.

```
import asyncio
import time
from typing import Dict, Any

# --- Hypothetical Asynchronous Tool ---
async def fetch_data_from_api(url: str) -> Dict[str, Any]:
    """Simulates an asynchronous API call."""
    print(f"\n[Tool]: Asynchronously fetching data from {url}...")
    await asyncio.sleep(2)  # Simulate network latency
    print(f"[Tool]: Data fetched from {url}.")
    return {"status": "success", "data": f"content from {url}"}

# --- Asynchronous Agent Class ---
class MyAsyncAgent:
    def __init__(self, agent_name: str):
        self.agent_name = agent_name
        self.processed_data = []

    async def run(self, query: str) -> str:
        """
        Agent ka asynchronous run method.
        Yeh agent ke main execution flow ko handle karta hai.
        """
        print(f"\n--- Agent {self.agent_name} started processing query: '{query}' ---")

        # Simulate agent's initial processing
        await asyncio.sleep(0.5)
        print(f"Agent {self.agent_name}: Initial thoughts on '{query}'.")
```

```
# Asynchronously call a tool
        if "data" in query.lower():
            print(f"Agent {self.agent_name}: Decided to fetch data using an async tool.")
            data = await fetch_data_from_api("https://example.com/api/data")
            self.processed_data.append(data)
            response = f"Agent {self.agent_name}: Fetched data and processed it. Status:
{data['status']}"
        elif "calculate" in query.lower():
            print(f"Agent {self.agent_name}: Performing an async calculation.")
            await asyncio.sleep(1) # Simulate async computation
            response = f"Agent {self.agent_name}: Calculation complete for '{query}'."
        else:
            response = f"Agent {self.agent_name}: I'm just thinking about '{query}'."

        print(f"--- Agent {self.agent_name} finished processing query: '{query}' ---")
        return response

# --- Main Asynchronous Function to Run Agents ---
async def main():
    agent1 = MyAsyncAgent("Alpha")
    agent2 = MyAsyncAgent("Beta")

    # Agents ko concurrently run karein
    print("Main: Launching Agents concurrently...")
    # asyncio.gather() ka istemal multiple awaitables (agent.run() calls) ko aik sath chalane ke
liye
    results = await asyncio.gather(
        agent1.run("Please fetch some data."),
        agent2.run("Let's do some calculations."),
        agent1.run("What about more data?") # Agent1 ko dobara bhi chala sakte hain
    )
    print("\nMain: All Agents finished their tasks.")
    for i, res in enumerate(results):
        print(f"Result {i+1}: {res}")

    print(f"\nAgent Alpha's processed data: {agent1.processed_data}")
    print(f"Agent Beta's processed data: {agent2.processed_data}")


# --- Entry Point for Asynchronous Program ---
if __name__ == "__main__":
    start_time = time.time()
    asyncio.run(main()) # asynchronous main function ko chalane ke liye
    end_time = time.time()
    print(f"\nTotal execution time: {end_time - start_time:.2f} seconds.")
    # Agar yeh synchronous hota, to har agent ka time sum up ho jata (approx 2s + 0.5s + 1s + 2s +
0.5s = 6s)
    # Asynchronously, yeh sab concurrently chalte hain (approx max of any individual async
operation, which is 2s)
```

Code Example ki Wazahat ():

1. **`fetch_data_from_api` (Asynchronous Tool):** Yeh aik `async def` function hai jo `await asyncio.sleep(2)` istemal karta hai taake yeh dikhaya ja sake ke yeh aik blocking I/O operation ko simulate kar raha hai.
2. **`MyAsyncAgent` Class:**
   - Ismein `async def run(self, query: str)` method hai. Yahi woh method hai jo agent ko asynchronously execute karta hai.
   - Agent ke andar, jab usay `fetch_data_from_api` (jo khud bhi async hai) ko call karna hota hai, to woh **`await`** keyword ka istemal karta hai. Is `await` ki wajah se, jab `fetch_data_from_api` data ka intezar kar raha hota hai, to `MyAsyncAgent.run()` method pause ho jata hai aur control event loop ko wapis de diya jata hai.
3. **`main()` (Asynchronous Entry Point):**
   - Yeh bhi aik `async def` function hai.
   - `asyncio.gather()` ka istemal kiya gaya hai taake mukhtalif agent instances ke `run()` methods ko **concurrently** chalaya ja sake. Iska matlab hai ke Agent Alpha aur Agent Beta aik hi waqt mein kaam shuru kar dete hain. Jab aik agent `await` karta hai, to doosra agent chalna shuru kar sakta hai.
4. **`asyncio.run(main())`:** Yeh standard Python ka tareeqa hai aik `async def` function ko chalane ka. Yeh aik event loop banata hai aur `main()` coroutine ko us loop mein execute karta hai.
5. **Total Execution Time:** Output mein aap dekhenge ke total execution time synchronous execution ke muqable mein kafi kam hai (approx 2.5-3 seconds). Yeh is liye hai kyunki blocking operations (jo `await asyncio.sleep()` se simulate kiye gaye hain) concurrently handle hote hain.

Summary ()

Agent ko **asynchronously execute** karne ka buniyadi maqsad **application ki responsiveness aur efficiency ko behtar banana** hai, khaas kar ke jab agent ko lambe arse tak chalne wale I/O-bound operations karne hon.

Is maqsad ke liye, agent classes mein aam taur par **`async def run()`** (ya is jaisa koi `async` method) ka istemal kiya jata hai. Yeh `async` method `await` keyword ka istemal karta hai takay blocking operations ke dauran control event loop ko wapis diya ja sake, jis se doosre tasks ko concurrently chalaya ja sakta hai. Natija yeh hota hai ke agent backgrounds mein efficiently kaam karta hai, aur poori application non-blocking rehti hai, jis se behtar user experience aur zyada scalability milti hai.

Agar aap ko ek synchronous context se asynchronous agent ko run karna hai, to aap **`asyncio.run()`** ka istemal karte hain apne main `async` function ko chalane ke liye.

3. What's the purpose of RunContextWrapper ?

`RunContextWrapper` Ka Maqsad Kya Hai?

Definition ()

`RunContextWrapper` aik aisa **utility class** ya **design pattern** hai jo kisi bhi "run" (execution) ke **context** ko manage aur standardize karne ke liye istemal hota hai, khaas tor par complex systems ya frameworks mein jahan multiple components ya steps shamil hon. Iska buniyadi maqsad yeh yaqeeni banana hai ke execution ke dauran zaroori maloomat aur services asani se aur yaksaa tareeqay se accessible hon.

Explanation ()

`RunContextWrapper` ka **buniyadi maqsad** hai:

"Execution Ke Mahol (Context) Ko Yaksaa Tareeqay Se Faraham Karna Aur Uski Aasani Se Rasai Mumkin Banana" (Standardizing and Facilitating Access to the Execution Environment (Context))

Iska matlab hai ke yeh kisi bhi code block ya operation ke liye aik aisa *container* ya *wrapper* banata hai jo us operation ke liye relevant tamam maloomat, configurations, aur services ko aik hi jagah par jama karta hai. Is se code zyada saaf, robust, aur maintainable banta hai.

Aaiye is maqsad ko mazeed tafseel se samajhte hain:

1. **Centralized Access to Contextual Information (Contextual Maloomat Tak Markazi Rasai):**
   - **Faida:** Kisi bhi run ke dauran, mukhtalif components ko aik hi tarah ki maloomat ki zarurat ho sakti hai (maslan, current user ID, session parameters, logging preferences, active database connection). `RunContextWrapper` in sab maloomat ko aik hi object mein wrap kar deta hai. Components ko ab in maloomat ko alag alag jagah se dhoondna nahin padta, balkay woh sirf `RunContextWrapper` object se access kar sakte hain.
   - **Misal:** Aik agent jab koi task perform kar raha hota hai, to usay `user_id`, `request_id`, aur `logging_level` jaisi maloomat ki zarurat hoti hai. `RunContextWrapper` yeh sab maloomat aik hi object ke zariye faraham karta hai.
2. **Resource Management (Resources ka Intezaam):**
   - **Faida:** Yeh wrapper resources (jaise database connections, network clients) ko manage karne mein madad kar sakta hai, khaas kar ke jab woh resources run ke lifecycle tak mehdood hon. Yeh yaqeeni banata hai ke resources sahi waqt par allocate aur deallocate hon.
   - **Misal:** Aik `RunContextWrapper` database connection ko initialize kar sakta hai jab run shuru ho aur usay close kar sakta hai jab run khatam ho.
3. **Dependency Injection (Dependencies ka Andar Dalna):**
   - **Faida:** `RunContextWrapper` aksar "dependency injection" ke liye bhi istemal hota hai. Is ka matlab hai ke woh un services ya objects ko automatically "inject" (andar dal) karta hai jin ki kisi component ko zarurat

hoti hai, bajaye iske ke component khud unhein create kare ya globally access kare. Is se code zyada testable aur loosely coupled (kam jura hua) banta hai.
    - **Misal:** Logging service ya telemetry client ko `RunContextWrapper` ke zariye mukhtalif functions tak pass kiya ja sakta hai.
4. **Error Handling and State Propagation (Ghalti Ki Shinakht Aur Halat Ka Phelao):**
    - **Faida:** Run ke dauran paida hone wali ghaltiyon ko handle karne ke liye aik yaksaa mechanism faraham kar sakta hai. Yeh error state ko context ke zariye agay propagate (phela) kar sakta hai, taake mukhtalif layers par usay handle kiya ja sake.
    - **Misal:** Agar run ke dauran koi unhandled exception ho jaye, to `RunContextWrapper` us exception ko capture kar sakta hai aur logging system ko alert kar sakta hai.
5. **Testability (Qabil-e-Azmaish Hona):**
    - **Faida:** Jab context aik wrapper mein hota hai, to unit testing asaan ho jata hai. Aap test ke dauran `RunContextWrapper` ke mock (naqli) versions bana sakte hain, jin mein aapko zaroorat ke mutabiq data aur services shamil kar sakte hain, asli dependencies ko istemal kiye baghair.
6. **Code Cleanliness and Readability (Code Ki Safai Aur Parhne Mein Asani):**
    - **Faida:** Functions aur methods ko bahut se alag alag parameters pass karne ke bajaye, aap sirf aik `RunContextWrapper` object pass kar sakte hain. Is se function signatures saaf rehte hain aur code parhne mein asaan hota hai.

Mukhtasar yeh ke, `RunContextWrapper` aik aisa tool hai jo complex applications mein execution context (sahaq-o-sabaq) ko manage karne ka aik saaf, efficient, aur robust tareeqa faraham karta hai.

Example Code ()

Chaliye aik simple Python example dekhte hain jo **`RunContextWrapper`** ke concept ko wazeh karta hai.

```
import logging
import time
from typing import Dict, Any, Optional

# --- 1. Resources/Services jo Context mein hongi ---
class DatabaseConnection:
    def __init__(self, db_name: str):
        self.db_name = db_name
        self.is_connected = False
        print(f"DB: Connecting to {self.db_name}...")
        time.sleep(0.1) # Simulate connection time
        self.is_connected = True
        print(f"DB: Connected to {self.db_name}.")

    def fetch_user_data(self, user_id: str) -> Dict[str, Any]:
        if not self.is_connected:
            raise ConnectionError("Database not connected.")
        print(f"DB: Fetching data for user_id={user_id} from {self.db_name}...")
        time.sleep(0.2)
        if user_id == "user_123":
            return {"id": user_id, "name": "Ali Raza", "email": "ali@example.com"}
        return {"id": user_id, "name": "Unknown", "email": "N/A"}

    def close(self):
        if self.is_connected:
            print(f"DB: Closing connection to {self.db_name}.")
            time.sleep(0.1)
            self.is_connected = False
```

```
class LoggingService:
    def __init__(self, log_level: str):
        self.logger = logging.getLogger("AppLogger")
        self.logger.setLevel(log_level.upper())
        handler = logging.StreamHandler()
        formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
        handler.setFormatter(formatter)
        if not self.logger.handlers: # Avoid adding multiple handlers if already present
            self.logger.addHandler(handler)
        print(f"Log: Logging service initialized with level {log_level.upper()}.")

    def info(self, message: str):
        self.logger.info(message)

    def error(self, message: str):
        self.logger.error(message)

# --- 2. RunContextWrapper Class ---
class RunContextWrapper:
    """
    A wrapper to hold all contextual information and services for a single run.
    """
    def __init__(self, request_id: str, current_user_id: str, config: Dict[str, Any]):
        self.request_id = request_id
        self.current_user_id = current_user_id
        self.config = config
        self._db_connection: Optional[DatabaseConnection] = None
        self._logging_service: Optional[LoggingService] = None



@property
    def db(self) -> DatabaseConnection:
        if self._db_connection is None:
            self._db_connection = DatabaseConnection(self.config.get("database_name",
"default_db"))
        return self._db_connection

    @property
    def log(self) -> LoggingService:
        if self._logging_service is None:
            self._logging_service = LoggingService(self.config.get("log_level", "INFO"))
        return self._logging_service

    def __enter__(self):
        # Context manager entry: Initialize resources (if not already)
        self.log.info(f"RunContextWrapper {self.request_id}: Entering context for user
{self.current_user_id}")
        # db connection will be lazily initialized when first accessed
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        # Context manager exit: Clean up resources
        if self._db_connection:
            self._db_connection.close()
        self.log.info(f"RunContextWrapper {self.request_id}: Exiting context.")
        if exc_type:
            self.log.error(f"RunContextWrapper {self.request_id}: Exited with error: {exc_val}")


# --- 3. Functions jo Context Wrapper istemal karengi ---

def process_user_request(context: RunContextWrapper, user_id_to_fetch: str):
    """
```

```
    This function processes a user request using the provided context.
    It doesn't need to know how DB or Log are initialized.
    """
    context.log.info(f"Processing request for {user_id_to_fetch} (Request ID:
{context.request_id})")

    user_data = context.db.fetch_user_data(user_id_to_fetch)
    context.log.info(f"Fetched user data: {user_data['name']} ({user_data['email']})")

    # Simulate some business logic
    if user_data['name'] == "Unknown":
        context.log.error(f"User {user_id_to_fetch} not found in database.")
        raise ValueError("User data not available.")

    context.log.info(f"Request for {user_id_to_fetch} processed successfully.")
    return user_data


# --- 4. Main Application Flow ---
if __name__ == "__main__":
    app_config = {
        "database_name": "app_users",
        "log_level": "DEBUG"
    }

    # Scenario 1: Successful run
    print("--- Scenario 1: Successful User Data Fetch ---")
    try:
        with RunContextWrapper(request_id="REQ_001", current_user_id="admin", config=app_config)


as context:
            fetched_data = process_user_request(context, user_id_to_fetch="user_123")
            print(f"Application received: {fetched_data}")
    except Exception as e:
        print(f"Application caught an error: {e}")
    print("-" * 40)

    # Scenario 2: User not found (Error handling via context)
    print("\n--- Scenario 2: User Not Found (Error Handled) ---")
    try:
        with RunContextWrapper(request_id="REQ_002", current_user_id="guest", config=app_config)
as context:
            fetched_data = process_user_request(context, user_id_to_fetch="user_999") # This will
fail
            print(f"Application received: {fetched_data}")
    except ValueError as e:
        print(f"Application specifically caught: {e}")
    except Exception as e:
        print(f"Application caught generic error: {e}")
    print("-" * 40)

    # Scenario 3: Multiple runs (each with its own context)
    print("\n--- Scenario 3: Multiple Independent Runs ---")
    with RunContextWrapper(request_id="REQ_003", current_user_id="reporter", config=app_config) as
ctx1:
        process_user_request(ctx1, user_id_to_fetch="user_123")

    with RunContextWrapper(request_id="REQ_004", current_user_id="auditor", config=app_config) as
ctx2:
        process_user_request(ctx2, user_id_to_fetch="user_123")
    print("-" * 40)
```

Code Example ki Wazahat ():

1. **Resources/Services (`DatabaseConnection`, `LoggingService`):** Yeh woh services hain jin ki aapko apni application mein zaroorat hoti hai. Yeh apne kaam khud karti hain aur inhein initialize karne mein kuch steps shamil ho sakte hain (jaise DB connection).
2. **`RunContextWrapper` Class:**
    - `__init__`: Constructor mein, yeh run ke liye khaas maloomat (jaise `request_id`, `current_user_id`) aur `config` store karta hai. Yeh asal services ko initialize nahin karta (yani `_db_connection` aur `_logging_service` shuru mein `None` hote hain).
    - **`@property` methods (`db`, `log`):** Yeh properties "lazy initialization" demonstrate karti hain. `DatabaseConnection` ya `LoggingService` tab tak create nahin hoti jab tak unhein pehli baar access na kiya jaye. Jab access kiya jata hai, to woh context mein ban jate hain aur agle access ke liye re-use hote hain.
    - **`__enter__` aur `__exit__` (Context Manager):** `RunContextWrapper` ko Python context manager banaya gaya hai (using `with` statement).
        - `__enter__` block `with` statement shuru hone par chalta hai. Yahan setup logic aa sakta hai (jaise logging ko shuru karna).
        - `__exit__` block `with` statement khatam hone par chalta hai, chahe koi ghalti ho ya na ho. Yahan cleanup logic aata hai (jaise database connection close karna).
3. **`process_user_request` Function:**
    - Yeh function `RunContextWrapper` object ko parameter ke taur par accept karta hai.
    - Function ke andar, `context.db` aur `context.log` ke zariye database aur logging services ko access kiya jata hai. Function ko is baat ki fikar nahin hai ke yeh services kaise initialize hui hain ya kaise clean up hongi. Yeh sirf unhein istemal karta hai.
4. **Main Application Flow:**
    - **Scenario 1 (Successful):** `with` statement ka istemal karte hue `RunContextWrapper` banaya gaya hai. `process_user_request` function ko `context` object pass kiya gaya hai. Sab kuch theek chalta hai, aur `__exit__` DB connection ko close kar deta hai.
    - **Scenario 2 (Error Handling):** Jab `user_999` ko fetch karne ki koshish ki jati hai, to `process_user_request` mein `ValueError` raise hota hai. `__exit__` block is ghalti ko handle karta hai (log karta hai) aur `with` block ke bahar `try-except` isay pakad leta hai.
    - **Scenario 3 (Multiple Runs):** Har `with` block apne alag `RunContextWrapper` instance ko create karta hai, jis ka matlab hai ke har run ka apna alag context aur resources (maslan, database connection) honge, jo aik doosre se mutasir nahin honge.

Yeh misal wazeh taur par dikhati hai ke **`RunContextWrapper`** kis tarah execution context ko centralize karta hai, resources ko manage karta hai, aur code ko zyada modular aur maintainable banata hai.

Summary ()

`RunContextWrapper` ka buniyadi maqsad complex applications mein kisi bhi "run" (execution) ke **context ko yaksaa tareeqay se faraham karna aur uski aasani se rasai mumkin banana** hai.

Yeh aik object hai jo execution ke liye zaroori tamam relevant maloomat (jaise request ID, user ID), configurations (jaise log level), aur services (jaise database connection, logging service) ko aik hi jagah par jama karta hai.

Iske aham fawaid mein shamil hain:

- **Markazi Rasai (Centralized Access):** Mukhtalif components ko context tak aik hi point se rasai milti hai.
- **Resources ka Behtar Intezaam (Better Resource Management):** Resources (jaise database connections) ko sahi tareeqay se initialize aur clean up karna.
- **Code Ki Safai (Cleaner Code):** Functions ko bahut se parameters pass karne ke bajaye, sirf aik context object pass karna.
- **Behtar Testability (Improved Testability):** Tests mein context ke mock versions banana asaan.

Mukhtasar yeh ke, `RunContextWrapper` aapke code ko zyada structured, robust, aur manage karne mein asaan banata hai, khaas kar ke un scenarios mein jahan execution environment ki details ko mukhtalif hisson mein baantne ki zarurat ho.

4. What does Runner.run_sync() do ?

`Runner.run_sync()`: Executing Synchronous Operations

Definition()

`Runner.run_sync()` ek aisa method hai, jo aam taur par `Runner` ya is jaise kisi orchestrating class se ta'alluq rakhta hai, aur is ka maqsad ek synchronous function ya callable ko execute karna hota hai. Iska bunyadi maqsad aksar blocking code ko aik controlled environment mein run karna hota hai, khaas tor par jab aas paas ka system asynchronous patterns par bana ho.

Explanation ()

`Runner.run_sync()` ko poori tarah samajhne ke liye, aaiye us masle par ghaur karen jo yeh hal karta hai:

- **Asynchronous Contexts:** Aaj kal ki applications aksar asynchronous programming (maslan, Python mein `async`/`await` ka istemal) ka sahara leti hain taa ke performance behtar ho sake. Yeh is tarah hota hai ke jab I/O operations (jaise network requests ya database queries) ka intezar ho raha ho, to doosre tasks chalte rahen.
- **Synchronous Code:** Lekin, boht saara mojooda code, aur yahan tak ke naya code bhi jis mein I/O shamil na ho, fitri taur par synchronous hota hai. Ek synchronous (blocking) function ko seedhe `async` function ke event loop ke andar chalana poore loop ko block kar sakta hai, jis se asynchronicity ke fawaid zaya ho jate hain.
- **The Bridge (Pul):** `Runner.run_sync()` ek pul ka kaam karta hai. Yeh aap ko ek synchronous function ko mehfooz aur asardaar tareeqe se execute karne ki ijazat deta hai. `Runner` ke implementation par depend karte hue, yeh is tarah kar sakta hai:
    - **Running in a Thread Pool:** Sab se aam tareeqa yeh hai ke synchronous function ko ek alag thread mein chalaya jaye (aksar ek thread pool ke zariye manage kiya jata hai). Is se main event loop azad ho jata hai taa ke woh doosre asynchronous tasks ko process kar sake, jabke synchronous function apne alag thread mein chalta rahe.
    - **Direct Execution (agar pehle se synchronous ho):** Agar `Runner.run_sync()` pehle se hi synchronous context se call kiya gaya ho, to yeh function ko seedhe execute kar sakta hai. Yahan "runner" dependency injection, error handling, ya context management jaise doosre fawaid faraham kar sakta hai synchronous call ke gird.
    - **Context Management:** Sirf execution ke ilawa, `Runner.run_sync()` aksar zaroori context setup aur teardown ko bhi handle karta hai, is baat ko yaqeeni banata hai ke resources sahi tareeqe se manage hon aur exceptions sahi tareeqe se propagate hon.

`Runner.run_sync()` ke istemal ke aham scenarios:

1. **Calling Blocking I/O from Async Code:** Agar aap ke paas koi `async` function hai jise kisi aisi library ya API se interact karna hai jo sirf synchronous (blocking) I/O methods provide karti hai (maslan, koi purana database connector, ya file system operation jis ka `async` counterpart na ho), to aap us blocking call ko doosre thread mein offload karne ke liye `Runner.run_sync()` ka istemal karenge.
2. **CPU-Bound Synchronous Tasks:** CPU-intensive synchronous tasks ke liye jo event loop ko block kar denge, `Runner.run_sync()` ka istemal unhein ek alag thread ya process mein chalane ke liye kiya ja sakta hai, jis se main application unresponsive hone se bachti hai.
3. **Integrating Legacy Code:** Jab ek synchronous codebase ko asynchronous framework mein migrate kiya ja raha ho, to `Runner.run_sync()` mojooda synchronous functions ko baghair poori tarah dobara likhe, ahista ahista integrate karne ka ek tareeqa faraham karta hai.

4. **Standardized Execution:** Purely synchronous applications mein bhi, ek `Runner` `run_sync()` faraham kar sakta hai taa ke tamaam core operations ek consistent execution path se guzrain, jis se centralized logging, error handling, ya resource management ki ijazat milti hai.

Example Code

" **Runner** ek tasawarati (conceptual) class hai aur iski implementation **framework** ke hisaab se mukhtalif hoti hai (maslan, `anyio.to_thread.run_sync`, `fastapi.background_tasks`, mukhtalif systems mein custom **Runner** classes), to aayiye hum ek aisi aam (generic) Python misaal se wazahat karein jo wazni tor par **Runner.run_sync()** ke kaam ko zahir karti ho, `asyncio.to_thread.run_sync` ka istemal karte hue (jo is functionality ke liye ek aam pattern hai)."

```
import asyncio
import time
import threading

# Conceptual Runner class (simplified for demonstration)
class MyRunner:
    def __init__(self):
        print(f"MyRunner initialized on thread: {threading.current_thread().name}")

    async def run_sync(self, func, *args, **kwargs):
        """
        Executes a synchronous function in a separate thread.
        In a real framework, this would likely use asyncio.to_thread.run_sync
        or a similar mechanism.
        """
        print(f"[{time.time():.2f}] run_sync called for {func.__name__} on thread:
{threading.current_thread().name}")
        # Simulate running in a thread pool (actual implementation would use
asyncio.to_thread.run_sync)
        result = await asyncio.to_thread(func, *args, **kwargs)
        print(f"[{time.time():.2f}] run_sync finished for {func.__name__} on thread:
{threading.current_thread().name}")
        return result

# --- Synchronous functions ---

def blocking_io_operation(file_path):
    """Simulates a blocking file read operation."""
    print(f"[{time.time():.2f}] Starting blocking I/O for {file_path} on thread:
{threading.current_thread().name}")
    time.sleep(2) # Simulate I/O delay
    content = f"Content from {file_path} (read at {time.time():.2f})"
    print(f"[{time.time():.2f}] Finished blocking I/O for {file_path} on thread:
{threading.current_thread().name}")
    return content

def cpu_bound_task(number):
    """Simulates a CPU-bound calculation."""
    print(f"[{time.time():.2f}] Starting CPU-bound task for {number} on thread:
{threading.current_thread().name}")
    result = sum(i * i for i in range(number * 1000)) # Simulate heavy computation
    print(f"[{time.time():.2f}] Finished CPU-bound task for {number} on thread:
{threading.current_thread().name}")
    return result

# --- Asynchronous function that uses the Runner ---
```

```
async def main():
    runner = MyRunner()

    print(f"\n[{time.time():.2f}] Main coroutine started on thread:
{threading.current_thread().name}")

    # Example 1: Running a blocking I/O operation
    print(f"\n[{time.time():.2f}] Calling blocking_io_operation via run_sync...")
    file_content_task = runner.run_sync(blocking_io_operation, "my_document.txt")
    # We can do other async work while the blocking_io_operation runs
    print(f"[{time.time():.2f}] Doing other async work while I/O is in progress...")
    await asyncio.sleep(0.5) # Simulate other async work

    file_content = await file_content_task
    print(f"[{time.time():.2f}] Received file content: {file_content}\n")

    # Example 2: Running a CPU-bound task
    print(f"[{time.time():.2f}] Calling cpu_bound_task via run_sync...")
    cpu_result_task = runner.run_sync(cpu_bound_task, 500)
    print(f"[{time.time():.2f}] Doing more async work while CPU task is in progress...")
    await asyncio.sleep(0.5)

    cpu_result = await cpu_result_task
    print(f"[{time.time():.2f}] Received CPU result: {cpu_result}\n")

    print(f"[{time.time():.2f}] Main coroutine finished on thread:
{threading.current_thread().name}")

if __name__ == "__main__":
    # Ensure the main thread is named for clarity in output
    threading.current_thread().name = "MainThread"
    asyncio.run(main())
```

Code Example ki Wazahat ():

1. **`MyRunner` Class:** Aik buhat hi saada `MyRunner` class banaya gaya hai, sirf concept dikhane ke liye. Iska `run_sync` method buniyadi hissa hai.
2. **`run_sync` Implementation:** Aham baat yeh hai ke `MyRunner.run_sync` mein `await asyncio.to_thread(func, *args, **kwargs)` ka istemal hota hai. Yeh `asyncio` ka standard tareeqa hai synchronous function ko event loop se alag ek thread mein chalane ka, jis se main asynchronous flow block nahin hota.
3. **`blocking_io_operation` aur `cpu_bound_task`:** Yeh mamooli, synchronous Python functions hain jo `time.sleep()` aur loops ka istemal karte hue blocking I/O aur CPU-intensive kaam ko simulate karte hain. Aap dekh sakte hain ke in ka output inhein doosre thread (aksar `ThreadPoolExecutor` thread) par chalte hue dikhayega.
4. **`main` Coroutine:** Yeh ek `async` function hai jo execution ko organize karta hai.
5. **Non-Blocking Behavior:** Jab `runner.run_sync()` ko call kiya jata hai, to yeh ek awaitable return karta hai. `main` coroutine doosre asynchronous tasks (`await asyncio.sleep(0.5)`) ko execute karna jari rakh sakta hai, jabke synchronous function background thread mein chal raha hota hai. Jab synchronous operation ka result waqai zaroori hota hai tabhi `await` us returned task par `main` coroutine ko block karta hai.
6. **Thread Naming:** `threading.current_thread().name` ko har jagah istemal kiya gaya hai taa ke wazeh taur par dikhaya ja sake ke code ka har hissa kis thread par chal raha hai, aur `run_sync` ki thread-pooling nature ko highlight kiya ja sake. Aap async operations ke liye "MainThread" aur synchronous tasks ke liye "ThreadPoolExecutor" threads dekhenge.

Summary ()

`Runner.run_sync()` asynchronous programming paradigms mein ek nihayat zaroori method hai jo synchronous (blocking) code ko main event loop ko roke baghair, mehfooz aur asardaar tareeqe se execute karne ki ijazat deta hai. Yeh aam taur par synchronous task ko ek alag thread (aksar ek thread pool ke zariye manage kiya jata hai) mein offload karke is maqsad ko haasil karta hai. Yeh pattern darj-e-zel ke liye zaroori hai:

- Legacy synchronous libraries ko asynchronous applications mein integrate karna.
- Blocking I/O operations ko event loop ko block kiye baghair perform karna.
- CPU-bound tasks ko is tarah handle karna ke application responsive rahe.

Asynchronous aur synchronous execution ke darmiyan intiqal (transition) ko manage karne ke liye ek saaf interface faraham karke, `Runner.run_sync()` developers ko mixed synchronous/asynchronous environments mein mazboot, performance-oriented, aur responsive applications banane mein madad karta hai.

5. What does extra= "forbid" do in a Pydantic model config ?

Pydantic Model Config mein `extra="forbid"` ka Kya Matlab Hai?

Definition ()

Pydantic model ki `Config` class mein **`extra="forbid"`** aik setting hai jo Pydantic ko ye batati hai ke agar incoming data (masalan, JSON ya dictionary) mein aise fields mojood hon jo model mein define na kiye gaye hon, to validation process mein error paida kiya jaye. Dusre alfaz mein, ye model ko "strict" banata hai aur sirf un fields ko qabool karta hai jo explicit tareeqay se declare kiye gaye hon.

Explanation ()

Pydantic by default kaafi flexible hota hai jab incoming data ko handle karta hai. Iski `extra` setting Pydantic ko ye batati hai ke agar data mein aise extra fields aa jain jo model mein define na hon, to un ke saath kya karna hai. `extra` ke teen possible values ho sakte hain:

1. **`extra="ignore"` (Default Behavior):** Yeh Pydantic ka default behavior hai. Agar incoming data mein model ke define kiye hue fields ke ilawa koi extra field ho, to Pydantic usko simply ignore kar deta hai (yani, usko validate nahin karta aur na hi final model object mein shamil karta hai). Koi error nahin aata.
    - **Faida:** Jab aapko sirf specific fields ki zarurat ho aur aapko fikar na ho ke baqi data mein kya hai.
    - **Nuqsan:** Agar aapko galat ya ghair-mutawaqqo data aane ka pata chalana ho.
2. **`extra="allow"`:** Is setting mein, Pydantic extra fields ko ignore nahin karta, balkay unko allow kar deta hai aur unhein model object ka hissa bana leta hai, baghair kisi validation ke (yani, unki data type check nahin hoti).
    - **Faida:** Jab aap dynamic data handle kar rahe hon jahan fields pehle se maloom na hon.
    - **Nuqsan:** Data consistency maintain karna mushkil ho sakta hai, aur unexpected data bugs paida kar sakta hai.
3. **`extra="forbid"`:** Yeh woh setting hai jis par hum baat kar rahe hain. Jab `extra="forbid"` set hota hai, to Pydantic incoming data mein kisi bhi aise field ko dekh kar foran `ValidationError` raise kar deta hai jo model mein explicitly define na kiya gaya ho.
    - **Faida:**
        - **Data Integrity (Data ki Durusti):** Yeh yaqeeni banata hai ke aapka application sirf woh data process kare jo aapne expect kiya hai. Agar koi extra ya ghair-zaroori field aa jaye, to aapko foran pata chal jata hai.
        - **Early Bug Detection (Bugs ki Jald Pehchan):** Agar client side se galat field names bheje ja rahe hon, to `forbid` aapko development stage mein hi is masle ka pata chalane mein madad karta hai.
        - **API Strictness (API ki Sakhti):** Jab aap strict APIs design kar rahe hon jahan input format bilkul specific hona chahiye.
    - **Nuqsan:** Agar aapko kabhi-kabhi data mein extra fields ki zarurat ho ya aap dynamic schemas handle kar rahe hon, to ye setting mushkil paida kar sakti hai.

`extra="forbid"` aksar APIs aur data processing pipelines mein use hota hai jahan data schema ki strictness bohot zaroori hoti hai.

Example Code ()

Aaiye examples se dekhte hain ke **extra="forbid"** kaise kaam karta hai.

```
from pydantic import BaseModel, ValidationError

# Case 1: Default behavior (extra="ignore") - jab extra field ignore ho jayega
print("--- Case 1: Default behavior (extra='ignore') ---")
class UserDefault(BaseModel):
    name: str
    age: int

try:
    data1 = {"name": "Alice", "age": 30, "city": "New York"}
    user1 = UserDefault(**data1)
    print(f"UserDefault: {user1.model_dump()}")
    # Output: UserDefault: {'name': 'Alice', 'age': 30}
    # Notice 'city' is ignored
except ValidationError as e:
    print(f"ValidationError: {e}")
print("-" * 40)


# Case 2: extra="allow" - jab extra field allow ho jayega
print("--- Case 2: extra='allow' ---")
class UserAllow(BaseModel):
    name: str
    age: int

    class Config:
        extra = "allow" # Extra fields allowed

try:
    data2 = {"name": "Bob", "age": 25, "country": "Canada"}
    user2 = UserAllow(**data2)
    print(f"UserAllow: {user2.model_dump()}")
    # Output: UserAllow: {'name': 'Bob', 'age': 25, 'country': 'Canada'}
    # 'country' is now part of the model_dump
except ValidationError as e:
    print(f"ValidationError: {e}")
print("-" * 40)


# Case 3: extra="forbid" - jab extra field forbid ho jayega (ERROR hoga)
print("--- Case 3: extra='forbid' ---")
class UserForbid(BaseModel):
    name: str
    age: int

    class Config:
        extra = "forbid" # Extra fields are forbidden

try:
    data3 = {"name": "Charlie", "age": 35, "email": "charlie@example.com"}
    user3 = UserForbid(**data3)
    print(f"UserForbid: {user3.model_dump()}")
except ValidationError as e:
    print(f"ValidationError: {e}")
    # Expected output:
    # ValidationError: 1 validation error for UserForbid
    # extra fields not permitted (type=value_error.extra)
print("-" * 40)
```

```
# Case 4: extra="forbid" with valid data (no extra fields)
print("--- Case 4: extra='forbid' with valid data ---")
class UserForbidValid(BaseModel):

name: str
    age: int

    class Config:
        extra = "forbid"

try:
    data4 = {"name": "David", "age": 40}
    user4 = UserForbidValid(**data4)
    print(f"UserForbidValid: {user4.model_dump()}")
    # Output: UserForbidValid: {'name': 'David', 'age': 40}
    # No error because no extra fields
except ValidationError as e:
    print(f"ValidationError: {e}")
print("-" * 40)
```

Code Example ki Wazahat ():

1. **`UserDefault` (Default Behavior):** Is model mein `Config` define nahin ki gayi, is liye `extra="ignore"` ka default behavior apply hota hai. Jab hum `{ "name": "Alice", "age": 30, "city": "New York" }` jaisa data dete hain, to `"city"` field ko ignore kar diya jata hai aur model object mein sirf `name` aur `age` shamil hote hain. Koi error nahin aata.
2. **`UserAllow` (`extra="allow"`):** Yahan humne explicitly `extra = "allow"` set kiya hai. Jab `{ "name": "Bob", "age": 25, "country": "Canada" }` jaisa data aata hai, to `"country"` field ko bhi model mein shamil kar liya jata hai, baghair kisi validation ke.
3. **`UserForbid` (`extra="forbid"` - Error Case):** Yeh sabse aham example hai. Humne `extra = "forbid"` set kiya hai. Jab hum `{ "name": "Charlie", "age": 35, "email": "charlie@example.com" }` jaisa data dete hain, to `email` field model mein define nahin hai. Is wajah se Pydantic foran `ValidationError` raise karta hai, aur output mein aapko "extra fields not permitted" jaisa error message milega. Model object create nahin hoga.
4. **`UserForbidValid` (`extra="forbid"` - Valid Case):** Is example mein bhi `extra = "forbid"` set hai, lekin incoming data `{ "name": "David", "age": 40 }` mein koi extra field nahin hai. Lehaza, validation kamyab hoti hai aur model object theek se ban jata hai.

Summary ()

Pydantic model config mein `extra="forbid"` aik powerfull setting hai jo aapko data validation par sakht control deti hai. Yeh is baat ko yaqeeni banati hai ke aapka Pydantic model sirf un fields ko accept karega jo explicitly define kiye gaye hain. Agar incoming data mein koi bhi ghair-mutawaqqo ya ghair-zaroori field mojood ho, to `extra="forbid"` foran `ValidationError` generate kar dega.

Iski khaas ahmiyat yeh hai:

- **Data Security aur Durusti:** Ghalat ya malicious data ko rokti hai.
- **API Contracting:** APIs ke input schemas ko sakhti se enforce karti hai.
- **Debugging:** Development ke dauran unexpected data ya typos ki waja se hone wale bugs ko jald pehchanne mein madad karti hai.

Mukhtasar yeh ke, agar aapko apne data schemas mein wazahat aur sakhti chahiye, to `extra="forbid"` aapki Pydantic models ke liye aik behtareen intikhab hai.

6. What's the main advantage of strictschemas over non-strict ?

Strict Schemas ka Buniyadi Faida Non-Strict Schemas Par Kya Hai?

Definition ()

Strict schema data validation ka aik aisa tareeqa hai jahan sirf woh data fields qabool kiye jate hain jo schema mein **explicitly define** kiye gaye hon. Agar incoming data mein koi bhi aisa field mojood ho jo schema mein shamil na ho, to yeh **validation error** paida karta hai.

Iske bar-aks, **non-strict schema** aise extra fields ko ya to **ignore** kar deta hai ya **allow** kar deta hai, baghair kisi error ke.

Explanation ()

`Strict schemas` ka **buniyadi faida** hai:

"Enhanced Data Integrity, Predictability, and Early Error Detection"

(Data ki Durusti, Qaabil-e-Peshgoi Hona, aur Ghalati ki Jald Shinakht mein Izafa)

Iska matlab yeh hai ke **`strict schemas`** aapke data ko zyada bharosemand, mutawaqqo, aur masle ki jald pehchan karne mein madad karte hain. Aaiye is faide ko mazeed tafseel se dekhte hain:

1. **Data Integrity (Data ki Durusti aur Salaahiyat):**
    - **Faida:** Jab aap `strict schema` istemal karte hain, to aap yaqeeni banate hain ke aapke system mein sirf woh data dakhil ho jo aapne expect kiya hai aur jiska aapki application ko matlab samajh aata hai. Yeh data corruption (data ka kharab hona) ya unexpected behavior (ghair-mutawaqqo ravaiya) ko rokti hai.
    - **Non-Strict vs Strict:** Non-strict mein, agar koi ghalti se extra ya galat field (misal ke taur par, `user_name` ki jagah `usr_name`) bheje, to non-strict schema usay ya to ignore kar dega ya allow kar dega. Isse data adhoora ya galat ho sakta hai lekin application ko pata nahin chalega ke kuch ghalat hai. Strict schema foran error dega.
2. **Predictability (Qaabil-e-Peshgoi Hona):**
    - **Faida:** Aapko hamesha pata hoga ke aapke data object mein kya shamil hai aur kya nahin. Jab aap apne code mein model ko load karte hain, to aap confidence ke sath assume kar sakte hain ke is mein sirf define kiye gaye fields honge. Is se code likhna aur maintain karna asaan ho jata hai.
    - **Non-Strict vs Strict:** Non-strict mein, model object mein achanak extra fields aa sakte hain jin ka aapke code ko pata hi nahin. Is se logic mein ghaltiyan paida ho sakti hain jinke liye aapne hisab nahin kiya tha.
3. **Early Error Detection (Ghalati ki Jald Shinakht):**
    - **Faida:** `Strict schemas` client-side ki ghaltiyon ko jald pakadne mein madad karte hain. Agar koi client (maslan, mobile app ya web frontend) galat field name bhej raha hai ya purane schema ke mutabiq data bhej raha hai, to `strict schema` foran `validation error` dega. Yeh development phase mein hi maslon ko highlight karta hai, jisse unhein theek karna asaan hota hai.
    - **Non-Strict vs Strict:** Non-strict mein, aisi ghaltiyan chup rehti hain aur baad mein application logic mein ajeeb o ghareeb bugs paida kar sakti hain jinhein diagnose karna mushkil hota hai.
    -

4. **Clear API Contracts (Wazeh API Muhahide):**
    - **Faida:** Jab aap APIs design karte hain, to `strict schemas` aik wazeh "contract" set karte hain ke client se kis tarah ka data expect kiya jata hai. Is se API documentation behtar hoti hai aur client developers ke liye API ko theek se istemal karna asaan ho jata hai.
    - **Non-Strict vs Strict:** Non-strict APIs "loose" hoti hain aur unke saath kaam karte waqt client developers ko andaza nahin hota ke woh extra data bhej sakte hain ya nahin, ya unka bheja hua data process hoga bhi ya nahin.
5. **Security (Security):**
    - **Faida:** `Strict schemas` aik had tak security bhi faraham karte hain. Woh maliciouly added fields (bad irade se shamil kiye gaye fields) ko rokte hain jinhein application ghalti se process kar sakti hai, ya un fields ko rokte hain jo database mein unwanted columns bana sakte hain agar ORM auto-mapping istemal ho.

Kab non-strict use karna behtar hai?

Non-strict schemas (jaise **`extra="ignore"`** ya **`extra="allow"`**) sirf un scenarios mein useful ho sakte hain jahan aapko data ke ek bade hisse se matlab nahin hai aur aap sirf kuch specific fields extract karna chahte hain (jaise large log files se kuch specific data points), ya jab aap bilkul dynamic data structures ko handle kar rahe hon jahan fields pehle se maloom na hon (jo ke kam hi hota hai). Lekin aam applications aur APIs mein, **strict schemas hamesha tarjeeh hote hain**.

Example Code ()

Pydantic mein **`extra="forbid"`** use karke hum **`strict schema`** ka behavior achieve karte hain. Aaiye iski misal dekhte hain:

```
from pydantic import BaseModel, ValidationError

# --- Strict Schema Example ---
# Hum `Config` mein `extra = "forbid"` set karte hain taake koi bhi extra field allow na ho.
print("--- Strict Schema Example (extra='forbid') ---")
class StrictUser(BaseModel):
    name: str
    age: int
    is_active: bool = True # Default value ke sath bhi defined field hai

    class Config:
        extra = "forbid" # Yeh strictness apply karta hai

# Valid data (all fields defined in schema, no extra)
print("\n--- Strict Schema: Valid Data ---")
try:
    data_valid = {"name": "Ali", "age": 28, "is_active": True}
    user_strict_valid = StrictUser(**data_valid)
    print(f"StrictUser (Valid): {user_strict_valid.model_dump()}")
except ValidationError as e:
    print(f"ValidationError (Valid): {e}")
print("-" * 40)


# Invalid data (extra field 'city')
print("\n--- Strict Schema: Invalid Data (Extra Field) ---")
try:
    data_invalid_extra = {"name": "Fatima", "age": 35, "city": "Lahore"}
    user_strict_invalid_extra = StrictUser(**data_invalid_extra)
    print(f"StrictUser (Invalid Extra): {user_strict_invalid_extra.model_dump()}")
```

```
except ValidationError as e:
    print(f"ValidationError (Invalid Extra): {e}")
    # Output will show 'extra fields not permitted' error
print("-" * 40)

# Invalid data (missing required field 'name') - Strictness ki wajah se yeh bhi catch hoga
print("\n--- Strict Schema: Invalid Data (Missing Required Field) ---")
try:
    data_invalid_missing = {"age": 22, "is_active": False}
    user_strict_invalid_missing = StrictUser(**data_invalid_missing)
    print(f"StrictUser (Invalid Missing): {user_strict_invalid_missing.model_dump()}")
except ValidationError as e:
    print(f"ValidationError (Invalid Missing): {e}")
    # Output will show 'field required' error
print("-" * 40)


# --- Non-Strict Schema Example (Default behavior: extra="ignore") ---
print("\n--- Non-Strict Schema Example (Default: extra='ignore') ---")
class NonStrictUser(BaseModel):
    name: str
    age: int

# Data with an extra field 'country'
print("\n--- Non-Strict Schema: Data with Extra Field ---")
try:
    data_non_strict_extra = {"name": "Zain", "age": 40, "country": "Pakistan"}
    user_non_strict = NonStrictUser(**data_non_strict_extra)
    print(f"NonStrictUser: {user_non_strict.model_dump()}")
    # Output: NonStrictUser: {'name': 'Zain', 'age': 40} -- 'country' is silently ignored
except ValidationError as e:
    print(f"ValidationError (Non-Strict): {e}")
print("-" * 40)
```

Code Example ki Wazahat ():

1. **`StrictUser` Model:** Is model mein humne `Config` class ke andar `extra = "forbid"` set kiya hai.
    - Jab `data_valid` (jis mein koi extra field nahin) ko pass karte hain, to validation kamyab hoti hai aur model object ban jata hai.
    - Jab `data_invalid_extra` (jis mein `"city"` nam ka extra field hai) ko pass karte hain, to Pydantic foran `ValidationError` raise karta hai, yeh batate hue ke "extra fields not permitted".
    - `Strict schemas` sirf extra fields ko hi nahin pakadte, balkay woh required fields ke missing hone par bhi error dete hain, jaisa ke `data_invalid_missing` ke case mein dikhaya gaya hai.
2. **`NonStrictUser` Model:** Is model mein `Config` class ya `extra` setting shamil nahin hai, is liye Pydantic ka default behavior (`extra="ignore"`) apply hota hai.
    - Jab `data_non_strict_extra` (jis mein `"country"` nam ka extra field hai) ko pass karte hain, to validation kamyab hoti hai, lekin `"country"` field ko **khamoshi se ignore** kar diya jata hai. Model object mein sirf `name` aur `age` fields shamil hote hain. Koi error nahin aata.

Yeh example wazeh taur par dikhata hai ke `strict schema` kis tarah extra data ko reject kar ke data ki durusti aur predictability ko yaqeeni banata hai, jabke `non-strict schema` aise data ko allow kar deta hai ya ignore kar deta hai jis se bugs chup sakte hain.

Summary ()

`Strict schemas` ka **`non-strict schemas`** par **buniyadi faida** yeh hai ke woh **behtar data integrity, zyada predictability, aur ghaltiyon ki jald shinakht** faraham karte hain.

`Strict schemas` aapke data input ko aik defined aur expect kiye gaye structure tak mehdood rakhte hain. Isse yeh yaqeeni hota hai ke aapki application sirf sahi, mutawaqqo, aur mozu data par kaam karegi, jabke ghair-zaroori ya galat fields ko foran reject kar diya jayega. Yeh development ke dauran hi maslon ko pakadne mein madad karta hai, code ko maintain karna asaan banata hai, aur aapki APIs ke liye wazeh aur mazboot contracts define karta hai.

Iske bar-aks, non-strict schemas data ke saath zyada lachakdar hote hain, lekin is lachak ki qeemat aksar data ki ghaltiyon ke chhup jane aur application ke ghair-mutawaqqo ravaiye ki shakal mein ada karni padti hai. Is liye, zyada tar real-world applications aur API development mein, **strict schemas hamesha zyada moassar aur pasandeeda tareeqa hain.**

7. What happens when you combine StopAtTools with multiple tool names ?

`StopAtTools` ko Multiple Tool Names ke Sath Combine Karne Par Kya Hota Hai?

Definition ()

Jab aap `StopAtTools` ko **multiple tool names** ke sath combine karte hain, to iska matlab hai ke aap aik workflow ya AI agent ko yeh hidayat de rahe hain ke woh apna amal **rok de** aur control (ya response) wapis kardey jaise hi woh diye gaye tools mein se **kisi bhi aik tool** ko istemal karne ka faisla kare. Yeh "break point" ki tarah kaam karta hai jo workflow ko mazeed agay badhne se rok deta hai jab aik khaas type ka action liya ja raha ho.

Explanation ()

Amoman, AI agents ya complex automation workflows mein steps ki aik series hoti hai. Agent ya workflow aik task ko mukammal karne ke liye mutadid steps leta hai, jis mein tools ka istemal bhi shamil ho sakta hai. `StopAtTools` aik mechanism hai jo is flow ko control karta hai.

Multiple tool names ke sath `StopAtTools` ka bunyadi maqsad:

"Workflow Execution ka Specific Tools ke Istemal Par Rok Dena"

(Kisi Khaas Tools ke Istemal Par Workflow ko Rokna)

Iska matlab hai ke aapka agent ya system aik task par kaam karta rahega, prompts ko process karega, aur zaroorat padne par information generate karega, lekin jaise hi usay aapke diye gaye tools ki list mein se **koi bhi tool** call karna hoga, woh wahin par ruk jayega. Control wapis aa jayega, aur aap ko bataya jayega ke agent ne kaunsa tool istemal karne ka irada kiya hai.

Yeh kyun faidamand hai?

1. **Intermediate State Inspection (Darmiyani Halat ki Jaanch):**
   - **Faida:** Aksar aap kisi complex process ke darmiyan mein dekhna chahte hain ke agent kya karna chahta hai. Multiple `StopAtTools` aapko mukhtalif "checkpoints" set karne ki ijazat deta hai. Maslan, aap chahte hain ke agent database query karne se pehle ruk jaye (tool A), ya koi email bhej ne se pehle ruk jaye (tool B). Agar aapne `StopAtTools = [ToolA, ToolB]` set kiya hai, to jaise hi agent ToolA ya ToolB mein se kisi ko call karega, woh ruk jayega.
   - **Misal:** Agar aapka agent customer ki shikayat hal kar raha hai. Aap chahte hain ke woh customer ko jawab dene se pehle (tool: `send_email`) aur refund process karne se pehle (tool: `process_refund`) ruk jaye taake aap review kar saken.
2. **User Confirmation (Sarif ki Tasdeeq):**
   - **Faida:** Kuch actions aise hote hain jin ke liye user ki tasdeeq zaroori hoti hai (maslan, paise transfer karna, ya kisi external service ko modify karna). `StopAtTools` aapko is maqam par rukne ki ijazat deta hai, user se confirmation leta hai, aur phir agent ko aage badhne deta hai.
   - **Misal:** Agent kehta hai "Mai customer ko refund process karna chahta hoon. Kya aap ijazat dete hain?" (Jab woh `process_refund` tool par ruk gaya ho).
   -

3. **Conditional Logic Application (Mashroot Logic ka Itlaq):**
    - **Faida:** Jab agent kisi tool par ruk jata hai, to aap code mein uski halat ko dekhte hue mukhtalif logic apply kar sakte hain. Maslan, agar agent ne `delete_file` tool call karna chaha hai, to aap mazeed safety checks laga sakte hain ya usey kisi aur function ki taraf redirect kar sakte hain.
    - **Misal:** Agar agent `delete_file` tool par rukta hai, to aap check kar sakte hain ke kya file "critical" hai. Agar hai, to usey delete na karein aur user ko alert dein.
4. **Cost Management (Akhrajaat ka Intezaam):**
    - **Faida:** Baaz tools (maslan, paid APIs) ke istemal par cost aati hai. `StopAtTools` aapko un tools ko call karne se pehle rukne ki ijazat deta hai, jis se aap baghair zaroorat ke expensive operations se bach sakte hain.
    - **Misal:** `StopAtTools = ['expensive_api_call', 'premium_service_tool']`
5. **Debugging (Bugs ki Shinakht):**
    - **Faida:** Development ke dauran, jab aap agent ke flow ko samajhna chahte hain, to mukhtalif tools par breakpoints set kar sakte hain. Is se aap dekh sakte hain ke agent har step par kya soch raha hai aur kaunse tools istemal karne ka irada kar raha hai.

Mukhtalif tool names ke sath `StopAtTools` ka istemal aapko aik fine-grained control faraham karta hai over the execution flow of your AI agent or automated system.

Example Code ()

Chaliye aik hypothetical scenario dekhte hain jahan aik AI assistant user ki inquiry hal kar raha hai.

```
from typing import List, Callable, Any
import time

# --- Hypothetical Tool Definitions ---
# Farz karein yeh aapke tools hain jo agent istemal kar sakta hai
class Tool:
    def __init__(self, name: str, description: str):
        self.name = name
        self.description = description

    def __call__(self, *args, **kwargs):
        print(f"Executing Tool: {self.name} with args={args}, kwargs={kwargs}")
        time.sleep(0.5) # Simulate work
        return f"Result from {self.name}"

# Tools jo agent istemal kar sakta hai
get_weather_tool = Tool("get_weather", "Retrieves current weather for a city.")
send_email_tool = Tool("send_email", "Sends an email to a recipient.")
process_payment_tool = Tool("process_payment", "Processes a financial transaction.")
search_database_tool = Tool("search_database", "Searches the internal database for information.")

all_tools = {
    "get_weather": get_weather_tool,
    "send_email": send_email_tool,
    "process_payment": process_payment_tool,
    "search_database": search_database_tool
}

# --- Hypothetical AI Agent/Workflow System ---
class AIAgent:
    def __init__(self, tools: dict[str, Tool]):
        self.tools = tools
```

```
        self.history = []

    def run_step(self, user_query: str, step: int):
        """Simulates one step of the AI agent's thinking and action."""
        print(f"\n--- Agent Step {step} ---")
        self.history.append(f"User Query: {user_query}")

        # Agent ka logic: Kaunsa tool istemal karna chahiye?
        # Real-world mein yeh LLM ka decision hota
        if "weather" in user_query.lower():
            self.history.append("Agent wants to use: get_weather")
            return {"tool_name": "get_weather", "args": ["London"], "status": "tool_call"}
        elif "email" in user_query.lower():
            self.history.append("Agent wants to use: send_email")
            return {"tool_name": "send_email", "args": ["user@example.com", "Hello!"], "status":
"tool_call"}
        elif "payment" in user_query.lower():
            self.history.append("Agent wants to use: process_payment")
            return {"tool_name": "process_payment", "args": [100.00, "invoice123"], "status":
"tool_call"}
        elif "info" in user_query.lower():
            self.history.append("Agent wants to use: search_database")
            return {"tool_name": "search_database", "args": ["customer_id:456"], "status":
"tool_call"}
        else:
            self.history.append("Agent provides simple response.")
            return {"response": f"I understand your query: '{user_query}'.", "status": "done"}

    def execute_workflow(self, initial_query: str, stop_at_tools: List[str] = None):
        """
        Executes the agent's workflow until a stop_at_tool is encountered or done.
        """
        print(f"Starting workflow for query: '{initial_query}'")
        current_query = initial_query
        step_count = 0

        while True:
            step_count += 1
            decision = self.run_step(current_query, step_count)

            if decision["status"] == "done":
                print(f"[Workflow]: Workflow completed. Response: {decision['response']}")
                break
            elif decision["status"] == "tool_call":
                tool_name = decision["tool_name"]
                tool_args = decision.get("args", [])
                tool_kwargs = decision.get("kwargs", {})

                print(f"[Workflow]: Agent decided to use tool: '{tool_name}'")

                # --- The Core Logic of StopAtTools with multiple names ---
                if stop_at_tools and tool_name in stop_at_tools:
                    print(f"\n[StopAtTools]: Workflow STOPPED at tool '{tool_name}' as
requested.")
                    print(f"   Agent's next intended action: Call '{tool_name}' with
args={tool_args}, kwargs={tool_kwargs}")
                    print("   You can now review or ask for user confirmation.")
                    # Ab yahan aap user se poochenge ya mazeed logic add karenge

                    # For demo, let's just break and simulate user allowing it
                    user_permission = input("Do you want to allow this tool execution? (yes/no):
").lower()
                    if user_permission == 'yes':
                        print("[Workflow]: User allowed. Resuming workflow...")
```

```
                        # Tool ko execute karein aur agle step ke liye result use karein
                        tool_func = self.tools[tool_name]
                        tool_result = tool_func(*tool_args, **tool_kwargs)
                        current_query = f"Tool {tool_name} returned: {tool_result}. What's next?"
# Simulate agent's next input
                        self.history.append(f"Executed Tool: {tool_name}, Result: {tool_result}")
                    else:
                        print("[Workflow]: User denied. Workflow terminated.")
                        break # Workflow roko
                else:
                    # Agar tool stop_at_tools mein nahin hai, to use execute karo
                    print(f"[Workflow]: Executing tool '{tool_name}' (not in stop list).")
                    tool_func = self.tools[tool_name]
                    tool_result = tool_func(*tool_args, **tool_kwargs)
                    current_query = f"Tool {tool_name} returned: {tool_result}. What's next?"
                    self.history.append(f"Executed Tool: {tool_name}, Result: {tool_result}")

            print(f"[Workflow]: Current History: {self.history[-2:]}") # Show last two entries
            if step_count > 5: # Infinite loop se bachne ke liye
                print("[Workflow]: Max steps reached. Terminating.")
                break


# --- Application Usage ---
if __name__ == "__main__":
    agent = AIAgent(tools=all_tools)

    print("===== Scenario 1: Stop at 'send_email' and 'process_payment' =====")
    # User email bhejne ya payment process karne se pehle agent ko rokna chahta hai
    agent.execute_workflow(
        initial_query="I need to send an email about a payment issue.",
        stop_at_tools=["send_email", "process_payment"] # Yahan multiple tool names diye gaye hain
    )
    print("\nWorkflow 1 finished.")

    print("\n" + "="*70 + "\n")

    print("===== Scenario 2: Stop at 'get_weather' only =====")
    # User sirf weather check karne se pehle rokna chahta hai
    agent.execute_workflow(
        initial_query="What's the weather like in London? Also, I need to send an email later.",
        stop_at_tools=["get_weather"] # Yahan sirf aik tool name hai
    )
    print("\nWorkflow 2 finished.")

    print("\n" + "="*70 + "\n")

    print("===== Scenario 3: No specific stop tools =====")
    # Agent freely tools istemal kar sakta hai
    agent.execute_workflow(
        initial_query="I need some info from the database.",
        stop_at_tools=[] # Ya None
    )
    print("\nWorkflow 3 finished.")
```

Code Example ki Wazahat ():

1. **`Tool` Class:** Yeh ek simple class hai jo hamare hypothetical tools (`get_weather`, `send_email`, etc.) ko represent karti hai. Har tool ka aik `name` aur `description` hai, aur woh call hone par kuch simulate karta hai.
2. **`AIAgent` Class:** Yeh hamara hypothetical AI agent ya workflow system hai.
    - `run_step` method simulate karta hai ke agent user ki query ke mutabiq kaunsa tool use karna chahta hai. (Asal mein yeh hissa aik Large Language Model (LLM) handle karta hai.)
    - `execute_workflow` method agent ke flow ko control karta hai.
3. **`stop_at_tools` Logic (Aham Hissa):**
    - `execute_workflow` method ke andar, jab agent koi tool call karne ka faisla karta hai (`decision["status"] == "tool_call"`), to hum check karte hain:

      Python

      ```
      if stop_at_tools and tool_name in stop_at_tools:
      ```

    - Yahan `tool_name in stop_at_tools` check karta hai ke kya agent ka irada shuda tool (maslan, `"send_email"`) hamari `stop_at_tools` list mein mojood hai. Agar woh list mein mojood hai, to workflow **wahin par ruk jata hai**.
    - Output mein aap dekhenge ke "Workflow STOPPED at tool..." message aata hai. Is point par humne user se permission lene ka input (`input("Do you want to allow...?")`) simulate kiya hai. Agar user "yes" kehta hai, to tool execute hota hai aur workflow resume hota hai. Agar "no" kehta hai, to workflow terminate ho jata hai.
    - Agar tool `stop_at_tools` list mein nahin hai, to woh seedha execute ho jata hai aur workflow baghair roke agay badhta hai.

Scenario 1 mein, jab `stop_at_tools=["send_email", "process_payment"]` set kiya gaya hai aur agent `send_email` tool ko call karne ka faisla karta hai, to workflow ruk jata hai.

Scenario 2 mein, `stop_at_tools=["get_weather"]` hai, aur agent `get_weather` ko call karta hai, to woh ruk jata hai. Lekin agar agent `send_email` ko call karega, to woh nahin rukega (kyunki `send_email` stop list mein nahin hai).

Is se wazeh hota hai ke `StopAtTools` ko multiple tool names ke sath combine karne se, workflow un mein se **kisi bhi aik tool** par ruk jata hai, jis se aapko flexibility milti hai ke aap kisi bhi "sensitive" ya "review-worthy" action par control hasil kar saken.

Summary ()

Jab aap `StopAtTools` ko **multiple tool names** ke sath istemal karte hain, to iska buniyadi matlab yeh hai ke aap apne workflow ya AI agent ko yeh hidayat de rahe hain ke woh apna amal **fori taur par rok de** aur control wapis karde **jaise hi woh diye gaye tools ki list mein se kisi bhi aik tool ko istemal karne ka irada kare.**

Iska sab se bada faida yeh hai ke aapko aik complex automation ya AI agent ke flow par **fine-grained control** mil jata hai. Aap darmiyani satah par agent ke iradon ka jaiza le sakte hain, user ki tasdeeq le sakte hain, mushroot logic (conditional logic) ka itlaq kar sakte hain, aur expensive operations ko control kar sakte hain. Yeh security, cost management, debugging, aur behtar user interaction ke liye aik bohot hi mufeed mechanism hai, kyunki yeh agent ko baghair ijazat ke kuch khas actions lene se rokta hai.

8. What is the purpose of context in the OpenAI Agents SDK ?

OpenAI Agents SDK mein Context Ka Maqsad Kya Hai?

Definition ()

Context () OpenAI Agents SDK mein woh tamam relevant maloomat aur data hai jo aik **agent** ko uske maujooda task ko samajhne, munasib faisle lene, aur effective jawab dene ke liye zaroori hota hai. Yeh agent ke liye aik qisam ka "short-term memory" aur "situational awareness" faraham karta hai, jiski bunyad par woh actions perform karta hai.

Explanation ()

Aik AI agent sirf aik bare LLM (Large Language Model) tak mehdood nahin hota. Usay aksar mukhtalif tools (jaise databases, APIs), history, aur specific user preferences jaise cheezon tak rasai ki zaroorat hoti hai taake woh complex tasks ko sahi tareeqe se hal kar sake. Yahan **context** ka kirdar aata hai.

Context ka buniyadi maqsad hai:

"AI Agent Ko Relevant Maloomat Faraham Kar Ke Uske Faisla Karne Aur Amal Karne Ki Salahiyat Ko Behtar Banana"

(Providing Relevant Information to the AI Agent to Enhance Its Decision-Making and Action Capabilities)

Is maqsad ko mazeed tafseel se samajhte hain:

1. **Task Understanding (Task Ko Samajhna):**
    - **Faida:** Context agent ko user ki query ya task ke peechhe ka asal maqsad samajhne mein madad karta hai. Agar user sirf "Kya haal hai?" poochta hai to yeh simple hai, lekin agar user kehta hai "Meri pichli order mein kya update hai?", to agent ko "pichli order" ko samajhne ke liye context ki zaroorat padegi (maslan, user ki ID, pichli orders ki history).
    - Context mein user input, conversation history, aur system state shamil ho sakti hai.
2. **Tool Selection aur Usage (Tools ka Intikhab aur Istemal):**
    - **Faida:** Agents aksar mukhtalif tools (functions) istemal karte hain, maslan, database query karna, email bhejna, ya weather check karna. Context agent ko yeh faisla karne mein madad karta hai ke konsa tool istemal karna hai aur us tool ko sahi tarah se istemal karne ke liye konsi maloomat (arguments) chahiye.
    - Misal ke taur par, agar context mein user ka location hai, to agent weather tool ko us location ke liye call kar sakta hai.
3. **State Management (Halat ka Intezaam):**
    - **Faida:** Lamba conversations (multiturn conversations) mein agent ko pichli interactions ki maloomat yaad rakhne ki zaroorat hoti hai. Context yahi state maintain karta hai. Agar user pehle kuch kehta hai aur phir uske mutalliq mazeed sawal karta hai, to context agent ko pichli baat yaad dilata hai.
    - Is mein session data, user profiles, aur pichle commands ke results shamil hote hain.

4. **Personalization (Infiradiyat):**
    - **Faida:** Context agent ko user ki infiradi zaruriyat aur preferences ke mutabiq jawab dene mein madad karta hai. Maslan, agar context mein user ki pasandeeda zaban shamil hai, to agent us zaban mein jawab de sakta hai.
    - User settings, past interactions, aur profile details context ka hissa ho sakte hain.
5. **Grounding (Buniyad Faraham Karna):**
    - **Faida:** Context agent ke jawab ko "ground" karta hai, yani unhein haqeeqat par mabni banata hai. LLMs mein "hallucination" (galat ya man-gharat maloomat paida karna) ka imkan hota hai. Jab unhein sahi context diya jata hai, to woh zyada durust aur relevant jawab dete hain.
    - Misal: Agar aap agent ko aik document ka context dete hain, to woh usi document se jawab dega bajaye apne general knowledge se.
6. **Constraint Adherence (Pabandiyon Par Amal):**
    - **Faida:** Context mein kuch rules ya constraints bhi shamil ho sakte hain jin par agent ko amal karna hota hai. Maslan, agent ko hidayat di jaye ke woh financial advice na de.

Mukhtasar yeh ke, **context** aik AI agent ke liye woh "environment" aur "information set" banata hai jo usay aik effective aur aqalmand sathi banne mein madad karta hai, taake woh sirf text generate na kare balkay meaningfully interact kar sake.

Example Code ()

OpenAI Agents SDK mein, **context** ko aksar `State` object ya `Thread` ke zariye manage kiya jata hai. Yahin par agent ko woh maloomat milti hain jin ki usay zaroorat hoti hai.

Hum aik simplified (sahoolat-pasand) example dekhte hain ke context kaise kaam karta hai, agar actual SDK implementation zyada pechida ho sakti hai.

```
from openai_agents import Agent, tool # Hypothetical imports
from typing import Dict, Any

# --- Hypothetical Tools ---
# Farz karein yeh aapke tools hain
@tool
def get_user_profile(user_id: str) -> Dict[str, Any]:
    """Retrieves a user's profile information from the database."""
    print(f"\n[Tool Call]: get_user_profile(user_id='{user_id}')")
    # Simulate database lookup
    if user_id == "user_123":
        return {"name": "Ahmed", "email": "ahmed@example.com", "plan": "premium", "location":
"Karachi"}
    return {"error": "User not found"}

@tool
def send_personalized_email(recipient_email: str, subject: str, body: str):
    """Sends a personalized email to the specified recipient."""
    print(f"\n[Tool Call]: send_personalized_email(to='{recipient_email}', subject='{subject}',
body='{body}')")
    return "Email sent successfully!"

# --- Agent Definition ---
class CustomerSupportAgent(Agent):
    def __init__(self, tools, initial_context: Dict[str, Any] = None):
        super().__init__(tools=tools)
```

```
        # Agent ka apna context, jo session ke sath change ho sakta hai
        self.session_context = initial_context if initial_context else {}
        print(f"Agent Initialized with Context: {self.session_context}")

    def handle_message(self, message: str) -> str:
        """
        Agent ka logic jo message ko handle karta hai.
        Context yahan use hota hai decisions lene ke liye.
        """
        print(f"\n--- Agent Handling Message: '{message}' ---")

        # 1. User ki query aur existing context ko istemal karein
        # Asal mein LLM yahan sab kuch process karta
        current_context = {
            "user_query": message,
            **self.session_context # Existing session context ko shamil karein
        }
        print(f"Current Processing Context: {current_context}")

        response = "Sorry, I couldn't understand that."
        tool_to_call = None
        tool_args = ()

        # 2. Context ki bunyad par tool selection ka logic
        if "profile" in message.lower() and "user_id" in current_context:
            print("[Agent Logic]: Decided to get user profile based on context.")
            tool_to_call = "get_user_profile"
            tool_args = (current_context["user_id"],)
        elif "send email" in message.lower() and "email" in current_context.get("user_profile",
{}):
            print("[Agent Logic]: Decided to send email based on context.")
            tool_to_call = "send_personalized_email"
            user_email = current_context["user_profile"]["email"]
            response_body = f"Dear {current_context['user_profile']['name']}, Regarding your
query: '{message}'."
            tool_args = (user_email, "Your Inquiry", response_body)
        elif "hello" in message.lower():
            user_name = current_context.get("user_profile", {}).get("name", "there")
            response = f"Hello {user_name}! How can I assist you today?"
            print("[Agent Logic]: Simple greeting based on context.")
        else:
            print("[Agent Logic]: No specific tool action. Default response.")
            response = f"I received your message: '{message}'. How else can I help?"


        # 3. Agar koi tool call karna hai, to call karein
        if tool_to_call:
            tool_result = None
            if tool_to_call == "get_user_profile":
                tool_result = get_user_profile(*tool_args)
                if "error" not in tool_result:
                    # Tool ka result context mein update karein for future steps
                    self.session_context["user_profile"] = tool_result
                    response = f"I have retrieved the profile for {tool_result['name']}."
                else:
                    response = "Could not find user profile."
            elif tool_to_call == "send_personalized_email":
                tool_result = send_personalized_email(*tool_args)
                response = f"Email sent confirmation: {tool_result}"

            print(f"Updated Session Context: {self.session_context}")

        return response
```

```
# --- Workflow Simulation ---
if __name__ == "__main__":
    # Initial context (maslan, user login ke waqt milta hai)
    initial_user_context = {"user_id": "user_123"}

    # Agent ko tools aur initial context ke sath initialize karein
    agent = CustomerSupportAgent(
        tools=[get_user_profile, send_personalized_email],
        initial_context=initial_user_context
    )

    print("\n--- Scenario 1: Agent uses context to retrieve profile ---")
    # Agent ko "profile" ka keyword milta hai, aur context mein "user_id" hai,
    # to woh `get_user_profile` tool call karta hai.
    agent_response1 = agent.handle_message("Can you tell me about my profile?")
    print(f"Agent's Response: {agent_response1}")

    print("\n--- Scenario 2: Agent uses updated context for personalization and email ---")
    # Ab agent ke paas user_profile context mein hai, woh usay personalized response aur email ke
liye istemal karta hai.
    agent_response2 = agent.handle_message("Hello! And please send email to me about this
conversation.")
    print(f"Agent's Response: {agent_response2}")

    print("\n--- Scenario 3: Agent without enough context for tool ---")
    # Yahan user_id context mein nahin hai, so tool call nahin hoga
    agent_no_context = CustomerSupportAgent(tools=[get_user_profile], initial_context={})
    agent_response3 = agent_no_context.handle_message("Can you tell me about my profile?")
    print(f"Agent's Response: {agent_response3}")
```

Code Example ki Wazahat ():

1. **`get_user_profile` aur `send_personalized_email` Tools:** Yeh mock tools hain jo simulate karte hain ke agent external systems se kaise interact karta hai.
2. **`CustomerSupportAgent` Class:** Yeh hamara agent hai.
    - `self.session_context`: Yeh woh dictionary hai jo agent ke liye **context** ka kaam karti hai. Is mein user ID, user profile jaisi maloomat store ki jaati hain jo conversation ke dauran update ho sakti hain.
    - `handle_message` method: Yeh agent ka "brain" hai.
        - `current_context = {"user_query": message, **self.session_context}`: Yahan user ka naya message aur session ka maujooda context combine kiya jata hai.
        - **Context-driven logic:** Agent ka decision making (maslan, `if "profile" in message.lower() and "user_id" in current_context:`) is `current_context` par mabni hai. Agar context mein `user_id` hai to woh `get_user_profile` tool ko call karne ka sochega.
        - **Context update:** Jab `get_user_profile` tool successfully execute hota hai, to uska result (`user_profile`) `self.session_context` mein update kar diya jata hai. Ab agent ko agle messages ke liye user ka naam aur email pata hoga.
3. **Workflow Simulation (`if __name__ == "__main__":`)**
    - **Scenario 1:** Agent ko `initial_user_context` (jis mein `user_id` hai) diya jata hai. Jab user "profile" poochta hai, to agent context mein `user_id` ko check karta hai aur `get_user_profile` tool call karta hai. Tool ka result context mein save ho jata hai.
    - **Scenario 2:** Agle message mein, agent ab apne update shuda context mein mojood `user_profile` (jis mein naam aur email hai) ka istemal karta hai. "Hello" ke liye woh personalized greeting deta hai, aur "send email" ke liye woh user ke actual email address par email bhejta hai, jo usne context se liya hai.

- **Scenario 3:** Agar agent ko shuru mein `user_id` context mein nahin diya jata, to woh `get_user_profile` tool ko call nahin kar pata, kyunki usay maloomat ki kami hoti hai.

Yeh misal wazeh karti hai ke context kaise agent ko na sirf user ki queries ko behtar tareeqe se samajhne mein madad karta hai, balkay usay previous interactions ki maloomat yaad rakhne, munasib tools call karne, aur personalized jawab dene ke qabil banata hai.

Summary ()

OpenAI Agents SDK mein **context** ka buniyadi maqsad **AI agent ko woh tamam zaruri maloomat faraham karna hai jin ki usay apne task ko samajhne, tools ko sahi tareeqe se istemal karne, aur aqalmandana faisle lene ke liye zaroorat hoti hai.**

Yeh agent ke liye aik qisam ka dinamik "working memory" ka kaam karta hai, jis mein user ki query, conversation history, user preferences, tool ke pichle results, aur system ki maujooda halat jaisi maloomat shamil hoti hain. Context ke baghair, aik AI agent aik "stateless" (halat-se-ghair-mutaliq) entity hoga jo har baar naye siray se har cheez ko samjhega, jis se uski performance, relevancy, aur personalization ki salahiyat buri tarah mutasir hogi.

Mukhtasar yeh ke, **context hi woh cheez hai jo aik AI agent ko sirf aik 'generative model' se 'intelligent and adaptive assistant' mein tabdeel karti hai.**

9. What's the key consideration when choosing between different tool_use_behavior modes ?

Tool_Use_Behavior Modes Ke Darmiyan Intekhab Karte Waqt Buniyadi Ghaur Talab Nuqta Kya Hai?

Definition ()

`tool_use_behavior` modes AI agents (khaas kar ke Large Language Models - LLMs par mabni agents) ke liye woh tareeqe hain jin se woh tools ko istemal karte hain. Yeh modes control karte hain ke agent kitni azadi aur kis had tak tools ko invoke kar sakta hai, ya user ya system se input ya tasdeeq talab kar sakta hai. Mukhtalif SDKs (jaise OpenAI Assistants API, LangChain Agents, ya Microsoft Autogen) mein in modes ke mukhtalif nam aur implementations ho sakte hain, lekin buniyadi concept aik hi rehta hai.

Explanation ()

`tool_use_behavior` modes ke darmiyan intekhab karte waqt sabse **buniyadi ghaur talab nuqta** hai:

"Control vs. Autonomy (Control vs. Khudmukhtari) Ka Tawazun" (Balancing the level of control you want over the agent's actions against the agent's autonomy to perform tasks independently.)

Iska matlab hai ke aapko yeh faisla karna hota hai ke aap agent ko kitni azadi dena chahte hain ke woh tools ko khud-ba-khud istemal kare aur kitna control aap apne paas rakhna chahte hain har tool ke istemal se pehle.

Aaiye is tawazun ko mukhtalif `tool_use_behavior` modes ke zariye samjhte hain:

1. **Fully Autonomous (Mukammal Khudmukhtar) / Auto-Execute Modes:**
    - **Description:** Is mode mein, agent ko tools ko istemal karne ki poori azadi hoti hai, baghair kisi insani mudakhlat ya tasdeeq ke. Jaise hi agent ko lagta hai ke kisi tool ki zarurat hai, woh usay foran call kar deta hai aur uska result istemal karta hai.
    - **Control vs. Autonomy:** Maximum autonomy, minimum control.
    - **Advantages:** Sabse tez execution, user interaction ki kam se kam zarurat.
    - **Disadvantages:** Ghalat ya ghair-mutawaqqo actions ka khatra zyada hota hai (maslan, ghalat email bhej dena, ghalat database entry delete kar dena), debugging mushkil ho sakta hai.
    - **Best for:** Kam khatre wale (low-risk) aur tez-tar execution wale tasks jahan agent ke actions par poora yaqeen ho, ya jahan insani mudakhlat waqt zaya karegi. Misal: Data retrieval, simple calculations, internal logging tools.
2. **Semi-Autonomous (Nim-Khudmukhtar) / Moderated / User-Confirmation Modes:**
    - **Description:** Is mode mein, agent jab bhi kisi tool ko istemal karne ka irada karta hai, to woh pehle user ya system se tasdeeq (confirmation) talab karta hai. Agent "ruke ga" aur batayega ke woh kya tool istemal karna chahta hai aur kyun, phir user ki ijazat milne par hi aage badhega.
    - **Control vs. Autonomy:** Balanced. Moderate control, moderate autonomy.
    - **Advantages:** Actions par zyada control aur safety, ghaltiyon ka khatra kam hota hai, user ko process mein shamil rakha jata hai. Debugging asaan hota hai kyunki aap har tool call se pehle check kar sakte hain.
    - **Disadvantages:** Execution mein waqt zyada lagta hai, user ko har action par input dena pad sakta hai jo pareshan kun ho sakta hai.

- **Best for:** Darmiyani khatre wale (medium-risk) tasks jahan confirmation zaroori ho, ya jahan expensive ya irreversible actions shamil hon. Misal: Email bhejna, financial transactions, data modification, external API calls jo cost karte hain.

3. **Manual / Fully Controlled Modes (Mukammal Control):**
   - **Description:** Is mode mein, agent sirf tool ka naam aur zaroori arguments bata kar "rukh jata" hai. Woh asal tool ko execute nahin karta. Tool ko execute karne ki zimmedari poori tarah user ya code ki hoti hai. Agent sirf next "thinking step" provide karta hai.
   - **Control vs. Autonomy:** Minimum autonomy, maximum control.
   - **Advantages:** Actions par poora control, sabse zyada safety. Developers ya users har tool call ko manual review kar sakte hain. Debugging sabse asaan.
   - **Disadvantages:** Sabse kam speed, user ko har tool call ko manually process karna padta hai, bohot zyada manual effort ki zarurat hoti hai.
   - **Best for:** Bohat zyada khatre wale (high-risk) tasks, sensitive operations, ya development aur debugging ke dauran jab aap agent ke har faisle ko closely monitor karna chahte hain. Misal: System configuration changes, production data deletion, new agent ke trials.

Buniyadi Ghaur Talab Nuqta: Task ka Khatra (Risk Level of the Task)

Apne task ki **risk level** ko samajhna sabse aham hai.

- **Agar ghalti ki gunjaish kam hai (Low Risk):** Jaise weather check karna, aik public database se maloomat hasil karna, ya aik simple calculator tool istemal karna. In cases mein `Fully Autonomous` mode behtar ho sakta hai.
- **Agar ghalti ke serious nataij ho sakte hain (High Risk):** Jaise customer ke account se paise transfer karna, company ki sensitive files delete karna, ya automated email marketing campaign launch karna. In cases mein `Semi-Autonomous` ya `Manual` modes ko tarjeeh deni chahiye.

Doosre aham factors mein **speed ki zarurat**, **user experience (UX)**, aur **debugging ki sahoolat** shamil hain.

Example Code ()

Hum Pydantic aur ek hypothetical `AgentFramework` ka istemal karte hain, jahan **`tool_use_behavior`** ko configure kiya ja sakta hai.

```
from pydantic import BaseModel, Field
from typing import Literal, Dict, Any
import time

# --- Tool Definitions ---
class Tool:
    def __init__(self, name: str, description: str, func: callable):
        self.name = name
        self.description = description
        self.func = func

    def __call__(self, *args, **kwargs):
        print(f"Executing Tool: {self.name} with args={args}, kwargs={kwargs}")
        time.sleep(0.5)
        return self.func(*args, **kwargs)

def get_current_weather(location: str) -> str:
    """Retrieves current weather for a specific location."""
    if location.lower() == "karachi":
```

```
        return "28°C, Sunny"
    return "Unknown weather"

def send_customer_email(recipient: str, subject: str, body: str) -> str:
    """Sends an email to a customer."""
    if "@" not in recipient:
        raise ValueError("Invalid email recipient")
    return f"Email sent to {recipient} with subject '{subject}'"

def delete_database_record(record_id: int) -> str:
    """Deletes a record from the database. IRREVERSIBLE ACTION."""
    if record_id < 100: # Simulate critical records
        raise ValueError("Cannot delete critical record below ID 100")
    return f"Record {record_id} deleted successfully."

# Available Tools
TOOLS = {
    "get_weather": Tool("get_weather", "Get current weather.", get_current_weather),
    "send_email": Tool("send_email", "Send an email to a customer.", send_customer_email),
    "delete_record": Tool("delete_record", "Delete a database record.", delete_database_record),
}

# --- Agent Framework (Hypothetical) ---
class AgentFramework:
    def __init__(self, tools: Dict[str, Tool], tool_use_behavior: Literal["auto", "confirm",
"manual"]):
        self.tools = tools
        self.tool_use_behavior = tool_use_behavior
        print(f"\nAgent Framework Initialized with behavior: '{self.tool_use_behavior}'")

    def _decide_tool(self, user_query: str) -> Dict[str, Any]:
        """
        Simulates agent's decision to use a tool.
        In a real scenario, an LLM would make this decision.
        """
        user_query_lower = user_query.lower()
        if "weather" in user_query_lower:
            return {"tool_name": "get_weather", "args": {"location": "Karachi"}}
        elif "send email" in user_query_lower:
            return {"tool_name": "send_email", "args": {"recipient": "customer@example.com",
"subject": "Update", "body": "Your order is on its way."}}
        elif "delete record" in user_query_lower:
            # Dangerous action for demo
            return {"tool_name": "delete_record", "args": {"record_id": 101}} # Non-critical
record ID
        return {"tool_name": None}

    def process_query(self, user_query: str):
        print(f"\nProcessing query: '{user_query}'")
        tool_decision = self._decide_tool(user_query)
        tool_name = tool_decision.get("tool_name")

        if not tool_name:
            print("Agent: I'm processing your request. No tool needed for now.")
            return

        print(f"Agent wants to use tool: '{tool_name}' with args: {tool_decision.get('args')}")

        if self.tool_use_behavior == "auto":
            print("[Behavior: AUTO]: Executing tool automatically.")
            try:
                tool_func = self.tools[tool_name]
                result = tool_func(**tool_decision["args"])
                print(f"Tool executed. Result: {result}")
```

```
            except Exception as e:
                print(f"Tool execution FAILED: {e}")

        elif self.tool_use_behavior == "confirm":
            print("[Behavior: CONFIRM]: Waiting for user confirmation.")
            confirmation = input(f"Agent wants to execute '{tool_name}'. Allow? (yes/no):
").lower()
            if confirmation == "yes":
                print("User confirmed. Executing tool.")
                try:
                    tool_func = self.tools[tool_name]
                    result = tool_func(**tool_decision["args"])
                    print(f"Tool executed. Result: {result}")
                except Exception as e:
                    print(f"Tool execution FAILED: {e}")
            else:
                print("User denied. Tool execution skipped.")

        elif self.tool_use_behavior == "manual":
            print("[Behavior: MANUAL]: Agent proposed a tool. Manual execution required.")
            print(f"Tool to execute: {tool_name}")
            print(f"Required arguments: {tool_decision.get('args')}")
            print("Please execute this tool manually or provide further instructions.")
            # Yahan system ya user ko khud tool call karna hoga. Agent iske result ka intezar
karega
            # Misal ke liye, hum yahan se ruk jayenge
            pass

# --- Usage Scenarios ---
if __name__ == "__main__":

    # Scenario 1: Auto-execution (Low Risk)
    # Weather check (low risk, usually fine to auto-execute)
    auto_agent = AgentFramework(TOOLS, tool_use_behavior="auto")
    auto_agent.process_query("What's the weather like?")

    # Scenario 2: Confirmation (Medium Risk)
    # Email sending (medium risk, confirmation often desired)
    confirm_agent = AgentFramework(TOOLS, tool_use_behavior="confirm")
    confirm_agent.process_query("Can you send an email to the customer?")

    # Scenario 3: Manual (High Risk)
    # Database record deletion (high risk, manual intervention critical)
    manual_agent = AgentFramework(TOOLS, tool_use_behavior="manual")
    manual_agent.process_query("Delete a database record for me.")

    # Scenario 4: Auto, but with a potentially risky tool (shows why 'auto' is risky for high-risk
tools)
    print("\n--- Scenario 4: Auto-execution of a HIGH-RISK tool (DANGEROUS if not careful) ---")
    dangerous_auto_agent = AgentFramework(TOOLS, tool_use_behavior="auto")
    # Yahan agar record_id 100 se kam hota to error aa jata, aur auto-execute ho jata
    dangerous_auto_agent.process_query("Please delete record 101.")
```

Code Example ki Wazahat ():

1. **Tools (`get_current_weather`, `send_customer_email`, `delete_database_record`):** Yeh mukhtalif risk levels wale tools hain. Weather check low-risk hai, email medium-risk (galat email ja sakta hai), aur database record delete karna high-risk (data loss ho sakta hai).
2. **`AgentFramework` Class:** Yeh hamara hypothetical agent framework hai.
    - `tool_use_behavior`: Yeh woh property hai jo agent ke behavior mode ko define karti hai (`"auto"`, `"confirm"`, `"manual"`).
    - `_decide_tool`: Yeh simulate karta hai ke agent user ki query ke mutabiq konsa tool istemal karna chahta hai.
    - `process_query`: Yeh asal logic hai jo `tool_use_behavior` par mabni hai.
        - **`"auto"`:** Agent tool ko foran execute kar deta hai. Koi rukawat nahin.
        - **`"confirm"`:** Agent tool ka naam batata hai aur user se `input()` ke zariye tasdeeq talab karta hai. Agar "yes" ho to execute karta hai.
        - **`"manual"`:** Agent sirf tool ka naam aur arguments batata hai. Woh tool ko execute nahin karta. Yeh user ya external system par hai ke woh usey kaise handle kare.

Usage Scenarios:

- **Scenario 1 (`auto_agent`):** `get_weather` tool low-risk hai, is liye `"auto"` mode mein yeh seamlessly execute ho jata hai.
- **Scenario 2 (`confirm_agent`):** `send_email` tool medium-risk hai, is liye `"confirm"` mode mein agent email bhejne se pehle user se ijazat talab karta hai. Agar aap "no" type karenge, to email nahin bheja jayega.
- **Scenario 3 (`manual_agent`):** `delete_record` tool high-risk hai, is liye `"manual"` mode mein agent sirf yeh batata hai ke woh kya karna chahta hai, lekin execute nahin karta. Isay manual confirmation aur execution ki zarurat hoti hai.
- **Scenario 4:** Yeh scenario dikhata hai ke agar aap `delete_record` jaise high-risk tool ko `"auto"` mode mein istemal karte hain to kya ho sakta hai. Agar `record_id` aik critical record hota, to yeh baghair ijazat ke delete ho jata!

Is misal se wazeh hota hai ke aap apne `tool_use_behavior` mode ka intekhab kitne ehtiyat se karen, khaas kar ke tools se jude khatre (risk) ko dekhte hue.

Summary ()

`tool_use_behavior` modes ke darmiyan intekhab karte waqt sabse **buniyadi ghaur talab nuqta Control aur Autonomy ke darmiyan sahi tawazun qaim karna** hai.

Yeh is baat par inhisaar karta hai ke aap kitni azadi apne AI agent ko dena chahte hain ke woh tools ko khud-ba-khud (fully autonomously) istemal kare, aur kitna control ya insani mudakhlat aap har tool ke istemal se pehle chahte hain.

- **Agar task ka khatra kam hai** aur aap tez execution chahte hain, to **`Fully Autonomous` (Auto-Execute)** mode behtar hai.
- **Agar task mein darmiyana khatra hai** aur aap safety aur user involvement chahte hain, to **`Semi-Autonomous` (Confirmation)** mode munasib hai.
- **Agar task ka khatra bohot zyada hai** aur aap actions par mukammal control chahte hain, to **`Manual` (Fully Controlled)** mode tarjeeh diya jata hai.

Mukhtasar yeh ke, aapka intikhab aapke system ki zaruriyat, tasks ki sensitivity, aur ghaltiyon ke mumkina nataij par mabni hona chahiye. Sahi mode ka intekhab karna agent ki efficiency, reliability, aur safety ke liye buhat aham hai.

10. What information does handoff_description provide ?

`handoff_description` Kya Maloomat Faraham Karta Hai?

Definition ()

`handoff_description` aik khaas property ya field hai jo **OpenAI Agents SDK** ya is jaisi dusri agent framework mein istemal hoti hai. Iska buniyadi maqsad yeh batana hai ke jab aik AI agent (ya aik automated process) apna task mukammal kar leta hai ya mazeed agay nahin badh sakta, to us task ko ya uske control ko **kis ko (ya kis cheez ko)** aur **kis wajah se** agay (handoff) kiya ja raha hai.

Explanation ()

`handoff_description` ka **buniyadi maqsad** hai:

"Agent ke Amal Ko Insani Mudakhlat Ya Doosre System Tak Muntaqil Karne Ki Wajah Aur Tareeqa Wazeh Karna" (Clarifying the Reason and Method for Transferring Agent's Action to Human Intervention or Another System)

Jab aik AI agent koi kaam kar raha hota hai, to aisa mauqa aa sakta hai jab woh ya to apna kaam poora kar leta hai, ya usay mazeed maloomat ki zarurat hoti hai jo woh khud hasil nahin kar sakta, ya phir usay aik sensitive faisla lena hota hai jis ke liye insani mudakhlat zaroori hoti hai. Aise waqton mein, agent ko apna control "handoff" karna hota hai. `handoff_description` us handoff ki tafseel faraham karta hai.

Yeh kya maloomat deta hai?

1. **Reason for Handoff (Handoff ki Wajah):**
   - Sabse aham baat yeh batata hai ke agent ne kaam kyun roka aur control kyun transfer kiya. Maslan, "User ki account maloomat tak rasai nahin hai," "Task mukammal ho gaya hai," ya "Mazeed tasdeeq ki zarurat hai."
   - Yeh wazahat agent ke faisle ko insaan ya doosre system ke liye qabil-e-fahm banati hai.
2. **Next Steps (Agley Ikdamaat):**
   - Yeh maloomat de sakta hai ke agla qadam kya hona chahiye. Maslan, "Customer support representative se rabta karein," "Customer ko email bhej kar tasdeeq talab karein," ya "Refund process karein."
   - Is se agla amal (human ya automated) asaan ho jata hai.
3. **Required Information for Handoff (Handoff ke Liye Zaruri Maloomat):**
   - Agar handoff ke liye kuch khaas data ki zarurat hai (maslan, user ID, conversation summary, ya agent ne ab tak kya kiya hai), to `handoff_description` mein yeh maloomat bhi shamil ho sakti hain.
   - Yeh is baat ko yaqeeni banata hai ke jis ko handoff kiya gaya hai usay kaam jari rakhne ke liye tamam relevant maloomat milen.
4. **Target of Handoff (Handoff ka Maqsad):**
   - Yeh wazeh kar sakta hai ke control kis entity ko diya ja raha hai. Maslan, "Human Agent (tier 1 support)," "Managerial Review," "External API for approval," ya "Specific automated workflow."

`handoff_description` kyun faidamand hai?

- **Seamless Transitions (Bagair Rukawat ke Tabdeeliyan):** Yeh agent aur insani ya doosre automated system ke darmiyan aik saaf aur wazeh pull banata hai. Is se aisi suratehaal se bacha ja sakta hai jahan agent achanak ruk jaye aur kisi ko pata na chale ke ab kya karna hai.
- **Improved User Experience (Sarif ke Tajurbe mein Behtari):** Jab agent kisi insaan ko handoff karta hai, to `handoff_description` ki wajah se customer ko dobara apni baat dohrani nahin padti, kyunki uski problem ki summary aur context pehle se mojood hoti hai.
- **Efficient Workflow (Kareegar Workflow):** Yeh automated processes ko zyada asardaar banata hai. Jab aik automated system ko pata hota hai ke agent ne control kyun transfer kiya hai, to woh munasib logic ya doosre agent ko trigger kar sakta hai.
- **Debugging aur Monitoring (Ghaltiyon ki Shinakht aur Nigrani):** Development aur deployment ke dauran, `handoff_description` se aap dekh sakte hain ke agent ne kis wajah se aur kis maqam par control transfer kiya, jis se performance ko monitor karna aur ghaltiyon ko theek karna asaan ho jata hai.

Example Code ()

Chaliye, aik hypothetical AI customer support agent ka scenario dekhte hain jo **`handoff_description`** istemal karta hai.

```
from typing import Optional, Dict, Any
import json

# --- Hypothetical Agent State ---
class AgentState:
    def __init__(self, user_id: str, conversation_history: list = None):
        self.user_id = user_id
        self.conversation_history = conversation_history if conversation_history else []
        self.resolved = False
        self.handoff_details: Optional[Dict[str, Any]] = None

    def add_message(self, role: str, content: str):
        self.conversation_history.append({"role": role, "content": content})

    def set_handoff(self, reason: str, next_steps: str, target: str, summary: str = ""):
        """
        Sets the handoff_description with relevant information.
        """
        self.handoff_details = {
            "reason": reason,
            "next_steps": next_steps,
            "target": target,
            "summary": summary if summary else "Agent could not fully resolve the query."
        }
        self.resolved = False # Not resolved by agent, handed off

# --- Hypothetical AI Agent ---
class CustomerServiceAgent:
    def __init__(self, agent_id: str):
        self.agent_id = agent_id
        self.known_issues = ["billing problem", "technical issue"]
        self.max_auto_attempts = 2

    def process_query(self, query: str, state: AgentState) -> str:
        state.add_message("user", query)
        print(f"\n--- Agent {self.agent_id} Processing: '{query}' ---")

        # Simulate agent's attempts to resolve
```

```
        current_attempts = len([m for m in state.conversation_history if m['role'] == 'agent' and
"trying to help" in m['content'].lower()])

        if "hello" in query.lower():
            state.add_message("agent", "Hello! How can I assist you today?")
            return "Greeting."
        elif any(issue in query.lower() for issue in self.known_issues) and current_attempts <
self.max_auto_attempts:
            state.add_message("agent", "I understand you have a " + query.lower() + ". I'm trying
to help you with that.")
            print(f"Agent attempting to resolve known issue. Attempts: {current_attempts + 1}")
            # Simulate some internal processing which might fail or need more info
            if "billing problem" in query.lower() and current_attempts == 0:
                return "Attempting to retrieve billing info..."
            else:
                 # Handoff if attempts max out or specific query
                 state.set_handoff(
                    reason="Could not fully resolve the specific billing/technical issue
automatically.",
                    next_steps="Please transfer to a specialized human agent for 'Billing
Department' or 'Technical Support'.",
                    target="Human Agent (Specialized Support)",
                    summary=f"User reported a '{query}'. Agent tried {current_attempts + 1} times
but needs human expertise."
                )
                 state.add_message("agent", "I need to hand you off to a human expert for this
issue.")
                 return "Handoff triggered."
        elif "thank you" in query.lower() or "bye" in query.lower():
            state.resolved = True
            state.add_message("agent", "You're welcome! Goodbye.")
            return "Resolved. Conversation ended."
        else:
            # General query, if not resolved and attempts exceed, handoff
            if current_attempts >= self.max_auto_attempts:
                 state.set_handoff(
                    reason="Agent reached maximum attempts to understand/resolve a general
query.",
                    next_steps="Please escalate to a general human support agent.",
                    target="Human Agent (General Support)",
                    summary=f"User query: '{query}'. Agent could not proceed after multiple
attempts."
                )
                 state.add_message("agent", "I apologize, I'm unable to assist further. I need to
hand you off to a human agent.")
                 return "Handoff triggered."
            else:
                state.add_message("agent", "I'm not sure how to help with that. Can you please
rephrase or provide more details?")
                return "Awaiting rephrase."

# --- Handoff System (Hypothetical) ---
def handle_handoff(state: AgentState):
    """
    This function processes the handoff details provided by the agent.
    In a real system, this would trigger notifications, create tickets, etc.
    """
    if state.handoff_details:
        print("\n=== Handoff Initiated! ===")
        print(f"  Reason: {state.handoff_details['reason']}")
        print(f"  Next Steps: {state.handoff_details['next_steps']}")
        print(f"  Target: {state.handoff_details['target']}")
        print(f"  Summary: {state.handoff_details['summary']}")
        print(f"  Conversation History: {json.dumps(state.conversation_history, indent=2)}")
```

```
        print("========================")
    else:
        print("\nNo handoff required or details not set.")


# --- Simulation ---
if __name__ == "__main__":
    print("--- Scenario 1: Agent tries to resolve, then hands off for a known issue ---")
    user_state1 = AgentState(user_id="user_001")
    agent1 = CustomerServiceAgent(agent_id="CS_Agent_Alpha")

    agent1.process_query("I have a billing problem.", user_state1)
    print(f"Agent's last response: {user_state1.conversation_history[-1]['content']}")

    # Simulate another attempt by the user/system
    agent1.process_query("Yes, it's about my last invoice.", user_state1)
    print(f"Agent's last response: {user_state1.conversation_history[-1]['content']}")

    handle_handoff(user_state1) # Check handoff details

    print("\n" + "="*60 + "\n")

    print("--- Scenario 2: Agent hands off after max general attempts ---")
    user_state2 = AgentState(user_id="user_002")
    agent2 = CustomerServiceAgent(agent_id="CS_Agent_Beta")
    agent2.max_auto_attempts = 1 # Set lower for quick demo handoff

    agent2.process_query("I need some help with my account.", user_state2)
    print(f"Agent's last response: {user_state2.conversation_history[-1]['content']}")

    agent2.process_query("It's about changing my plan.", user_state2)
    print(f"Agent's last response: {user_state2.conversation_history[-1]['content']}")

    handle_handoff(user_state2) # Check handoff details

    print("\n" + "="*60 + "\n")

    print("--- Scenario 3: Agent resolves, no handoff ---")
    user_state3 = AgentState(user_id="user_003")
    agent3 = CustomerServiceAgent(agent_id="CS_Agent_Gamma")

    agent3.process_query("Hello!", user_state3)
    agent3.process_query("Thank you, bye!", user_state3)
    handle_handoff(user_state3) # Check handoff details
```

Code Example ki Wazahat ():

1. **`AgentState` Class:** Yeh user ki conversation state ko maintain karti hai, jis mein `conversation_history` aur sabse aham, `handoff_details` shamil hain.
    - `set_handoff` method: Yahan hum `handoff_description` ki maloomat (reason, next_steps, target, summary) ko `handoff_details` dictionary mein store kar rahe hain.
2. **`CustomerServiceAgent` Class:** Yeh hamara AI agent hai jo user queries ko process karta hai.
    - Ismein kuch `known_issues` hain aur `max_auto_attempts` ki limit hai.
    - `process_query` method: Is method mein agent ki logic hai.
        - Agar agent kisi `known_issue` ko `max_auto_attempts` tak hal nahin kar pata (Scenario 1), ya agar woh general query ko `max_auto_attempts` tak nahin samajh pata (Scenario 2), to woh `state.set_handoff()` ko call karke `handoff_description` tayar karta hai.
        - Notice karein ke `handoff_description` mein **wajah (reason)**, **agley ikdamaat (next_steps)**, **kis ko handoff karna hai (target)**, aur **summary** di ja rahi hai.

3. **`handle_handoff` Function:** Yeh hypothetical system ka hissa hai jo `handoff_details` ko process karta hai. Jab `state.handoff_details` set hota hai, to yeh maloomat print kar di jaati hain. Aik asal system mein yeh maloomat kisi human agent ke dashboard par dikhai jati hain ya aik naya support ticket banati hain.

Simulation Scenarios:

- **Scenario 1:** Agent `billing problem` ko aik attempt ke baad hal nahin kar pata, to woh `handoff_description` set karta hai jis mein wazeh kiya gaya hai ke yeh "specialized human agent" ko transfer kiya jana chahiye.
- **Scenario 2:** Agent general query ko 1 attempt ke baad hal nahin kar pata, to woh `handoff_description` set karta hai jis mein wazeh kiya gaya hai ke yeh "general human support agent" ko transfer kiya jana chahiye.
- **Scenario 3:** Agent user ki query ko hal kar deta hai ("Thank you, bye!"), is liye koi `handoff_description` set nahin hota.

Yeh misal wazeh taur par dikhati hai ke `handoff_description` kis tarah zaroori maloomat faraham karta hai takay aik kam ya na-mukammal kaam ko behtar tareeqe se agay (handoff) kiya ja sake.

Summary ()

`handoff_description` ka buniyadi maqsad yeh hai ke jab aik AI agent (ya automated process) apni zimmedari poori kar le ya mazeed agay na badh sake, to us task ya control ko **kis wajah se** aur **kis ko** agay (handoff) kiya ja raha hai, is ki **wazeh aur mufassal maloomat** faraham karna.

Yeh maloomat aam taur par shamil hoti hain:

- **Handoff ki wajah (Reason):** Agent ne kyun roka.
- **Agley ikdamaat (Next Steps):** Agla qadam kya hona chahiye.
- **Handoff ka maqsad (Target):** Control kis ko transfer kiya ja raha hai (maslan, insani support agent, doosra system).
- **Mukhtasar Khulasa (Summary):** Agent ne ab tak kya kiya aur iski maujooda halat kya hai.

`handoff_description` ka istemal workflows ko behtar, users ke tajurbe ko behtar, aur debugging ko asaan banata hai, kyunki yeh agent aur human ya doosre automated systems ke darmiyan aik saaf aur effective communication channel banata hai.

11.What does ToolsToFinalOutputResult.is_final_output indicate ?

`ToolsToFinalOutputResult.is_final_output` Kya Zahir Karta Hai?

Definition ()

`ToolsToFinalOutputResult.is_final_output` aik **boolean flag** (yaani `True` ya `False` value) hai jo OpenAI Agents SDK ya is jaisi dusri agent frameworks mein istemal hota hai. Yeh is baat ki nishandahi karta hai ke agent ne apne mukammal amal ke baad jo output paida kiya hai, woh **task ka aakhri aur mutawwaqo natija (final result)** hai, aur mazeed koi processing, tool calls, ya user interaction ki zarurat nahin hai.

Explanation ()

Jab aik AI agent koi task perform karta hai, to woh aksar steps ki aik series se guzarta hai. Ismein mukhtalif tools ka istemal shamil ho sakta hai, data processing, aur user se sawal-jawab. Har step ke baad agent aik `output` paida karta hai. `ToolsToFinalOutputResult.is_final_output` ka maqsad is output ki "finality" (mukammal hone ki haalat) ko batana hai.

`ToolsToFinalOutputResult.is_final_output` ka **buniyadi maqsad** hai:

"Agent Ke Output Ki Task Ke Lihaz Se Aakhri Halat Ki Wazahat Karna"

(To Clarify the Finality of the Agent's Output with Respect to the Task)

Iska matlab hai ke yeh flag system ko (aur insaan ko) batata hai ke:

1. **Task Completion (Task Ki Takmeel):**
    - **Faida:** Agar `is_final_output` **`True`** hai, to iska matlab hai ke agent ne apna maqsad poora kar liya hai. Ab system is output ko direct user tak pohuncha sakta hai ya doosre downstream systems (jo agent ke output par munhasir hain) ko de sakta hai, baghair mazeed validation ya processing ke.
    - **Misal:** Agar agent ne user ke sawal ka sahi jawab dhoond liya hai, ya aik request ko safalta se process kar liya hai, to `is_final_output` `True` hoga.
2. **No Further Action Required (Mazeed Amal Ki Zarurat Nahin):**
    - **Faida:** Yeh flag signals deta hai ke agent ki taraf se mazeed koi tool call, koi naya thought process, ya user se koi mazeed sawal poochhne ki zarurat nahin hai. Agent apni zimmedariyan mukammal kar chuka hai.
    - **Misal:** Agar `is_final_output` `True` hai, to aap agent ke session ko khatam kar sakte hain ya user ko "Task Complete" ka message dikha sakte hain.
3. **Distinguishing Intermediate from Final (Darmiyani Aur Aakhri Nataij Mein Farq):**
    - **Faida:** Bahut se workflows mein, agent darmiyani (intermediate) outputs bhi paida karta hai (maslan, "Mai database search kar raha hoon," ya "Mujhe yeh maloomat mili hai, ab mai isay analyze karunga"). `is_final_output` in darmiyani outputs ko **aakhri, action-able output** se alag karta hai. System in darmiyani outputs ko users tak direct nahin pohunchata, balkay unhein internaly use karta hai.
    - **Misal:** Agent pehle weather tool call karta hai, phir us data ko analyze karta hai, aur akhir mein user ko aik summarized jawab deta hai. Weather tool ka result intermediate hai, summarized jawab final.

4. **Workflow Control (Workflow Ka Control):**
    - **Faida:** Ye flag system ke overall workflow ko control karne mein madad karta hai. System ko pata hota hai ke kab agent ka role khatam ho gaya hai aur kab next step (maslan, user ko jawab dena, ya aik aur agent ko task handoff karna) liya ja sakta hai.
    - **Misal:** Agar `is_final_output` `True` nahin hai, to system agent ko mazeed steps lene ki ijazat dega (ya user se mazeed input maangega).

Jab `is_final_output` `False` ho to kya hota hai?

Agar **`is_final_output`** `False` hai, to iska matlab hai ke agent ka kaam abhi mukammal nahin hua. Agent mazeed tool calls kar sakta hai, mazeed thinking steps le sakta hai, ya user se mazeed maloomat talab kar sakta hai. System ko is output ko final samajh kar aage nahin badhna chahiye.

Example Code ()

Chaliye aik hypothetical agent ki misal dekhte hain jo **`is_final_output`** flag ka istemal karta hai.

```
from typing import Dict, Any, Literal, NamedTuple

# Hypothetical result structure from agent's processing
class ToolsToFinalOutputResult(NamedTuple):
    # This represents what the agent outputs after a step or entire process
    output_message: str
    is_final_output: bool  # The flag we are discussing
    next_action_suggestion: Optional[str] = None # For demo purposes

# --- Hypothetical Tool ---
def get_stock_price(symbol: str) -> float:
    """Retrieves the current stock price for a given symbol."""
    print(f"\n[Tool]: Getting stock price for {symbol}...")
    if symbol.upper() == "AAPL":
        return 180.50
    elif symbol.upper() == "GOOG":
        return 1500.25
    return 0.0

# --- Hypothetical AI Agent ---
class FinancialAgent:
    def __init__(self, agent_name: str):
        self.agent_name = agent_name
        self.conversation_steps = 0
        self.tools = {"get_stock_price": get_stock_price}

    def process_query(self, query: str) -> ToolsToFinalOutputResult:
        self.conversation_steps += 1
        print(f"\n--- Agent {self.agent_name} processing step {self.conversation_steps} for query:
'{query}' ---")

        # Scenario 1: Needs a tool call, not final yet
        if "stock price of" in query.lower() and self.conversation_steps == 1:
            symbol = query.lower().split("stock price of")[-1].strip().upper()
            if symbol:
                print(f"Agent: Decided to use 'get_stock_price' tool for {symbol}.")
                # In a real SDK, the agent would just suggest this tool call,
                # and the framework would execute it, then provide the result back to the agent.
                # For this demo, we simulate direct tool call and immediate response.
```

```
                price = self.tools["get_stock_price"](symbol)

                # This is an intermediate result from agent's perspective.
                # Agent still needs to format it or ask clarification.
                if price > 0:
                    return ToolsToFinalOutputResult(
                        output_message=f"I found the stock price for {symbol}: ${price}. Would you
like more details?",
                        is_final_output=False, # NOT FINAL - Agent expects more interaction
                        next_action_suggestion="Ask user for more details or provide related
info."
                    )
                else:
                    return ToolsToFinalOutputResult(
                        output_message=f"Could not find stock price for {symbol}. Is the symbol
correct?",
                        is_final_output=False, # NOT FINAL - Agent expects user to rephrase
                        next_action_suggestion="Ask user to rephrase query."
                    )

        # Scenario 2: Final output after follow-up
        elif "no, that's all" in query.lower() or "thank you" in query.lower():
            return ToolsToFinalOutputResult(
                output_message="You're welcome! Feel free to ask if you have more questions.",
                is_final_output=True, # FINAL - Conversation ends
                next_action_suggestion="End session."
            )

        # Scenario 3: Directly final output (no tool or follow-up needed)
        elif "hello" in query.lower():
            return ToolsToFinalOutputResult(
                output_message="Hello there! How can I help you today?",
                is_final_output=True, # FINAL - Simple greeting, no follow-up expected
                next_action_suggestion="None (already final)."
            )

        # Default intermediate output
        return ToolsToFinalOutputResult(
            output_message="I'm still processing your request or need more information. Can you
elaborate?",
            is_final_output=False, # NOT FINAL
            next_action_suggestion="Ask for clarification."
        )

# --- Simulation ---
if __name__ == "__main__":
    agent = FinancialAgent("FinBot")
    current_query = "What's the stock price of AAPL?"

    print("--- Scenario 1: Agent provides intermediate output, awaiting follow-up ---")
    result1 = agent.process_query(current_query)
    print(f"Agent's Output: '{result1.output_message}'")
    print(f"Is Final Output? {result1.is_final_output}")
    print(f"Next Suggested Action: {result1.next_action_suggestion}")

    if not result1.is_final_output:
        print("\n[System]: Agent's output is NOT final. Waiting for user's next input.")
        user_follow_up = input("User (responding to agent): ")

        result2 = agent.process_query(user_follow_up)
        print(f"Agent's Output: '{result2.output_message}'")
        print(f"Is Final Output? {result2.is_final_output}")
        print(f"Next Suggested Action: {result2.next_action_suggestion}")
```

```
        if result2.is_final_output:
            print("[System]: Agent's output IS final. Task completed.")
        else:
            print("[System]: Agent's output is still NOT final. Something went wrong or needs more
info.")

    print("\n" + "="*60 + "\n")

    print("--- Scenario 2: Agent provides direct final output ---")
    result3 = agent.process_query("Hello!")
    print(f"Agent's Output: '{result3.output_message}'")
    print(f"Is Final Output? {result3.is_final_output}")
    print(f"Next Suggested Action: {result3.next_action_suggestion}")

    if result3.is_final_output:
        print("[System]: Agent's output IS final. Task completed.")
    else:
        print("[System]: Agent's output is NOT final. Needs more interaction.")

    print("\n" + "="*60 + "\n")

    print("--- Scenario 3: Agent can't find stock ---")
    result4 = agent.process_query("What's the stock price of XYZ?")
    print(f"Agent's Output: '{result4.output_message}'")
    print(f"Is Final Output? {result4.is_final_output}")
    print(f"Next Suggested Action: {result4.next_action_suggestion}")
    if not result4.is_final_output:
        print("[System]: Agent's output is NOT final. User might rephrase.")
```

Code Example ki Wazahat ():

1. **`ToolsToFinalOutputResult` (NamedTuple):** Yeh class represent karti hai woh information jo agent apne har processing step ke baad deta hai. Is mein `output_message` ke sath, sabse aham `is_final_output` boolean flag hai. `next_action_suggestion` sirf demo ke liye hai.
2. **`get_stock_price` (Tool):** Yeh ek simple tool hai jo stock price dhoondhta hai.
3. **`FinancialAgent` Class:**
    - `process_query` method agent ke "brain" ko simulate karta hai.
    - **Scenario 1 (`"stock price of"` query):** Jab user stock price poochhta hai, to agent `get_stock_price` tool ko call karta hai. Tool ka result milne ke baad, agent user ko batata hai ke usay price mil gaya hai, **lekin `is_final_output` ko `False` set karta hai**. Kyunki agent ko abhi bhi lagta hai ke user mazeed details ya follow-up sawal kar sakta hai (maslan, "would you like more details?"). Yahan system ko pata chalta hai ke conversation abhi jari rahegi.
    - **Scenario 2 (`"thank you"` query):** Jab user "thank you" kehta hai, to agent samajh jata hai ke conversation khatam ho gayi hai. Is case mein, `is_final_output` ko **`True`** set kiya jata hai, yeh batane ke liye ke agent ka kaam mukammal ho gaya.
    - **Scenario 3 (`"hello"` query):** Simple "Hello" ka jawab bhi agent ke liye final output ho sakta hai, kyunki is mein mazeed interaction ya tool call ki zarurat nahin.
4. **Simulation (`if __name__ == "__main__":`)**
    - System `is_final_output` flag ko check karta hai.
    - Agar `False` hai, to system user se mazeed input ka intezar karta hai.
    - Agar `True` hai, to system samajh jata hai ke agent ka task mukammal ho gaya hai aur woh conversation ko end kar sakta hai ya agley process ko trigger kar sakta hai.

Summary ()

`ToolsToFinalOutputResult.is_final_output` aik **boolean flag** hai jo wazeh taur par zahir karta hai ke agent ne jo output paida kiya hai, woh **maujooda task ka aakhri aur mukammal natija (final result)** hai, aur mazeed kisi action, processing, ya interaction ki zarurat nahin hai.

Iska buniyadi maqsad yeh hai:

- **Task ki takmeel ka ishara dena:** System ko batana ke agent ne apna kaam poora kar liya hai.
- **Workflow control:** System ko allow karna ke woh aakhri output ko user tak pohunchaye ya agley steps shuru kare (maslan, conversation end karna, ya ticket close karna).
- **Darmiyani aur aakhri nataij mein farq:** Intermediate outputs (jo mazeed processing ya user input ki talabgar hon) ko final outputs se alag karna.

Jab `is_final_output` `True` ho, to system samajh jata hai ke agent ka role is task ke liye khatam ho gaya hai. Jab yeh `False` ho, to system ko pata hota hai ke agent abhi bhi task par kaam kar raha hai aur mazeed interaction ki ummeed hai.

12. What's the difference between hosted tools and function tools in terms of tool_use_behavior ?

Hosted Tools aur Function Tools Mein `tool_use_behavior` Ke Lehaz Se Kya Farq Hai?

Definition ()

Hosted Tools (Host Kiye Gaye Tools) woh tools hote hain jin ki functionality kisi external service ya platform par mojood hoti hai aur agent sirf unhein network request (jaise API call) ke zariye access karta hai. Agent ko un tools ka andaruni code ya logic tak direct rasai nahin hoti.

Function Tools (Function Tools) woh tools hote hain jin ki functionality agent ke apne environment (ya usi code execution context) mein available hoti hai, aam taur par Python functions ke roop mein. Agent unhein directly call kar sakta hai jaise woh kisi doosre Python function ko call karta hai.

Explanation ()

`tool_use_behavior` ke lehaaz se in do qism ke tools mein buniyadi farq **"Execution Context aur Control Flow"** ka hai.

"Hosted tools ka control flow network-based hota hai, jabke function tools ka control flow local aur direct hota hai, jis se tool_use_behavior mein farq aata hai."

(The control flow for hosted tools is network-based, while for function tools it's local and direct, which impacts their tool_use_behavior.)

Aaiye in dono mein farq ko tafseel se dekhte hain:

Hosted Tools (Host Kiye Gaye Tools)

- **Jise Agent Dekhta Hai:** Agent ko sirf tool ka naam, description, aur zaroori parameters (API schema/signature) bataya jata hai. Usay is baat se matlab nahin hota ke yeh tool pichhe kaise kaam karta hai.
- **`tool_use_behavior` Par Asar:**
    1. **Execution Model (Execution Model):** Execution hamesha network-based hoti hai. Jab agent hosted tool ko istemal karne ka faisla karta hai, to system aik HTTP request (GET, POST, etc.) us external service ko bhejta hai jahan tool hosted hai. Response wapas network se aata hai.
    2. **Latency (Dairi):** Network latency ki wajah se response mein waqt lag sakta hai. `tool_use_behavior` modes (jaise `auto`, `confirm`) is latency ko manage karne mein madad kar sakte hain. Agar `auto` mode hai, to user ko lambe intezar ka saamna karna pad sakta hai.
    3. **Error Handling (Ghalti ki Shinakht):** Errors network-related ho sakte hain (maslan, connection lost, timeout), HTTP status codes (404, 500), ya API-specific errors. `tool_use_behavior` ka `confirm` mode aisi ghaltiyon ko hone se pehle user ko mauqa de sakta hai ke woh action ko review kare.
    4. **Security/Permissions (Security/Ijazat):** Hosted tools tak rasai ke liye aksar API keys, tokens, ya OAuth flows ki zarurat hoti hai. Iska matlab hai ke har tool call par authentication aur authorization checks hote hain. `tool_use_behavior` ke zariye user confirmation lekar sensitive hosted tools ko unauthorized access se bachaya ja sakta hai.
    5. **Side Effects (Side Effects):** Hosted tools ke aksar "side effects" hote hain (maslan, database mein data change karna, email bhejna, payment process karna) jo external systems par asar andaz hote hain. Is wajah se, `tool_use_behavior` mein **`confirm`** ya **`manual`** modes ko zyada tarjeeh di jati hai taake in side effects ko control kiya ja sake.
- **Misal:** Weather API, Google Maps API, Stripe Payment Gateway, SendGrid Email Service.

Function Tools (Function Tools)

- **Jise Agent Dekhta Hai:** Agent ko tool ka naam, description, parameters, aur woh function milta hai jise woh call kar sakta hai. Iski functionality agent ke code environment mein hi mojood hoti hai.
- **`tool_use_behavior` Par Asar:**
    1. **Execution Model (Execution Model):** Execution local aur direct hoti hai. Jab agent function tool ko istemal karne ka faisla karta hai, to system sirf us Python function ko directly call karta hai (jaise `my_function(arg1, arg2)`).
    2. **Latency (Dairi):** Latency aam taur par kam hoti hai, sirf function ki computational complexity par munhasir hai. Network latency nahin hoti. Is wajah se, `auto` `tool_use_behavior` mode zyada istemal kiya ja sakta hai kyunki response waqt kam hota hai.
    3. **Error Handling (Ghalti ki Shinakht):** Errors aam taur par code-related hotay hain (Python exceptions). Debugging asaan hota hai kyunki control local code base mein rehta hai. `confirm` mode yahan bhi istemal ho sakta hai lekin zaroorat kam hoti hai, sivaay high-risk local operations ke.
    4. **Security/Permissions (Security/Ijazat):** Security local environment (operating system permissions, code access) par munhasir hoti hai. Agar agent ka code environment secure hai, to function tools bhi secure hain.
    5. **Side Effects (Side Effects):** Function tools ke bhi side effects ho sakte hain (jaise local file system par data likhna/delete karna, in-memory data structures change karna). Agar yeh side effects sensitive hon, to `tool_use_behavior` mein `confirm` ya `manual` modes istemal kiye ja sakte hain.
- **Misal:** Python function jo memory mein data ko sort karta hai, aik local file ko read karta hai, ya aik complicated calculation perform karta hai.

Summary ()

Hosted tools aur function tools mein **`tool_use_behavior`** ke lehaaz se buniyadi farq unke **Execution Context aur Control Flow** mein hai.

Feature	Hosted Tools	Function Tools
Execution Context	External service / Network API	Local code environment (Python function call)
Control Flow	Network requests (HTTP/RPC), then external processing, then network response	Direct local function call
Latency	High (network delay, server processing)	Low (local computation, no network)
Error Types	Network errors, HTTP errors, API-specific errors	Code errors (Python exceptions), computational errors
Security/Permissions	API keys, OAuth, external service policies	Local system/code permissions
Typical Behavior Mode	Often `confirm` or `manual` (due to risk, latency, cost)	Often `auto` (due to speed, local control)
Side Effects Handling	Critical, as they affect external systems	Important, as they affect local environment/data

Export to Sheets

Chunav karte waqt, aapko hamesha tool ke **khatre (risk)**, **execution speed ki zarurat**, aur **usoolon (constraints)** ko mad-e-nazr rakhna chahiye. Hosted tools zyada versatile hote hain kyunki woh kisi bhi external functionality ko integrate kar sakte hain, lekin unhein `tool_use_behavior` ke control ki zyada zarurat hoti hai. Function tools tez aur asaan hote hain implement karne mein, khaas kar ke internal logic ya light-weight operations ke liye.

13. How do you convert an **agent into a tool** for other agents ?

Aik Agent Ko Doosre Agents Ke Liye Tool Kaise Banate Hain?

Definition ()

Aik **agent ko doosre agents ke liye tool banana** ka matlab hai aik mukammal AI agent (jo khud apne tools, logic, aur state manage karta hai) ko wrapper (ghilaaf) mein band kar ke is tarah pesh karna ke doosre agents usay aik single, reusable function (ya tool) ke tor par istemal kar saken. Is concept ko **Agent as a Tool** ya **Nested Agents** bhi kehte hain, jahan aik bada (higher-level) agent chhote (sub-agent) agents ki salahiyaton ko istemal karta hai.

Explanation ()

`Agent as a Tool` ka **buniyadi maqsad** hai:

"Complex Tasks Ko Chote, Specialized Agents Mein Taqseem Kar Ke Modular aur Scaleable AI Systems Tameer Karna" (Decomposing Complex Tasks into Smaller, Specialized Agents to Build Modular and Scalable AI Systems)

Yeh concept bohot taqatwar hai kyunke yeh hamein AI systems ko organize karne ka aik behtar tareeqa faraham karta hai, bilkul usi tarah jaise programming mein functions ya modules code ko organize karte hain.

Iska matlab hai ke:

1. **Modularity aur Reusability (Modularity aur Dobara Istemal):**
   - **Faida:** Aap har khaas kaam ke liye aik alag agent bana sakte hain (maslan, aik agent sirf database queries ke liye, aik aur email bhejne ke liye, aik aur data analysis ke liye). Phir, jab kisi bade task ke liye in abilities ki zarurat ho, to aap in chote agents ko tools ke tor par istemal kar sakte hain. Is se code modular ho jata hai aur agents ko mukhtalif scenarios mein dobara istemal kiya ja sakta hai.
   - **Misal:** Aik `WeatherAgent` sirf mausam ki maloomat deta hai. Aik `TravelPlannerAgent` mausam, flights, aur hotels ke liye alag agents ko tools ke tor par istemal karta hai.
2. **Specialization (Maharat):**
   - **Faida:** Har sub-agent (jo tool ke tor par kaam kar raha hai) aik khaas domain ya task mein maharat hasil kar sakta hai. Is se uski performance aur reliability behtar hoti hai. Higher-level agent ko un andaruni details ki fikar nahin karni padti.
   - **Misal:** Aik `ResearchAgent` Internet search aur document analysis mein expert ho sakta hai. Jab main agent ko research ki zarurat ho, to woh `ResearchAgent` ko call kar deta hai.
3. **Abstraction (Amaliyat ki Poshidagi):**
   - **Faida:** Main agent ko is baat se matlab nahin hota ke sub-agent andar kya logic istemal karta hai ya kaunse tools chalata hai. Usay sirf sub-agent ki "external interface" (input/output) se matlab hota hai. Is se system ki complexity kam hoti hai.
   - **Misal:** `FinancialAnalystAgent` ko bas `StockPredictorAgent` se stock ki prediction chahiye, usay is baat se matlab nahin ke `StockPredictorAgent` ne kaunse models ya APIs istemal kiye.

4. **Error Isolation (Ghaltiyon ko Alag Karna):**
    - **Faida:** Agar sub-agent mein koi ghalti hoti hai, to woh aksar ussi agent tak mehdood rehti hai. Main agent ko sirf yeh pata chalta hai ke tool (sub-agent) fail ho gaya, jis se debugging aur error handling asaan ho jati hai.
5. **Scalability (Paimanai Salahiyat):**
    - **Faida:** Aap har sub-agent ko alag se deploy aur scale kar sakte hain. Agar `DatabaseAgent` par bojh zyada hai, to aap sirf usay scale kar sakte hain, na ke poore system ko.

Agent ko Tool banane ka Tareeqa:

Buniyadi tareeqa yeh hai ke aap aik function banate hain jo dusre agent ko initialize karta hai, usay input deta hai, uski execution ko handle karta hai, aur phir us agent ka final output wapis karta hai. Yeh function phir standard tool definition (naam, description, parameters) ke sath wrapper (ghilaaf) mein band kar diya jata hai taake isay OpenAI Agents SDK ya kisi aur framework mein aik tool ke tor par recognize kiya ja sake.

Example Code ()

Chaliye aik misal dekhte hain jahan aik `TravelAgent` (main agent) aik `WeatherAgent` (sub-agent) ko tool ke tor par istemal karta hai.

```
import asyncio
from typing import Dict, Any, NamedTuple, Literal, Optional

# --- Sub-Agent's Output Structure ---
class WeatherForecast(NamedTuple):
    location: str
    temperature: float
    condition: str
    is_final: bool # Does this sub-agent have its final result?
    error: Optional[str] = None

# --- 1. Sub-Agent (Jo Tool Banega) ---
class WeatherAgent:
    def __init__(self, agent_name: str = "WeatherChecker"):
        self.agent_name = agent_name
        print(f"[{self.agent_name}]: Initialized.")

    async def get_forecast(self, city: str, days: int = 1) -> WeatherForecast:
        """
        Retrieves the weather forecast for a specified city for the next 'days'.
        This is an async method as it might involve external API calls.
        """
        print(f"[{self.agent_name}]: Getting forecast for {city} for {days} day(s)...")
        await asyncio.sleep(1.5) # Simulate API call latency

        if city.lower() == "karachi":
            return WeatherForecast(city, 32.5, "Sunny", is_final=True)
        elif city.lower() == "london":
            return WeatherForecast(city, 18.0, "Cloudy", is_final=True)
        elif city.lower() == "new york":
            return WeatherForecast(city, 25.0, "Partly Cloudy", is_final=True)
        else:
            return WeatherForecast(city, 0.0, "Unknown", is_final=True, error="City not found or
data unavailable.")

# --- 2. Wrapper Function (Jo Sub-Agent ko Tool Banata Hai) ---
# Yehi woh function hai jo "Agent as a Tool" banata hai.
```

```
# Isay main agent istemal karega.
async def run_weather_agent_tool(city: str, days: int = 1) -> Dict[str, Any]:
    """
    Tool: Get a detailed weather forecast using the specialized WeatherAgent.

    Args:
        city (str): The name of the city to get the weather for.
        days (int): The number of days for the forecast (default is 1).

    Returns:
        Dict[str, Any]: A dictionary containing weather details or an error message.
                        Example: {"location": "Karachi", "temperature": 32.5, "condition": "Sunny"}
    """
    print(f"\n[Tool Wrapper]: Invoking WeatherAgent for {city}...")
    weather_agent_instance = WeatherAgent() # Har tool call ke liye naya instance (ya pool se
lein)

    forecast = await weather_agent_instance.get_forecast(city, days)

    if forecast.is_final:
        if forecast.error:
            print(f"[Tool Wrapper]: WeatherAgent returned an error: {forecast.error}")
            return {"status": "error", "message": forecast.error, "location": forecast.location}
        else:
            print(f"[Tool Wrapper]: WeatherAgent successfully returned forecast for {city}.")
            return {
                "status": "success",
                "location": forecast.location,
                "temperature": forecast.temperature,
                "condition": forecast.condition
            }
    else:
        # If WeatherAgent had intermediate steps, we'd handle them here or require it to be fully
autonomous
        return {"status": "error", "message": "Weather agent did not provide a final output in one
go."}


# --- 3. Main Agent (Jo Sub-Agent ko Tool ke Tor Par Istemal Karega) ---
class TravelAgent:
    def __init__(self, agent_name: str = "TravelPlanner"):
        self.agent_name = agent_name
        # Register the wrapped WeatherAgent as a tool
        # Real SDKs provide a way to register async functions as tools
        self.available_tools = {
            "weather_forecast_tool": run_weather_agent_tool
        }
        print(f"[{self.agent_name}]: Initialized with available tools.")

    async def plan_trip(self, destination: str) -> str:
        """
        Plans a trip to a destination, potentially using other agents as tools.
        """
        print(f"\n[{self.agent_name}]: Planning trip to {destination}...")

        # Scenario: Need weather information, so call the WeatherAgent tool
        if destination.lower() in ["karachi", "london", "new york"]:
            print(f"[{self.agent_name}]: Decided to use 'weather_forecast_tool' for
{destination}.")

            # Call the tool function. This is how the main agent "uses" the sub-agent.
            weather_result = await self.available_tools["weather_forecast_tool"](city=destination)

            if weather_result["status"] == "success":
```

```
                return (f"[{self.agent_name}]: Trip planned for {destination}. "
                        f"Current weather: {weather_result['temperature']}°C and
{weather_result['condition']}.")
            else:
                return (f"[{self.agent_name}]: Could not get weather for {destination}. "
                        f"Error: {weather_result['message']}. Cannot plan trip fully.")
        else:
            return f"[{self.agent_name}]: Planning trip to {destination}. Weather info not
available for this city."


# --- Simulation ---
async def main():
    travel_agent = TravelAgent()

    print("--- Scenario 1: Successful trip planning with WeatherAgent tool ---")
    response1 = await travel_agent.plan_trip("Karachi")
    print(f"\nFinal Response: {response1}")

    print("\n" + "="*70 + "\n")

    print("--- Scenario 2: Trip planning for unknown city, WeatherAgent tool returns error ---")
    response2 = await travel_agent.plan_trip("Sydney")
    print(f"\nFinal Response: {response2}")

# Run the asynchronous main function
if __name__ == "__main__":
    asyncio.run(main())
```

Code Example ki Wazahat ():

1. **`WeatherAgent` (Sub-Agent):**
    - Yeh aik mukammal agent hai jo sirf mausam ki maloomat faraham karta hai. Iska apna `get_forecast` method hai jo internal logic (yahan `asyncio.sleep` se simulate kiya gaya API call) ko handle karta hai.
    - Iski output `WeatherForecast` NamedTuple hai, jis mein `is_final` flag shamil hai.
2. **`run_weather_agent_tool` (Wrapper Function):**
    - **Yahi woh aham hissa hai jo `WeatherAgent` ko doosre agents ke liye tool banata hai.**
    - Yeh aik `async` function hai jo standard tool parameters (yahan `city` aur `days`) leta hai.
    - Is function ke andar:
        - Hum `WeatherAgent` ka aik instance banate hain (`weather_agent_instance = WeatherAgent()`).
        - Hum `weather_agent_instance.get_forecast()` ko `await` karte hain, yaani sub-agent ke task ko chalate hain.
        - Hum sub-agent ke result ko process karte hain aur usay aik standard dictionary format mein wapas karte hain jo tool ke output ke tor par serve ho sake. Is mein `is_final` flag ka check bhi shamil hai (agar sub-agent mein mazeed steps hote to yeh zaroori hota).
    - Is function ka `docstring` (triple quotes) tool ki description aur parameters define karta hai, jo LLM-based agent frameworks mein bohot zaroori hota hai taake LLM ko pata chale ke yeh tool kya karta hai.
3. **`TravelAgent` (Main Agent):**
    - Yeh hamara main agent hai jo trip plan karta hai.
    - `self.available_tools` dictionary mein, `run_weather_agent_tool` ko `weather_forecast_tool` ke naam se register kiya gaya hai.
    - `plan_trip` method ke andar, jab `TravelAgent` ko mausam ki maloomat ki zarurat hoti hai, to woh **`await self.available_tools["weather_forecast_tool"](city=destination)`** call karta hai.
    - **Aham Nuqta:** `TravelAgent` ko is baat se koi matlab nahin hai ke `weather_forecast_tool` asal mein khud aik mukammal agent hai. Usay bas yeh pata hai ke yeh aik function hai jo usay mausam ki maloomat deta hai.

Simulation:

- **Scenario 1:** `TravelAgent` `Karachi` ke liye trip plan karta hai. Usay mausam ki zarurat hoti hai, to woh `weather_forecast_tool` ko call karta hai. `weather_forecast_tool` andar hi andar `WeatherAgent` ko chalata hai, mausam ki maloomat hasil karta hai, aur usay `TravelAgent` ko wapas bhej deta hai.
- **Scenario 2:** `Sydney` ke liye mausam available nahin hai. `WeatherAgent` tool `error` return karta hai, aur `TravelAgent` us error ko handle karta hai.

Is misal se wazeh hota hai ke `run_weather_agent_tool` function kaise `WeatherAgent` ki complex functionality ko aik simple, re-usable tool mein convert karta hai jise koi bhi doosra agent asani se istemal kar sakta hai.

Summary ()

Aik agent ko doosre agents ke liye tool banane ka maqsad **complex tasks ko chote, specialized agents mein taqseem kar ke modular aur scaleable AI systems tameer karna** hai.

Is tareeqe mein:

1. **Chota Agent (Sub-Agent):** Aap aik specialized agent banate hain jo aik khaas kaam mein maharat rakhta hai (maslan, `WeatherAgent`).
2. **Wrapper Function (Tool):** Aap aik **wrapper function** (ya method) banate hain. Yahi woh function hai jo doosre agent ke liye "tool" ka kaam karega.
3. **Execution Logic:** Is wrapper function ke andar, aap sub-agent ka instance banate hain, usay input dete hain, uski execution ko handle karte hain (aksar `await` ke zariye agar sub-agent async ho), aur phir sub-agent ka final output le kar usay aik standard tool result format mein wapas karte hain.
4. **Main Agent:** Ab main agent (maslan, `TravelAgent`) is wrapper function ko aik standard tool ki tarah call kar sakta hai.

Buniyadi faida yeh hai ke is se system modular ho jata hai, har agent apni maharat ke domain tak mehdood rehta hai, aur higher-level agents ko andaruni pechidegiyon (internal complexities) se azadi milti hai, jis se overall system zyada saaf, asaan-fahum, aur scalable banta hai.

14. What method returns all tools available to an agent ?

Kaunsa Method Agent Ko Dastiyab Tamam Tools Wapas Karta Hai?

Definition ()

Aik **tool** woh functionality hoti hai jo aik AI agent istemal kar sakta hai apne tasks ko mukammal karne ke liye (maslan, database query karna, email bhejna, ya web search karna). Agent ko yeh maloomat hona zaroori hai ke uske paas kaun kaunse tools hain aur woh unhein kaise istemal kar sakta hai. Jis method ke zariye agent ko yeh maloomat milti hain, woh **dastiyab tamam tools ko wapas karta hai**.

Explanation ()

Jab aik AI agent koi kaam karta hai, to usay aksar external systems ya specific functionalities tak rasai ki zarurat hoti hai. Yeh functionalities **tools** ke roop mein agent ko faraham ki jati hain. Agent ka "brain" (aam taur par aik Large Language Model - LLM) in tools ki description ko parhta hai aur faisla karta hai ke kis sawal ya task ke liye kaunsa tool sabse behtar rahega.

Jis method ke zariye agent ko uske saare tools ki fehrist milti hai, us ka **buniyadi maqsad** hai:

"Agent Ko Apne Ikhtiyarat Ke Bare Mein Mukammal Aur Mauzoo Maloomat Faraham Karna Taake Woh Munasib Faisle Le Sake." (To Provide the Agent with Complete and Relevant Information About Its Capabilities So It Can Make Appropriate Decisions.)

Iska matlab hai ke yeh method agent ko aik "toolbelt" (auzaar patti) ya "skill set" (hunar ka set) dikhata hai.

- **Agent ka Faisla Sazi (Agent's Decision Making):** Yeh method LLM-based agent ko yeh samajhne mein madad karta hai ke woh kya kya kar sakta hai. LLM user ki query ko dekhta hai, phir available tools (jo is method se milte hain) ki descriptions ko dekhta hai, aur phir faisla karta hai ke kis tool ko call karna hai aur kis parameters ke sath.
- **Dynamic Tool Discovery (Dynamic Tools Ki Daryaft):** Kuch systems mein, tools dynamically load ya unload ho sakte hain. Yeh method hamesha agent ko current aur up-to-date tools ki fehrist faraham karega.
- **Framework Integration (Framework Ke Sath Integration):** Agent frameworks (jaise LangChain, AutoGen, ya OpenAI Assistants API) mein, yeh method framework ke liye aik standard interface faraham karta hai taake woh agent ke liye tools ki list hasil kar sake aur unhein LLM ke context mein inject kar sake.

Kaunsa Method Istemal Hota Hai?

OpenAI Agents SDK (aur is se milte julte frameworks) mein, aam taur par yeh method ya property hoti hai:

`agent.available_tools` (ya `agent.tools`)

Ya koi aisa method jo tools ki list (ya dictionary) wapas karta hai, maslan:

`agent.get_tools()`

Yeh method (ya property) mukhtalif tareeqon se tools ko wapas kar sakta hai:

- **List of Tool Objects:** Har object mein tool ka naam, description, aur parameters shamil hote hain.
- **Dictionary of Tools:** Jahan keys tools ke naam hon aur values tool objects hon.
- **JSON Schema:** Aam taur par LLMs ko tools ki jo descriptions di jati hain woh JSON schema format mein hoti hain taake LLM unhein asani se samajh sake.

Example Code ()

Chaliye aik simple `MyAgent` class dekhte hain jo `available_tools` property ka istemal karta hai.

```
from typing import List, Dict, Any, Callable, NamedTuple

# --- Tool Definition (A hypothetical structure for a tool) ---
class Tool(NamedTuple):
    name: str
    description: str
    func: Callable # The actual Python function to execute
    parameters_schema: Dict[str, Any] # Schema for LLM to understand parameters

# --- Example Tools ---
def get_current_weather(location: str) -> Dict[str, str]:
    """Retrieves the current weather for a specific location."""
    print(f"\n[Executing Tool: get_current_weather] for {location}")
    if location.lower() == "karachi":
        return {"location": "Karachi", "temperature": "30°C", "condition": "Sunny"}
    return {"location": location, "temperature": "N/A", "condition": "Unknown"}

def send_email(recipient: str, subject: str, body: str) -> str:
    """Sends an email to a specified recipient with a subject and body."""
    print(f"\n[Executing Tool: send_email] to {recipient} with subject '{subject}'")
    return f"Email sent successfully to {recipient}."

# --- Agent Class ---
class MyAgent:
    def __init__(self, name: str):
        self.name = name
        self._tools: List[Tool] = [] # Internal list to store tools

    def add_tool(self, tool: Tool):
        """Adds a tool to the agent's available tools."""
        self._tools.append(tool)
        print(f"Agent '{self.name}': Tool '{tool.name}' added.")

    @property
    def available_tools(self) -> List[Tool]:
        """
        This method/property returns all tools currently available to the agent.
        In a real SDK, this might also return tools in a format digestible by an LLM (e.g., JSON
schema).
        """
        print(f"Agent '{self.name}': Providing list of available tools.")
        return self._tools
```

```
    def run_tool(self, tool_name: str, **kwargs: Any) -> Any:
        """Simulates the agent executing a chosen tool."""
        for tool in self._tools:
            if tool.name == tool_name:
                print(f"Agent '{self.name}': Executing tool '{tool_name}'...")
                return tool.func(**kwargs)
        raise ValueError(f"Tool '{tool_name}' not found.")

    def process_request(self, user_query: str) -> str:
        """
        Simulates agent processing a request, which involves looking at available tools.
        In a real scenario, an LLM would make the decision.
        """
        print(f"\nAgent '{self.name}': Processing request: '{user_query}'")

        # Agent first looks at its available tools
        tools_for_llm = [
            {"name": t.name, "description": t.description, "parameters": t.parameters_schema}
            for t in self.available_tools # <--- Yahan 'available_tools' ko access kiya ja raha
hai
        ]

        print(f"Agent '{self.name}': Considering these tools: {tools_for_llm}")

        # --- Hypothetical LLM Decision Logic ---
        # A real LLM would decide based on query and tools_for_llm
        if "weather" in user_query.lower():
            if any(tool['name'] == 'get_current_weather' for tool in tools_for_llm):
                print(f"Agent '{self.name}': Decided to use 'get_current_weather'.")
                location = user_query.lower().replace("what's the weather in",
"").strip().replace("?", "")
                if not location: location = "Karachi" # Default if not specified
                result = self.run_tool('get_current_weather', location=location.title())
                return f"The weather in {result['location']} is {result['temperature']} and
{result['condition']}."
            else:
                return "I don't have a tool to get weather information."
        elif "send email to" in user_query.lower():
            if any(tool['name'] == 'send_email' for tool in tools_for_llm):
                print(f"Agent '{self.name}': Decided to use 'send_email'.")
                # Very basic parsing for demo
                parts = user_query.split("send email to")
                recipient_part = parts[1].split("subject")[0].strip()
                subject_part = user_query.split("subject")[1].split("body")[0].strip()
                body_part = user_query.split("body")[1].strip()

                result = self.run_tool('send_email', recipient=recipient_part,
subject=subject_part, body=body_part)
                return result
            else:
                return "I don't have a tool to send emails."
        else:
            return "I can help with weather or sending emails."


# --- Simulation ---
if __name__ == "__main__":
    my_agent = MyAgent(name="AssistantBot")

    # Add tools to the agent
    my_agent.add_tool(Tool(
        name="get_current_weather",
        description="Retrieves the current weather for a specific location.",
        func=get_current_weather,
```

```
        parameters_schema={"location": {"type": "string", "description": "The city name"}}
    ))
    my_agent.add_tool(Tool(
        name="send_email",
        description="Sends an email to a specified recipient.",
        func=send_email,
        parameters_schema={
            "recipient": {"type": "string", "description": "Email address of the recipient"},
            "subject": {"type": "string", "description": "Subject of the email"},
            "body": {"type": "string", "description": "Body content of the email"}
        }
    ))

    print("\n" + "="*50 + "\n")

    # Agent processes requests using its available tools
    response1 = my_agent.process_request("What's the weather in Lahore?")
    print(f"\nFinal Agent Response: {response1}")

    print("\n" + "="*50 + "\n")

    response2 = my_agent.process_request("send email to test@example.com subject Hello Agent body
This is a test email.")
    print(f"\nFinal Agent Response: {response2}")

    print("\n" + "="*50 + "\n")

    response3 = my_agent.process_request("Tell me a joke.")
    print(f"\nFinal Agent Response: {response3}")

    print("\n" + "="*50 + "\n")

    # Create another agent without the email tool
    another_agent = MyAgent(name="WeatherBot")
    another_agent.add_tool(Tool(
        name="get_current_weather",
        description="Retrieves the current weather for a specific location.",
        func=get_current_weather,
        parameters_schema={"location": {"type": "string", "description": "The city name"}}
    ))

    print("\n--- Agent 'WeatherBot' (without email tool) ---")
    response4 = another_agent.process_request("send email to user@example.com subject Test body
Test.")
    print(f"\nFinal Agent Response: {response4}")
```

Code Example ki Wazahat ():

1. **`Tool` (NamedTuple):** Yeh aik simple structure hai jo tool ki basic maloomat ko define karta hai: `name`, `description`, asal `func` (jo tool ka kaam karega), aur `parameters_schema` (jo LLM ko batata hai ke tool ko kaunse arguments chahiye).
2. **`MyAgent` Class:**
    - `_tools`: Yeh aik private list hai jahan agent ke tools store kiye jate hain.
    - `add_tool(self, tool: Tool)`: Yeh method agent mein naye tools shamil karta hai.
    - **`@property available_tools(self) -> List[Tool]:`: Yahi woh method/property hai jis ke bare mein hum baat kar rahe hain.** Jab bhi `my_agent.available_tools` ko call kiya jata hai, to yeh agent ko dastiyab tamam tools ki fehrist wapas karta hai. Real SDKs mein, yeh tools ko LLM-friendly format (jaise JSON schema) mein bhi convert kar sakta hai.
    - `run_tool`: Yeh simulate karta hai ke agent actually kis tarah aik tool ko call karta hai uske naam ki bunyad par.
    - `process_request`: Yeh method simulate karta hai ke agent kis tarah user ki query ko process karta hai. Is ke andar, agent sabse pehle **`self.available_tools`** ko access karta hai taake usay pata chale ke uske paas kaunse tools hain jinhein woh istemal kar sakta hai.

Simulation:

- `MyAgent` ko `get_current_weather` aur `send_email` tools diye jate hain. Jab `process_request` call hota hai, to agent `available_tools` ke zariye apni tools ki list dekhta hai aur uske mutabiq faisla karta hai.
- `WeatherBot` ko sirf `get_current_weather` tool diya jata hai. Jab us se email bhejne ko kaha jata hai, to woh `available_tools` mein `send_email` na paane ki wajah se inkar kar deta hai.

Summary ()

Agent ko dastiyab tamam tools wapas karne wala method (ya property) aam taur par **`agent.available_tools`** (ya **`agent.get_tools()`**) hota hai.

Is method ka **buniyadi maqsad** yeh hai ke **agent ko apne ikhtiyarat (capabilities) ke bare mein mukammal aur mauzoo maloomat faraham karna taake woh user ki queries ko behtar tareeqe se samajh sake aur munasib tool ka intekhab kar sake**.

Yeh method agent ke "brain" (LLM) ko un functions ki descriptions aur parameters ki list deta hai jin tak agent ki rasai hai. Is maloomat ki bunyad par hi agent faisla karta hai ke kab, kaunsa, aur kis parameters ke sath tool ko istemal karna hai.

15. What is the first parameter of every function tool ?

Har Function Tool Frist Parameter Kya Hota Hai?

Definition ()

Aik **function tool** aik aam Python function hota hai (ya kisi aur programming language ka function) jise aik AI agent (khaas kar ke Large Language Models - LLMs) istemal karne ke liye design kiya jata hai. Yeh function koi khaas operation perform karta hai, jaise database se data lana, calculations karna, ya kisi API ko call karna.

Explanation ()

Jab aik LLM-based agent kisi function tool ko call karta hai, to woh asal mein us function ko uske zaroori arguments ke sath invoke karta hai. Is silsile mein, har function tool ka pehla parameter aik khaas maqsad ke liye hota hai:

"Har Function Tool Ka Pehla Parameter Us Ki Input Values Ko Receive Karta Hai Jo LLM Ki Taraf Se Faraham Ki Jati Hain." (The First Parameter of Every Function Tool Receives the Input Values Provided by the LLM.)

Iska matlab hai ke yeh pehla parameter woh jagah hai jahan LLM us tool ko chalane ke liye zaroori maloomat "dalta" hai. Aam taur par, yeh pehla parameter `ToolSpec` ya is se milte-julte naam se hota hai, ya phir direct woh arguments hote hain jo LLM ne tool ko call karte waqt specify kiye hotay hain.

Aaiye is concept ko mazeed tafseel se samjhte hain:

1. **LLM Aur Tool Ke Darmiyan Communication (LLM and Tool Communication):**
    - LLM ko user ki query milti hai aur woh faisla karta hai ke kaunsa tool istemal karna hai.
    - Jab woh tool ka intekhab kar leta hai, to LLM us tool ki `description` aur `parameters_schema` ko dekhta hai. Is schema ki bunyad par, LLM woh values generate karta hai jo us tool ke parameters ke liye zaroori hain.
    - Yeh generated values phir tool ke pehle parameter ke zariye tool function tak pohanchayi jati hain.
2. **Parameter Structure (Parameter Ki Banawat):**
    - Aam taur par, OpenAI Agents SDK aur is jaisi doosri frameworks mein, yeh pehla parameter aik **dictionary** (`Dict`) ya aik **structured object** (jaise Pydantic model) hota hai.
    - Is dictionary ya object ke andar woh tamam arguments hote hain jo LLM ne tool call ke liye mukarrar kiye hain. Har argument ka naam (key) aur uski value is dictionary/object mein mojood hoti hai.
    - **Misal:** Agar tool ko `city` aur `unit` (Celsius/Fahrenheit) chahiye, to LLM aik dictionary generate karega `{"city": "Karachi", "unit": "Celsius"}` aur yeh dictionary tool ke pehle parameter mein aa jayegi.
3. **Flexibility (Lachak):**
    - Yeh tareeqa kar tools ko bohot flexible banata hai. LLM ko sirf tool ke schema ko follow karna hota hai, aur woh mukhtalif combinations mein arguments de sakta hai.
    - Tool function phir is dictionary/object se zaroori arguments ko extract kar ke apna kaam karta hai.
4. **Decoupling LLM from Tool Implementation (LLM ko Tool Implementation se Alag Karna):**

- Yeh structure LLM ko tool ki andaruni implementation details se azad rakhta hai. LLM ko sirf tool ke `name`, `description`, aur `parameters_schema` se matlab hota hai. Usay is baat ki fikar nahin hoti ke tool function ke andar actual logic kaise kaam karta hai.
- Tool function ko bhi is baat ki fikar nahin hoti ke arguments kis tarah generate hue hain, usay bas dictionary mein apne arguments mil jate hain.

5. **Error Handling (Ghalti Ki Shinakht):**
   - Agar LLM ghalat arguments generate karta hai, ya zaroori arguments miss kar deta hai, to tool function is pehle parameter (dictionary/object) mein unko check kar sakta hai aur munasib error throw kar sakta hai.

Example Code ()

Chaliye aik misal dekhte hain jahan aik function tool ka pehla parameter LLM se aane wali input values ko kaise receive karta hai. Hum yahan Python aur Pydantic ka istemal karenge, jo LLM tool definition ke liye aam taur par use hota hai.

```
from pydantic import BaseModel, Field
from typing import Dict, Any, Literal

# --- 1. Tool Input Schema (LLM ko yeh schema diya jata hai) ---
# Yeh define karta hai ke LLM ko 'get_weather' tool chalane ke liye kya arguments chahiye.
class GetWeatherParams(BaseModel):
    """Parameters for getting weather information."""
    location: str = Field(..., description="The city and state, e.g., 'San Francisco, CA'")
    unit: Literal["celsius", "fahrenheit"] = Field("celsius", description="The unit of temperature
to use.")

# --- 2. Function Tool Definition ---
# Yahan `params: GetWeatherParams` hamara pehla parameter hai.
# Yeh woh parameter hai jo LLM se aane wali input values ko receive karega.
def get_weather(params: GetWeatherParams) -> Dict[str, Any]:
    """
    Retrieves the current weather for a specified location.
    """
    print(f"\n[Tool Execution]: 'get_weather' called with params: {params.model_dump()}")

    # Ab hum params object se values ko access kar sakte hain
    location = params.location
    unit = params.unit

    if location.lower() == "karachi":
        if unit == "celsius":
            temp = "30°C"
        else:
            temp = "86°F"
        return {"location": location, "temperature": temp, "condition": "Sunny"}
    elif location.lower() == "london":
        if unit == "celsius":
            temp = "15°C"
        else:
            temp = "59°F"
        return {"location": location, "temperature": temp, "condition": "Cloudy"}
    else:
        return {"location": location, "temperature": "N/A", "condition": "Unknown"}

# --- 3. Hypothetical LLM Call Simulation ---
# Yeh simulate karta hai ke LLM tool ko kaise call karta hai.
def simulate_llm_tool_call(tool_function: Callable, llm_generated_args: Dict[str, Any]) -> Any:
    """
    Simulates an LLM calling a function tool with generated arguments.
```

```
    """
    print(f"\n[LLM Simulation]: LLM generated args for tool: {llm_generated_args}")

    # LLM ke generated arguments ko tool ke pehle parameter mein fit karna.
    # Pydantic model automatically dict ko validate aur parse kar lega.
    tool_input_object = tool_function.__annotations__['params'](**llm_generated_args)

    # Ab tool function ko call karein us input object ke sath
    result = tool_function(tool_input_object)
    print(f"[LLM Simulation]: Tool returned: {result}")
    return result

# --- Main Execution ---
if __name__ == "__main__":
    print("--- Scenario 1: LLM calls 'get_weather' for Karachi in Celsius ---")
    llm_args_1 = {"location": "Karachi", "unit": "celsius"}
    weather_result_1 = simulate_llm_tool_call(get_weather, llm_args_1)
    print(f"Final Weather Result: {weather_result_1}")

    print("\n--- Scenario 2: LLM calls 'get_weather' for London in Fahrenheit ---")
    llm_args_2 = {"location": "London", "unit": "fahrenheit"}
    weather_result_2 = simulate_llm_tool_call(get_weather, llm_args_2)
    print(f"Final Weather Result: {weather_result_2}")

    print("\n--- Scenario 3: LLM calls 'get_weather' for an unknown city ---")
    llm_args_3 = {"location": "Sydney", "unit": "celsius"}
    weather_result_3 = simulate_llm_tool_call(get_weather, llm_args_3)
    print(f"Final Weather Result: {weather_result_3}")

    print("\n--- Scenario 4: LLM calls 'get_weather' with missing 'location' (Pydantic will raise
error) ---")
    try:
        llm_args_4 = {"unit": "celsius"} # Missing required 'location'
        simulate_llm_tool_call(get_weather, llm_args_4)
    except Exception as e:
        print(f"[ERROR]: Simulation failed due to missing parameter: {e}")
```

Code Example ki Wazahat ():

1. **`GetWeatherParams(BaseModel):`** Yeh aik Pydantic model hai jo `get_weather` function tool ke liye **input schema** define karta hai. LLMs ko aksar aise models ya JSON schema diye jate hain taake woh samajh saken ke tool ko kya arguments chahiye.
2. **`def get_weather(params: GetWeatherParams) -> Dict[str, Any]:`**:
    - Yahan, **`params: GetWeatherParams`** hamara function tool ka **pehla parameter** hai.
    - Jab LLM is tool ko call karne ka faisla karta hai, to woh apne understanding ke mutabiq `location` aur `unit` ki values generate karta hai (maslan, `{"location": "Karachi", "unit": "celsius"}`).
    - Yeh dictionary phir automatically `GetWeatherParams` object mein convert ho kar `params` variable mein aa jati hai.
    - Function ke andar, hum `params.location` aur `params.unit` ke zariye un values ko asani se access kar sakte hain.
3. **`simulate_llm_tool_call` Function:**
    - Yeh function is baat ko simulate karta hai ke aik agent ya framework kis tarah LLM ke generated arguments ko pakadta hai aur unhein tool function tak pohanchata hai.
    - `tool_function.__annotations__['params'](**llm_generated_args)` line LLM ke generated dictionary ko `GetWeatherParams` Pydantic model mein convert karti hai. Agar arguments theek na hon (maslan, `location` missing ho), to Pydantic automatically validation error dega.

Khulasa: Is misal se wazeh hota hai ke `params` (ya koi bhi naam jo aap dein) **function tool ka pehla parameter** hota hai, aur yahi woh parameter hai jo LLM se aane wali input values ko aik structured tareeqe se receive karta hai.

Summary ()

Har function tool ka **frist parameter** woh hota hai jo LLM (Large Language Model) ki taraf se **tool ko chalane ke liye faraham ki gayi input values ko receive karta hai**.

Aam taur par, yeh pehla parameter aik **structured object** (jaise Python mein Pydantic model ka instance) ya aik **dictionary** (`Dict`) hota hai. Is object/dictionary ke andar woh tamam arguments (keys aur values ke roop mein) mojood hoti hain jo LLM ne tool ko call karte waqt specify ki thin.

Iska buniyadi maqsad LLM aur tool ki andaruni functionality ke darmiyan aik saaf interface faraham karna hai, jis se:

- LLM ko sirf tool ke schema ka pata ho, implementation details ka nahin.
- Tool function ko asani se zaroori arguments mil jaayen.
- Validation aur error handling asaan ho.

16. What is the purpose of the get_system_prompt() method ?

`get_system_prompt()` Method Ka Maqsad Kya Hai?

Definition ()

`get_system_prompt()` aik method hai jo OpenAI Agents SDK ya Large Language Model (LLM) based agent frameworks mein istemal hota hai. Iska buniyadi maqsad **LLM ko uske role, maqsad, qawaid (rules), aur tareeqa-e-kaar ke bare mein ibtidayi hidayat (initial instructions) faraham karna** hai. Yeh woh "context" ya "guidance" hai jo LLM ko di jati hai taake woh apni zimmedariyon ko sahi tareeqay se samajh aur poora kar sake.

Explanation ()

Jab aap aik Large Language Model (LLM) ko aik agent ke tor par istemal karte hain, to woh aik "clean slate" (khaali zehen) ki tarah hota hai. Usay batana padta hai ke usay kya karna hai, kis hadd tak karna hai, aur kaunse qawaid par amal karna hai. Yahan par `get_system_prompt()` method ka kaam aata hai.

`get_system_prompt()` ka **buniyadi maqsad** hai:

"LLM (Agent) Ko Us Ke Role, Maqsad, Qawaid, Aur Andaruni Tools Ke Istemal Ke Bare Mein Comprehensive Ibtidayi Hidayat Faraham Karna Taake Woh Task Ko Behtar Tareeqay Se Anjaam De Sake." (To Provide the LLM (Agent) with Comprehensive Initial Instructions Regarding Its Role, Purpose, Rules, and the Use of Internal Tools, Enabling It to Perform the Task More Effectively.)

Iska matlab hai ke yeh method LLM ko uske "mission briefing" (muhim ki tafseel) deta hai.

1. **Role Definition (Kirdar ki Tareef):**
   - **Faida:** Yeh LLM ko batata hai ke woh kaun hai (maslan, "You are a helpful customer support agent," ya "You are a financial analyst"). Yeh LLM ke jawab dene ke andaaz (tone), uske focus, aur uski salahiyaton ko influence karta hai.
   - **Misal:** Agar prompt mein hai "You are a polite and empathetic assistant," to LLM us tone ko apnayega.
2. **Task Objective (Task Ka Maqsad):**
   - **Faida:** Yeh LLM ko batata hai ke usay kya hasil karna hai. Yeh specific goals set karta hai jinhein LLM ko poora karna hai.
   - **Misal:** "Your goal is to answer user questions about current weather conditions."
3. **Constraints and Rules (Pabandiyan aur Qawaid):**
   - **Faida:** Yeh LLM ko batata hai ke usay kya nahin karna chahiye, ya usay kin hudood mein rehna chahiye. Is mein safety instructions, privacy guidelines, ya specific output format ki pabandiyan shamil ho sakti hain.
   - **Misal:** "Do not provide personal financial advice." ya "Keep your answers concise and to the point."
4. **Tool Usage Guidelines (Tool Istemal Karne Ki Hidayat):**
   - **Faida:** Agar agent ke paas tools hain, to system prompt LLM ko yeh hidayat de sakta hai ke un tools ko kab aur kaise istemal karna hai. Is mein tool ka naam, uski description, aur uske parameters shamil hote hain, aam taur par JSON schema format mein.
   - **Misal:** "You have access to a `get_weather` tool. Use it whenever the user asks about weather. Its schema is: `<JSON_SCHEMA_FOR_WEATHER_TOOL>`."
   -

5. **Output Format (Output Ka Format):**
    - **Faida:** Prompt LLM ko yeh bata sakta hai ke usay kis format mein jawab dena hai (maslan, "Always respond in JSON," ya "Summarize in bullet points").
    - **Misal:** "Respond in the following JSON format: `{'answer': '...', 'tool_used': '...'}`."
6. **Contextual Information (Contextual Maloomat):**
    - **Faida:** `get_system_prompt()` kuch initial context bhi de sakta hai jo user ki query se mutasir nahin hota lekin agent ke liye buniyadi maloomat faraham karta hai (maslan, current date, system version).

Yeh Prompt LLM Tak Kaise Pohanchta Hai?

`get_system_prompt()` method jo string return karta hai, woh LLM API call ke `system` role message ke taur par bheja jata hai. LLM is `system` message ko user ke input (jo `user` role message hota hai) se bhi zyada tarjeeh deta hai, kyunke yeh uski buniyadi hidayat hoti hai.

Example Code ()

Chaliye aik `WeatherAgent` ki misal dekhte hain jo `get_system_prompt()` method istemal karta hai.

```
from typing import Dict, Any, List

# --- Hypothetical Tool Schema ---
# Aam taur par tool schemas aise define kiye jaate hain LLM ko batane ke liye
WEATHER_TOOL_SCHEMA = {
    "name": "get_current_weather",
    "description": "Get the current weather conditions for a specific location.",
    "parameters": {
        "type": "object",
        "properties": {
            "location": {
                "type": "string",
                "description": "The city name, e.g., Karachi or London."
            },
            "unit": {
                "type": "string",
                "enum": ["celsius", "fahrenheit"],
                "description": "The unit of temperature to use."
            }
        },
        "required": ["location"]
    }
}

# --- Agent Class ---
class WeatherAgent:
    def __init__(self, location_bias: str = "Karachi"):
        self.location_bias = location_bias
        self.tools_available = [WEATHER_TOOL_SCHEMA] # Simplified tool list

    def get_system_prompt(self) -> str:
        """
        Returns the system prompt that defines the agent's role, rules,
        and available tools for the LLM.
        """
        system_instructions = (
            f"You are a helpful and accurate weather assistant. "
            f"Your main goal is to provide current weather information to the user.\n"
```

```
            f"Always prioritize providing accurate data. If you cannot find information for a
requested location, "
            f"politely state that you don't have the data.\n"
            f"The current date is June 20, 2025. Be mindful of this when answering date-related
queries.\n\n"
            f"You have access to the following tools:\n"
            f"{self._format_tools_for_prompt(self.tools_available)}\n\n"
            f"When using a tool, respond with the exact JSON tool call. "
            f"If the user's query is completely handled by a tool, provide a concise summary. "
            f"If the user asks for a location not supported by the tool, respond gracefully."
            # Adding a "bias" or initial context based on agent's configuration
            f"\n\nBy default, consider questions about local weather to be for
{self.location_bias}."
        )
        return system_instructions

    def _format_tools_for_prompt(self, tools: List[Dict[str, Any]]) -> str:
        """Helper to format tool schemas nicely for the prompt."""
        import json
        formatted_tools = []
        for tool in tools:
            # For a real LLM, you'd usually pass this as part of the API call's 'tools' parameter,
            # but for a text-based prompt, this format helps.
            formatted_tools.append(f"Tool Name: {tool['name']}\nDescription:
{tool['description']}\nParameters: {json.dumps(tool['parameters'], indent=2)}")
        return "\n\n".join(formatted_tools)

    # --- Simulate LLM interaction ---
    # In a real scenario, an LLM would interpret the system prompt and user query.
    def simulate_llm_response(self, user_query: str) -> str:
        system_prompt = self.get_system_prompt()
        print(f"\n--- System Prompt (sent to LLM) ---")
        print(system_prompt)
        print(f"-----------------------------------")
        print(f"User Query: '{user_query}'")

        # This is a very simplified simulation of LLM's understanding and action.
        # A real LLM would use the system prompt to guide its thinking process.
        if "weather in" in user_query.lower():
            if "lahore" in user_query.lower():
                # LLM decided to use the tool based on system prompt and tool schema
                return ('{"tool_call": {"name": "get_current_weather", "parameters": {"location":
"Lahore", "unit": "celsius"}}}')
            elif "default" in user_query.lower() or "here" in user_query.lower():
                # LLM uses the default location bias from the system prompt
                return ('{"tool_call": {"name": "get_current_weather", "parameters": {"location":
"' + self.location_bias + '", "unit": "celsius"}}}')
            else:
                return "I need a specific city name to check the weather."
        elif "hello" in user_query.lower():
            return "Hello! How can I assist you with weather today?"
        else:
            return "I am a weather assistant. Please ask me about the weather."

# --- Main Execution ---
if __name__ == "__main__":
    weather_bot = WeatherAgent(location_bias="Islamabad")

    print("\n--- Scenario 1: User asks for specific weather ---")
    llm_output1 = weather_bot.simulate_llm_response("What's the weather in Lahore?")
    print(f"LLM's Simulated Output: {llm_output1}")

    print("\n--- Scenario 2: User asks for default location weather ---")
    llm_output2 = weather_bot.simulate_llm_response("What's the weather here?")
```

```
    print(f"LLM's Simulated Output: {llm_output2}")

    print("\n--- Scenario 3: User asks out of scope question ---")
    llm_output3 = weather_bot.simulate_llm_response("Tell me a joke.")
    print(f"LLM's Simulated Output: {llm_output3}")
```

Code Example ki Wazahat ():

1. **`WEATHER_TOOL_SCHEMA`:** Yeh dictionary LLM ko `get_current_weather` tool ke bare mein maloomat deti hai, jismein uska naam, description, aur parameters shamil hain. LLM is schema ko parh kar tool ko sahi tareeqay se istemal karna seekhta hai.
2. **`WeatherAgent` Class:**
    - `self.location_bias`: Yeh aik configuration hai jo agent ko initial context deti hai.
    - **`get_system_prompt()` Method:**
        - Yahi woh aham method hai jo LLM ke liye system prompt banata hai.
        - Ismein agent ka `role` ("helpful and accurate weather assistant"), `goal` ("provide current weather information"), `constraints` ("If you cannot find information... politely state"), aur `tool usage guidelines` ("You have access to the following tools...") shamil hain.
        - `_format_tools_for_prompt` helper function tool schemas ko LLM ke liye readable format mein layega (asal mein LLM APIs mein `tools` parameter alag se bheja jata hai, lekin concept yahi hai ke system prompt mein tools ki maloomat bhi shamil hoti hain).
        - `location_bias` ko bhi prompt mein dala gaya hai taake LLM default locations ko samajh sake.
3. **`simulate_llm_response()`:** Yeh function sirf LLM ke behaviour ko simulate karta hai. Yeh `get_system_prompt()` se hasil shuda `system_prompt` ko print karta hai (jo LLM ko bheja jata hai) aur phir user query ke bunyad par LLM ke potential jawab ko represent karta hai.

Simulation:

- Jab `WeatherAgent` ko initialize kiya jata hai (`location_bias="Islamabad"` ke sath), to `get_system_prompt()` method ke zariye LLM ko bataya jata hai ke uski default location Islamabad hai.
- **Scenario 1:** User Lahore ke mausam ke bare mein poochhta hai. LLM `get_system_prompt()` se tools ki maloomat le kar `get_current_weather` tool ko Lahore ke liye call karne ka faisla karta hai.
- **Scenario 2:** User sirf "What's the weather here?" poochhta hai. `get_system_prompt()` mein di gayi hidayat ("By default, consider questions about local weather to be for Islamabad") ki wajah se LLM Islamabad ke liye tool call karta hai.
- **Scenario 3:** User out of scope question poochhta hai. LLM system prompt mein di gayi role definition ("I am a weather assistant...") ki wajah se jawab deta hai ke woh sirf mausam ke bare mein madad kar sakta hai.

Is misal se wazeh hota hai ke `get_system_prompt()` method kis tarah LLM ke behaviour ko buniyadi taur par define aur guide karta hai.

Summary ()

`get_system_prompt()` method ka buniyadi maqsad **LLM (Large Language Model) ko uske role, maqsad, qawaid (rules), aur dastiyab tools ke istemal ke bare mein comprehensive ibtidayi hidayat (initial instructions) faraham karna** hai.

Yeh method LLM ko aik "mission briefing" deta hai, jismein shamil hota hai:

- **Kirdar ki Tareef (Role Definition):** LLM kaun hai aur usay kis tone mein baat karni hai.
- **Task Ka Maqsad (Task Objective):** LLM ko kya hasil karna hai.
- **Pabandiyan aur Qawaid (Constraints and Rules):** Kin hudood mein rehna hai ya kya nahin karna.
- **Tool Istemal Karne Ki Hidayat (Tool Usage Guidelines):** Available tools aur unhein kaise istemal karna hai.
- **Output Ka Format (Output Format):** Jawab kis format mein hona chahiye.

Yeh system prompt LLM API call ke `system` role message ke taur par bheja jata hai, aur LLM isay apni buniyadi hidayat ke taur par istemal karta hai taake woh user ki queries ko behtar tareeqay se process kar sake aur munasib jawab de sake.

17. What's the difference between InputGuardrail and OutputGuardrail ?

Input Guardrail aur Output Guardrail Mein Kya Farq Hai?

Definition ()

Guardrails aise mechanisms hote hain jo AI systems (khaas kar ke LLMs aur agents) ke behaviour ko control aur regulate karte hain. Inka maqsad yeh yaqeeni banana hai ke AI system **safe, responsible, aur intended tareeqay** se kaam kare. `InputGuardrail` aur `OutputGuardrail` do buniyadi qism ke guardrails hain jo system ke **data flow ke do mukhtalif marhalon par** kaam karte hain.

Explanation ()

`InputGuardrail` aur `OutputGuardrail` ke darmiyan buniyadi farq unke **Amal Ke Maqam (Point of Operation)** aur **Control Ke Maqsad (Purpose of Control)** mein hai.

"Input Guardrails aane wali maloomat (user input) ko control karte hain taake LLM tak ghalat ya na-munasib content na pahunche, jabke Output Guardrails LLM ke tayar jawab ko control karte hain taake woh safe, relevant, aur policy ke mutabiq ho."

(Input Guardrails control incoming information (user input) to prevent inappropriate content from reaching the LLM, while Output Guardrails control the LLM's generated response to ensure it's safe, relevant, and policy-compliant.)

Aaiye in dono mein farq ko tafseel se dekhte hain:

Input Guardrail ()

- **Amal Ka Maqam (Point of Operation):** `InputGuardrail` **LLM tak pahunchne se pehle** user ke input (ya kisi bhi source se aane wali maloomat) par lagoo hota hai. Yeh LLM ke "inbox" ka darwaza hai.
- **Maqsad-e-Control (Purpose of Control):**
    1. **Safety and Harm Prevention (Safety aur Nuqsan Se Bachao):** Sabse aham maqsad yeh yaqeeni banana hai ke LLM tak koi bhi **nuksan-deh (harmful), illegal, ya policy ki khilaf warzi karne wali maloomat** na pahunche. Is mein hate speech, harassment, violence, ya sensitive personal information shamil ho sakti hai.
    2. **Relevance and Scope (Relevance aur Scope):** Input ko filter karna taake LLM ko sirf woh maloomat mile jo uske **role aur scope** ke mutabiq ho. Agar agent sirf mausam ke bare mein hai, to aik stock market query ko input guardrail roak sakta hai.
    3. **Data Quality and Formatting (Data Quality aur Formatting):** Input data ki quality check karna aur yaqeeni banana ke woh sahi format mein ho, taake LLM behtar tareeqay se process kar sake.
    4. **Prompt Injection Prevention (Prompt Injection Se Bachao):** Bad-niyat sarifeen (malicious users) LLM ke original instructions ko badalne ki koshish kar sakte hain (prompt injection). Input guardrails aisi koshishon ko roak sakte hain.
- **Kaise Kaam Karta Hai:** Input guardrails aam taur par keyword filtering, regex patterns, AI-based content moderation models, ya semantic analysis ka istemal karte hain taake incoming text ko scan aur analyze kiya ja sake.

Agar koi red flag milta hai, to input ko block kar diya jata hai, ya sanitize kiya jata hai, ya user ko warning di jati hai.

- **Misal:**
  - Agar user aik prompt deta hai jis mein dhamki (threat) hai, to `InputGuardrail` usay rok dega aur LLM tak pahunchne nahin dega.
  - Aik customer service bot ka `InputGuardrail` customer ke credit card details type karne par rok dega taake sensitive data LLM ke logs mein na jaye.

Output Guardrail ()

- **Amal Ka Maqam (Point of Operation):** `OutputGuardrail` **LLM ke jawab dene ke baad** us generated output par lagoo hota hai, **lekin user tak pahunchne se pehle**. Yeh LLM ke "outbox" ka darwaza hai.
- **Maqsad-e-Control (Purpose of Control):**
  1. **Safety and Harm Prevention (Safety aur Nuqsan Se Bachao):** Yeh yaqeeni banana hai ke LLM ka jawab **nuksan-deh, ghalat (misleading), ya policy ki khilaf warzi karne wala** na ho. LLM kabhi ghalti se offensive, biased, ya dangerous content generate kar sakta hai. Output guardrail isay roakta hai.
  2. **Relevance and Consistency (Relevance aur Consistency):** LLM ke jawab ko check karna ke woh user ki original query aur agent ke defined role ke mutabiq hai. Agar LLM "hallucinate" karta hai (galat maloomat deta hai), to output guardrail isay pakad sakta hai.
  3. **Policy Compliance (Policy Ki Pabandi):** System ki policies (maslan, "no medical advice," "no financial advice") ko enforce karna.
  4. **Format Compliance (Format Ki Pabandi):** Yeh yaqeeni banana ke jawab munasib format mein ho (maslan, JSON, bullet points) jaisa ke system prompt mein bataya gaya tha.
  5. **Factuality Check (Haqeeqat Ki Jaanch):** Baaz auqat, output guardrails factual accuracy (haqeeqat ki durustagi) ko bhi check kar sakte hain, khaas kar ke sensitive domains mein.
- **Kaise Kaam Karta Hai:** Output guardrails bhi content moderation techniques, semantic analysis, fact-checking mechanisms, aur policy rules ka istemal karte hain. Agar output mein koi masla milta hai, to usay block kiya ja sakta hai, edit kiya ja sakta hai, ya ek disclaimer ke sath forward kiya ja sakta hai.
- **Misal:**
  - Agar LLM ghalti se aisa jawab deta hai jo sensitive ya offensive hai, to `OutputGuardrail` usay user tak pahunchne se pehle roak dega.
  - Aik AI doctor bot ka `OutputGuardrail` yeh yaqeeni banayega ke LLM koi medical advice na de, balkay sirf jankari faraham kare aur user ko asli doctor se rabta karne ka kahe.

Summary ()

`InputGuardrail` aur `OutputGuardrail` dono AI system ki safety aur effectiveness ke liye zaroori hain, lekin woh **data flow ke mukhtalif marhalon par kaam karte hain**:

Feature	Input Guardrail	Output Guardrail
Amal Ka Maqam	**LLM tak pahunchne se pehle** user input par.	**LLM ke jawab dene ke baad** user tak pahunchne se pehle.
Maqsad-e-Control	LLM ko **harmful, irrelevant, ya malicious content** se bachana.	LLM ke jawab ko **harmful, incorrect, ya policy-violating** hone se bachana.
Buniyadi Sawal	"Kya yeh input LLM ke liye safe aur munasib hai?"	"Kya yeh output user ke liye safe, sahi, aur munasib hai?"
Khatra Jis Se Bachao	Prompt injection, offensive input, sensitive data leakage.	Hallucinations, biased/offensive output, policy violations.
Action	Input ko block, sanitize, ya warning.	Output ko block, edit, ya disclaimer ke sath forward.

Export to Sheets

Mukhtasaran, `InputGuardrail` LLM ko **safai se bachata hai**, jabke `OutputGuardrail` is baat ko yaqeeni banata hai ke LLM ka **jawab khud safai se jawab de**. Yeh dono mil kar aik AI system ko robust aur responsible banate hain.

18. What is the primary purpose of the Instructions Parameter in an Agent ?

Agent Mein instructions Parameter Ka Buniyadi Maqsad Kya Hai?

Definition ()

`instructions` **parameter** aik **text-based input** hai jo Large Language Model (LLM) based agents ko initialize karte waqt ya unki functionality configure karte waqt faraham kiya jata hai. Iska buniyadi maqsad **agent ko uske role, maqsad, qawaid, aur tareeqa-e-kaar ke bare mein aam (general) aur comprehensive hidayat (instructions) dena** hai. Yeh agent ke **core system prompt** ka kaam karta hai.

Explanation ()

Jab aap aik AI agent banate hain, khaas kar ke jo LLM par mabni ho, to usay batana padta hai ke usay kya karna hai, kis tarah karna hai, aur uski hudood kya hain. Yahi kaam `instructions` parameter karta hai.

`instructions` **parameter** ka **buniyadi maqsad** hai:

"Agent (LLM) Ko Us Ke Buniyadi Kirdar, Maqsad, aur Amal Ke Qawaid Ke Bare Mein Mustaqil aur Comprehensive Hidayat Faraham Karna Taake Woh Behtar Tareeqay Se User Ki Queries Ko Handle Kar Sake." (To Provide the Agent (LLM) with Consistent and Comprehensive Instructions Regarding Its Core Role, Purpose, and Operating Rules, Enabling It to Handle User Queries More Effectively.)

Iska matlab hai ke yeh parameter LLM ko uske "identity" (pehchan) aur "mandate" (ikhtiyarat) ke bare mein batata hai, jo har interaction ke dauran uske zehen mein rehta hai.

1. **Core Identity and Role (Buniyadi Pehchan aur Kirdar):**
   - **Faida:** `instructions` agent ko batata hai ke woh kaun hai. Maslan, "You are a helpful travel assistant," ya "You are a polite customer support chatbot." Yeh agent ke jawab dene ke andaaz (tone), uske focus, aur uski umumil qism (general nature) ko set karta hai.
   - **Misal:** Agar `instructions` mein hai "You are a friendly and encouraging tutor," to agent us tone mein baat karega.
2. **Primary Objective (Buniyadi Maqsad):**
   - **Faida:** Yeh LLM ko uske sabse aham maqsad ke bare mein batata hai. Agent har halat mein is maqsad ko haasil karne ki koshish karega.
   - **Misal:** "Your primary goal is to help users find the best flight options."
3. **General Constraints and Rules (Aam Pabandiyan aur Qawaid):**
   - **Faida:** `instructions` mein woh aam qawaid shamil hote hain jin par agent ko har waqt amal karna hota hai. Is mein safety guidelines, ethical considerations, ya privacy rules shamil ho sakte hain. Yeh woh pabandiyan hain jo kisi khaas tool se mutasir nahin hoti balkay poore agent ke behaviour par lagoo hoti hain.
   - **Misal:** "Do not engage in political discussions." ya "Always maintain user privacy."

4. **Implicit Tool Usage (Ghair Wazeh Tool Ka Istemal):**
    - **Faida:** Jabke tools ki specific details (`name`, `description`, `parameters_schema`) alag se LLM context mein shamil ki jati hain (jaise `tool_use_behavior` ya `get_system_prompt()` ke zariye), `instructions` LLM ko tools ke bare mein **general approach** bata sakta hai.
    - **Misal:** "Use the available tools whenever necessary to fulfill the user's request, but confirm with the user before performing any irreversible actions."
5. **Output Guidelines (Output Ki Hidayat):**
    - **Faida:** Yeh LLM ko aam hidayat de sakta hai ke usay kis tarah ke jawab dene hain. Maslan, "Be concise," "Ask clarifying questions if needed," ya "Always summarize the information."

`instructions` Parameter vs. `get_system_prompt()` Method:

Aam taur par, `instructions` parameter hi woh raw text hota hai jo `get_system_prompt()` method ke andar istemal hota hai (ya us ka buniyadi hissa hota hai). `get_system_prompt()` method aksar is `instructions` string ko tools ke schemas, run context ki maloomat (jaise current date), aur doosri dynamic details ke sath mila kar final system prompt banata hai jo LLM API ko bheja jata hai.

`instructions` parameter agent ki zindagi ke dauran tabdeel nahin hota (jab tak explicitly na kiya jaye). Yeh uski buniyadi shakhsiyat aur maqsad ko define karta hai.

Example Code ()

Chaliye aik `CustomerSupportAgent` ki misal dekhte hain jo `instructions` parameter ka istemal karta hai.

```
import json
from typing import Dict, Any, List, Optional

# --- Hypothetical Tool (simplified for this example) ---
# Normally, tools would be defined with full schemas for LLM parsing.
# Here, we're just showing a dummy tool function.
def lookup_order_status(order_id: str) -> Dict[str, str]:
    """Looks up the status of a customer order."""
    print(f"\n[Tool Execution]: Looking up order ID: {order_id}")
    if order_id == "ORD123":
        return {"status": "shipped", "tracking_number": "TRACK456", "estimated_delivery": "2025-
06-25"}
    return {"status": "not found", "message": "Order ID not recognized."}

# --- Agent Class ---
class CustomerSupportAgent:
    def __init__(self, agent_name: str, instructions: str):
        self.agent_name = agent_name
        self._base_instructions = instructions # This is our 'instructions' parameter

        # Tools would be formally registered in a real SDK
        self.available_tools_schema = [
            {
                "name": "lookup_order_status",
                "description": "Looks up the status of a customer order by its ID.",
                "parameters": {
                    "type": "object",
                    "properties": {"order_id": {"type": "string"}},
                    "required": ["order_id"]
```

```
                }
            }
        ]
        self.tool_functions = {"lookup_order_status": lookup_order_status}

    def get_system_prompt(self) -> str:
        """
        Combines the base instructions with dynamic elements like tool definitions
        to form the full system prompt for the LLM.
        """
        tools_str = "\n".join([
            f"- Tool Name: {tool['name']}\n  Description: {tool['description']}\n  Parameters:
{json.dumps(tool['parameters'])}"
            for tool in self.available_tools_schema
        ])

        system_prompt = (
            f"{self._base_instructions}\n\n" # Primary instructions
            f"You have access to the following tools:\n{tools_str}\n\n"
            f"When you use a tool, respond with a JSON object like: "
            f"`{{\"tool_call\": {{\"name\": \"tool_name\", \"parameters\": {{...}}}}}}`. "
            f"After a tool call, provide a polite summary of the result."
        )
        return system_prompt

    def simulate_llm_response(self, user_query: str) -> str:
        """
        Simulates how an LLM would respond based on its system prompt and user query.
        """
        system_prompt = self.get_system_prompt()
        print(f"\n--- LLM's System Prompt ---")
        print(system_prompt)
        print(f"----------------------------")
        print(f"User Query: '{user_query}'")

        # --- Hypothetical LLM Logic ---
        # A real LLM would decide based on the combined system prompt and query.
        if "order status" in user_query.lower() and "ord" in user_query.lower():
            order_id = user_query.upper().split("ORD")[-1].strip()
            # If LLM decides to use the tool
            return (f'{{"tool_call": {{"name": "lookup_order_status", "parameters": {{"order_id":
"ORD{order_id}"}}}}}}')
        elif "hello" in user_query.lower():
            return "Hello! How can I help you with your order today?"
        else:
            return f"I am a customer support agent. Please ask me about your order status or
general inquiries."

    def process_user_input(self, user_input: str) -> str:
        """Simulates the end-to-end processing by the agent."""
        llm_response = self.simulate_llm_response(user_input)

        # This part would typically be handled by the Agent SDK's orchestration
        if "tool_call" in llm_response:
            try:
                tool_data = json.loads(llm_response)['tool_call']
                tool_name = tool_data['name']
                tool_params = tool_data['parameters']

                if tool_name in self.tool_functions:
                    tool_result = self.tool_functions[tool_name](**tool_params)
                    if tool_result["status"] == "shipped":
                        return (f"Your order (ID: {tool_params['order_id']}) has been shipped. "
                                f"Tracking number: {tool_result['tracking_number']}. "
```

```
                                f"Estimated delivery: {tool_result['estimated_delivery']}. Is
there anything else?")
                    elif tool_result["status"] == "not found":
                        return f"I'm sorry, I couldn't find an order with ID:
{tool_params['order_id']}. Please double-check the ID."
                else:
                    return f"Error: Agent tried to call an unknown tool: {tool_name}"
            except json.JSONDecodeError:
                return "Agent generated an invalid tool call format."
            except Exception as e:
                return f"An error occurred during tool execution: {e}"
        else:
            return llm_response # Direct text response from LLM

# --- Main Execution ---
if __name__ == "__main__":
    # The 'instructions' parameter is passed here.
    agent_instructions = (
        "You are a friendly and efficient customer support agent for an e-commerce company. "
        "Your primary goal is to assist customers with their order inquiries and general
questions. "
        "Always be polite and clear. If you need more information, politely ask the user for it. "
        "Do not provide personal financial advice or handle payment details directly."
    )
    customer_agent = CustomerSupportAgent(agent_name="SupportBot",
instructions=agent_instructions)

    print("\n--- Scenario 1: User asks for order status ---")
    response1 = customer_agent.process_user_input("What is the status of my order ORD123?")
    print(f"\nAgent's Final Response: {response1}")

    print("\n" + "="*70 + "\n")

    print("\n--- Scenario 2: User asks for unknown order status ---")
    response2 = customer_agent.process_user_input("What is the status of my order ORD999?")
    print(f"\nAgent's Final Response: {response2}")

    print("\n" + "="*70 + "\n")

    print("\n--- Scenario 3: User asks a general greeting ---")
    response3 = customer_agent.process_user_input("Hello there!")
    print(f"\nAgent's Final Response: {response3}")

    print("\n" + "="*70 + "\n")

    print("\n--- Scenario 4: User asks out of scope question (guided by instructions) ---")
    response4 = customer_agent.process_user_input("Can you recommend a good investment strategy?")
    print(f"\nAgent's Final Response: {response4}")
```

Code Example ki Wazahat ():

1. **`CustomerSupportAgent` Class:**
    - `self._base_instructions = instructions`: Constructor mein, `instructions` parameter ko agent ke internal `_base_instructions` variable mein store kiya jata hai.
    - **`get_system_prompt()` Method:** Yeh method asal mein final `system_prompt` string banata hai jo LLM ko bheja jayega. Ismein `self._base_instructions` (hamara `instructions` parameter) ko shamil kiya gaya hai, sath hi available tools ki details aur tool usage ke qawaid bhi shamil hain.
2. **`simulate_llm_response()`:** Yeh simulate karta hai ke LLM kis tarah system prompt (jo `instructions` se banta hai) aur user query ko parh kar jawab generate karta hai. Aap dekh sakte hain ke `system_prompt` mein woh sari hidayat mojood hain jo `instructions` parameter mein di gayi thin.

3. **`process_user_input()`:** Yeh user ke input ko agent ke zariye process karne ka end-to-end flow dikhata hai. LLM se milne wale jawab (jo tool call ho sakta hai ya direct text) ko handle kiya jata hai.

Simulation:

- **Scenario 1 & 2:** Agent `instructions` mein diye gaye role ("friendly and efficient customer support agent") aur maqsad ("assist customers with order inquiries") ke mutabiq order status tool ko call karta hai aur uske results ko handle karta hai.
- **Scenario 3:** Agent `instructions` mein di gayi friendly tone ko apnata hai.
- **Scenario 4:** Jab user out-of-scope sawal (investment strategy) poochhta hai, to agent `instructions` mein di gayi pabandi ("Do not provide personal financial advice") ki wajah se us sawal ka jawab dene se inkar kar deta hai.

Is misal se wazeh hota hai ke `instructions` parameter kis tarah agent ke buniyadi behaviour, tone, aur boundaries ko define karta hai, jo har interaction mein uske zehen mein rehta hai.

Summary ()

`Agent` mein `instructions` parameter ka buniyadi maqsad **agent (LLM) ko uske buniyadi kirdar (core role), maqsad (purpose), aur amal ke qawaid (operating rules) ke bare mein mustaqil aur comprehensive hidayat faraham karna** hai.

Yeh aik text string hota hai jo agent ki identity, uske primary goals, aur usay kin hudood mein rehna hai, is sab ko define karta hai. Yeh LLM ke liye aik tarah ka **constant system prompt** hota hai jo har user interaction ke dauran uske faisle sazi ko guide karta hai.

Mukhtasar yeh ke, **`instructions` parameter** LLM-based agent ko uski "personality" aur "job description" deta hai, jis par woh apni har karwai ko buniyad karta hai.

19. What does the reset_tool_choice parameter control ?

`reset_tool_choice` Parameter Kya Control Karta Hai?

Definition ()

`reset_tool_choice` aik boolean parameter (yani `True` ya `False` value) hai jo OpenAI Assistants API mein istemal hota hai. Yeh is baat ko control karta hai ke Assistants API ka system agle turn (ya message exchange) ke liye Large Language Model (LLM) ke **tool selection behavior** ko kis tarah **reset ya modify** karega, khaas kar ke jab Assistant ne pichhle turn mein koi tool call kiya ho ya koi khaas tool use karne ka faisla kiya ho.

Explanation ()

OpenAI Assistants API mein, jab aap `tools` define karte hain, to Assistant (jo asal mein LLM hota hai) unhein kab aur kaise istemal kare, iske liye aik `tool_choice` setting hoti hai. Yeh setting LLM ko batati hai ke usay agle jawab mein kya karna chahiye: kya koi tool call karna hai, ya user ko seedha jawab dena hai, ya kisi khaas tool ko istemal karna hai.

`reset_tool_choice` parameter ka **buniyadi maqsad** hai:

"Assistant Ke Agle Turn Ke Liye Tool Selection Behavior Ko Restore Ya Badlna, Khaas Taur Par Pichhle Explicit Tool Call Ke Baad."

(To Restore or Change the Assistant's Tool Selection Behavior for the Next Turn, Especially After a Previous Explicit Tool Call.)

Iska matlab hai ke yeh parameter OpenAI Assistants system ko is baat ki hidayat deta hai ke woh agli dafa LLM ke tool selection ko kaise manage kare.

Aaiye iski tafseel dekhte hain:

1. **Default `tool_choice` Behavior (Default `tool_choice` Behavior):**
   - Jab aap `tool_choice` parameter ko explicitly set nahin karte hain, to default behavior **`"auto"`** hota hai. Iska matlab hai ke Assistant khud (LLM ki intelligence ki bunyad par) faisla karta hai ke user ki query ka jawab dene ke liye tool ki zarurat hai ya nahin. Yeh sabse aam aur flexible tareeqa hai.
2. **Explicit `tool_choice` (Wazeh `tool_choice`):**
   - Kuch scenarios mein, aap Assistant ko explicitly bata sakte hain ke woh **zaroor** koi tool call kare (`"required"`), ya **kisi khaas tool ko hi call kare** (maslan, `{"type": "function", "function": {"name": "my_specific_tool"}}`). Yeh "explicit tool choice" LLM ke default `auto` behavior ko override karta hai.
3. **`reset_tool_choice` Ka Role (Role of `reset_tool_choice`):**
   - **Pichle Explicit Tool Choice Ko Khatam Karna:** Agar Assistant ko pichhle API call mein koi "explicit `tool_choice`" di gayi thi (maslan, usay zabardasti aik khaas tool call karwaya gaya tha ya koi bhi tool call na karne par force kiya gaya tha), to **`reset_tool_choice=True`** karne se yeh explicit setting khatam ho jati

hai. Assistant (LLM) wapis **`"auto"` mode** mein aa jata hai aur khud faisla karta hai ke tool istemal karna hai ya nahin.

- **Tool Behavior Ko Restore Karna:** Yeh Assistant ko uski default, khud-mukhtar tool selection behavior par wapis le aata hai. Jab Assistant aik tool ko successfully execute kar leta hai aur uska result hasil kar leta hai, to aksar yeh zaroori hota hai ke Assistant ab phir se azaadana faisla le sake ke conversation ko kaise aage barhana hai. Kya mazeed tools ki zarurat hai, ya user ko jawab dena hai, ya kuch aur? `reset_tool_choice` yahi azadi wapis deta hai.
- **Avoidance of Unintended Loops or Forced Behavior (Ghair Mutawaqo Loops Se Bachao):** Agar `reset_tool_choice` na kiya jaye aur pichle turn mein koi explicit `tool_choice` setting thi (maslan, "always call this tool"), to Assistant ghalti se dobara usi tool ko call kar sakta hai, ya bar bar aik hi tarah ke tool calls mein phans sakta hai, chahe uski zarurat na bhi ho. `reset_tool_choice` isse bachata hai.

Kahan istemal hota hai?

`reset_tool_choice` aam taur par **`beta/threads/{thread_id}/runs`** **ya** **`beta/threads/{thread_id}/messages`** **API calls** mein istemal hota hai. Khaas kar ke:

- **Tool Output Ke Baad:** Jab aap Assistant ko tool ka output provide karte hain (yani user ne tool call generate kiya, aap ne usay execute kiya, aur ab aap output wapas Assistant ko de rahe hain), to us waqt `reset_tool_choice=True` set karna aik common practice hai. Isse Assistant tool output ko analyze karta hai aur phir `auto` mode mein rehte hue agla munasib action leta hai (ya to final response deta hai, ya mazeed tools call karta hai).
- **Kisi Specific Action Ke Baad:** Agar aap ne Assistant ko temporarily kisi khaas tareeqay se behave karne par majboor kiya tha (maslan, sirf aik specific tool call karne ke liye), aur ab aap chahte hain ke woh apni normal, intelligent decision-making par wapis aa jaye, to aap `reset_tool_choice=True` istemal kar sakte hain.

Example Code ()

Chaliye OpenAI Assistants API ke context mein **`reset_tool_choice`** ka istemal dekhte hain. Yaad rahe, yeh server-side code hoga jo client aur OpenAI API ke darmiyan coordination karega.

```
from openai import OpenAI
import os
import time
import json
from dotenv import load_dotenv

load_dotenv() # Load environment variables from .env file (for OPENAI_API_KEY)

# Initialize OpenAI client
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

# --- Simplified Tool Definition ---
def get_current_time(location: str) -> str:
    """Returns the current time for a given location."""
    current_time_str = time.strftime("%H:%M:%S %Z", time.localtime()) # Get current local time
    return f"The current time in {location} is {current_time_str}."

# --- Helper to register and manage tools ---
available_functions = {
    "get_current_time": get_current_time
}

def call_tool_function(tool_call):
```

```
    function_name = tool_call.function.name
    function_args = json.loads(tool_call.function.arguments)

    if function_name in available_functions:
        function_to_call = available_functions[function_name]
        return function_to_call(**function_args)
    else:
        return f"Error: Tool '{function_name}' not found."

# --- Main Assistant Logic ---
async def run_assistant_flow():
    # 1. Create an Assistant (if not already created)
    # This step is typically done once.
    try:
        my_assistant = client.beta.assistants.retrieve("asst_YOUR_ASSISTANT_ID") # Use an existing
ID
        print(f"Using existing Assistant: {my_assistant.id}")
    except Exception:
        print("Creating a new Assistant...")
        my_assistant = client.beta.assistants.create(
            name="Time & Chat Assistant",
            instructions=(
                "You are a helpful assistant that can answer general questions "
                "and tell the current time for various locations using the available tools. "
                "Always be polite and precise."
            ),
            model="gpt-4o", # Or "gpt-4-turbo", etc.
            tools=[
                {"type": "function", "function": {
                    "name": "get_current_time",
                    "description": "Returns the current time for a given location.",
                    "parameters": {
                        "type": "object",
                        "properties": {"location": {"type": "string", "description": "The city or
region"}},
                        "required": ["location"]
                    }
                }}
            ]
        )
        print(f"New Assistant created with ID: {my_assistant.id}")

    # 2. Create a Thread (conversation session)
    print("\nCreating a new Thread...")
    thread = client.beta.threads.create()
    print(f"New Thread created with ID: {thread.id}")

    # --- Scenario 1: Default tool_choice (auto) ---
    print("\n--- Scenario 1: User asks for time (Assistant in auto tool_choice) ---")
    user_message_1 = "What time is it in Karachi right now?"

    print(f"User: {user_message_1}")
    client.beta.threads.messages.create(
        thread_id=thread.id,
        role="user",
        content=user_message_1,
    )

    run = client.beta.threads.runs.create(
        thread_id=thread.id,
        assistant_id=my_assistant.id,
        # tool_choice is 'auto' by default here
    )
```

```
    # Wait for run to complete or require action
    while run.status == "queued" or run.status == "in_progress":
        time.sleep(0.5)
        run = client.beta.threads.runs.retrieve(thread_id=thread.id, run_id=run.id)

    if run.status == "requires_action":
        print("\nAssistant requires tool action (simulating tool execution)...")
        tool_outputs = []
        for tool_call in run.required_action.submit_tool_outputs.tool_calls:
            output = call_tool_function(tool_call)
            tool_outputs.append({
                "tool_call_id": tool_call.id,
                "output": output,
            })
            print(f"Tool Call ID: {tool_call.id}, Output: {output}")

        # Submit tool outputs. Here we use reset_tool_choice=True (which is often default post-
tool-output)
        # but showing it explicitly for clarity of concept.
        print("\nSubmitting tool outputs with reset_tool_choice=True for the next turn.")
        run = client.beta.threads.runs.submit_tool_outputs(
            thread_id=thread.id,
            run_id=run.id,
            tool_outputs=tool_outputs,
            # reset_tool_choice=True # This is often implicitly true after tool_outputs in
Assistants API
                                    # or controlled by higher-level framework.
                                    # For explicit control in specific scenarios, one might use it
in message/run creation.
        )

        while run.status == "queued" or run.status == "in_progress":
            time.sleep(0.5)
            run = client.beta.threads.runs.retrieve(thread_id=thread.id, run_id=run.id)

    messages = client.beta.threads.messages.list(thread_id=thread.id, order="asc")
    final_response = [m for m in messages.data if m.role == "assistant" and m.run_id == run.id]
    print(f"\nAssistant: {final_response[0].content[0].text.value if final_response else 'No
response.'}")

    # --- Scenario 2: Explicit tool_choice (forcing 'none') for next run, then resetting ---
    print("\n" + "="*80 + "\n")
    print("--- Scenario 2: Forcing Assistant NOT to use tools, then resetting ---")

    user_message_2 = "Can you tell me the time in London?"
    print(f"User: {user_message_2}")
    client.beta.threads.messages.create(
        thread_id=thread.id,
        role="user",
        content=user_message_2,
    )

    # Here, we explicitly set tool_choice to "none" for this run.
    # The Assistant is forced NOT to call any tools, even if it could.
    run_forced_none = client.beta.threads.runs.create(
        thread_id=thread.id,
        assistant_id=my_assistant.id,
        tool_choice="none" # Explicitly tells the LLM not to use any tools
    )

    while run_forced_none.status == "queued" or run_forced_none.status == "in_progress":
        time.sleep(0.5)
        run_forced_none = client.beta.threads.runs.retrieve(thread_id=thread.id,
run_id=run_forced_none.id)
```

```
    messages_forced = client.beta.threads.messages.list(thread_id=thread.id, order="asc")
    assistant_response_forced = [m for m in messages_forced.data if m.role == "assistant" and
m.run_id == run_forced_none.id]
    print(f"\nAssistant (Forced 'none'): {assistant_response_forced[0].content[0].text.value if
assistant_response_forced else 'No response.'}")

    print("\n--- Now, let's reset tool_choice and ask again ---")
    user_message_3 = "Actually, please tell me the time in London now that you can use your
tools."
    print(f"User: {user_message_3}")
    client.beta.threads.messages.create(
        thread_id=thread.id,
        role="user",
        content=user_message_3,
    )

    # THIS IS WHERE reset_tool_choice=True comes into play explicitly.
    # It resets the tool_choice for *this* new run, overriding any previous explicit settings.
    run_reset_choice = client.beta.threads.runs.create(
        thread_id=thread.id,
        assistant_id=my_assistant.id,
        tool_choice="auto", # Explicitly set to auto, or omit for default behavior which is
usually auto
        # Note: In OpenAI Assistants, tool_choice is part of the Run creation.
        # 'reset_tool_choice' is more relevant in certain SDK layers where the framework manages
this.
        # For direct OpenAI API, setting 'tool_choice' explicitly in a new Run effectively serves
        # the purpose of 'resetting' from a previous forced state.
    )

    while run_reset_choice.status == "queued" or run_reset_choice.status == "in_progress":
        time.sleep(0.5)
        run_reset_choice = client.beta.threads.runs.retrieve(thread_id=thread.id,
run_id=run_reset_choice.id)

    if run_reset_choice.status == "requires_action":
        print("\nAssistant requires tool action (simulating tool execution after reset)...")
        tool_outputs_reset = []
        for tool_call in run_reset_choice.required_action.submit_tool_outputs.tool_calls:
            output = call_tool_function(tool_call)
            tool_outputs_reset.append({
                "tool_call_id": tool_call.id,
                "output": output,
            })
            print(f"Tool Call ID: {tool_call.id}, Output: {output}")

        run_final_response = client.beta.threads.runs.submit_tool_outputs(
            thread_id=thread.id,
            run_id=run_reset_choice.id,
            tool_outputs=tool_outputs_reset,
        )

        while run_final_response.status == "queued" or run_final_response.status == "in_progress":
            time.sleep(0.5)
            run_final_response = client.beta.threads.runs.retrieve(thread_id=thread.id,
run_id=run_final_response.id)

    messages_final = client.beta.threads.messages.list(thread_id=thread.id, order="asc")
    final_response_reset = [m for m in messages_final.data if m.role == "assistant" and m.run_id
== run_final_response.id]
    print(f"\nAssistant (After Reset): {final_response_reset[0].content[0].text.value if
final_response_reset else 'No response.'}")
```

```
# To run this example, you'll need your OpenAI API Key set as an environment variable
(OPENAI_API_KEY)
# and a created Assistant ID. Replace "asst_YOUR_ASSISTANT_ID" with your actual Assistant ID.
# Asynchronous execution for a cleaner demonstration.
if __name__ == "__main__":
    import asyncio
    asyncio.run(run_assistant_flow())
```

Code Example ki Wazahat ():

1. **OpenAI Client and Assistant Setup:** Code Assistant ko initialize karta hai (ya retrieve karta hai agar pehle se bana ho) aur `get_current_time` tool ko register karta hai.
2. **Thread Creation:** Har conversation ke liye aik `Thread` banaya jata hai.
3. **Scenario 1: Default `tool_choice` (`auto`):**
    - User time poochhta hai.
    - `client.beta.threads.runs.create()` call mein koi `tool_choice` parameter set nahin kiya gaya hai. Iska matlab hai ke `tool_choice` default (`auto`) hai.
    - Assistant khud se samajhta hai ke `get_current_time` tool istemal karna hai, `requires_action` status deta hai.
    - Code tool ko execute karta hai aur `submit_tool_outputs` karta hai. Is point par, Assistant apni intelligent decision-making par wapis aa jata hai.
4. **Scenario 2: Explicit `tool_choice` (`none`) for next run:**
    - User dobara time poochhta hai.
    - Lekin is dafa `run_forced_none = client.beta.threads.runs.create(..., tool_choice="none")` mein `tool_choice="none"` set kiya gaya hai.
    - Iski wajah se Assistant tool call karne ke bajaye yeh keh deta hai ke woh tools istemal nahin kar sakta. **Yahan `reset_tool_choice` `False` hai (implicitly, kyunki parameter mojood nahin), isliye force ki gayi setting asar andaz hoti hai.**
5. **Scenario 3: Resetting `tool_choice` for the *next* run:**
    - Ab user phir se time poochhta hai, aur is dafa yeh batata hai ke Assistant tools istemal kar sakta hai.
    - **`run_reset_choice = client.beta.threads.runs.create(..., tool_choice="auto")`**: Yahan, hum explicitly `tool_choice` ko `"auto"` par set karte hain. OpenAI Assistants API mein, `reset_tool_choice` parameter ka maqsad `Run` object ke `tool_choice` parameter ko set karne se hasil hota hai. Jab aap naya `Run` banate hain, to uski `tool_choice` property us run ke liye LLM ke tool selection behavior ko define karti hai. Agar pichhle run mein koi `tool_choice` forced thi, to naye run mein `tool_choice="auto"` set karna us previous force ko override kar deta hai aur LLM ko default behavior par wapis le aata hai.
    - Iski wajah se Assistant ab `get_current_time` tool ko istemal kar sakta hai aur sahi jawab deta hai.

Summary ()

OpenAI Assistants API mein **`reset_tool_choice`** parameter (ya is ka equivalent functionality, jo naye Run mein `tool_choice` parameter ko set karne se hasil hota hai) ka buniyadi maqsad **Assistant (LLM) ke agle turn ke liye tool selection behavior ko restore ya tabdeel karna** hai.

Khaas kar ke:

1. **Pichli Explicit Setting Ko Khatam Karna:** Agar Assistant ko pichle API call mein `tool_choice` parameter ke zariye koi khaas tool call karne par force kiya gaya tha (maslan, `"required"` ya kisi specific tool ke naam par), ya koi bhi tool call na karne par (`"none"`), to `reset_tool_choice=True` (ya naye `Run` mein `tool_choice="auto"` set karna) is zabardasti ko khatam kar deta hai.
2. **`auto` Mode Par Wapis Lana:** Yeh Assistant ko uski default, khud-mukhtar tool selection behavior (`"auto"`) par wapis le aata hai. Jab Assistant tool output ko process kar chuka hota hai, to `reset_tool_choice` usay azaadi deta hai ke woh us output ki bunyad par agla munasib action (final jawab dena ya mazeed tools call karna) khud se faisla kar sake.
3. **Ghair Mutawaqo Loops Se Bachao:** Iske baghair, Assistant ghalti se aik hi tool call ko bar bar dohra sakta hai, ya aise tool calls mein phans sakta hai jin ki ab zarurat nahin hai.

Mukhtasar yeh ke, **`reset_tool_choice`** Assistant ko pichli tool selection hidayat se azad kar ke usay apni 'azad marzi' se agla action (tool call ya text response) lene ki ijazat deta hai.

20. What happens if a function tool raises an exception ?

Kya Hota Hai Agar Aik Function Tool Exception Raise Karta Hai?

Definition ()

Aik **function tool** aik Python function (ya koi bhi executable code) hota hai jise OpenAI Assistants API ke Assistant (LLM) ke zariye call kiya ja sakta hai. Jab hum kehte hain ke aik function tool "exception raise karta hai," to iska matlab hai ke us function ke andar koi ghalti (error) hoti hai jis ki wajah se woh normal tareeqay se execution complete nahin kar pata aur Python mein aik **exception** (jaise `ValueError`, `TypeError`, `Exception`, etc.) phenk deta hai.

Explanation ()

Jab aik Assistant (LLM) kisi function tool ko call karta hai, to woh asal mein aapke code mein define kiye gaye us Python function ko invoke karta hai. Agar us function ke andar koi unexpected situation ya invalid state paida ho jaye, to woh aik exception raise kar sakta hai.

Agar aik function tool exception raise karta hai, to OpenAI Assistants API ke context mein **buniyadi taur par yeh hota hai:**

"Tool Ki Execution Fail Ho Jati Hai, Aur Us Exception Ki Maloomat Assistant (LLM) Ko Wapas Bheji Jati Hai Taake Woh Usay Handle Kar Sake Ya User Ko Bataye."

(The Tool's Execution Fails, and Information About That Exception Is Sent Back to the Assistant (LLM) So It Can Handle It or Inform the User.)

Iska matlab hai ke system is ghalti ko chupata nahin, balkay Assistant ko iske bare mein inform karta hai.

Aaiye is process ko tafseel se dekhte hain:

1. **Tool Execution Request (Tool Execution Ki Darkhwast):**
   - Assistant (LLM) user ki query par base karte hue aik tool call generate karta hai (maslan, `{"tool_calls": [{"name": "my_tool", "parameters": {"arg1": "value"}}]}`).
   - Assistants API is tool call ko `requires_action` status ke sath aapke application ko wapas karta hai.
2. **Your Application's Role (Aapki Application Ka Kirdar):**
   - Aapka application (server-side code) `requires_action` status ko detect karta hai.
   - Phir, aapka code `tool_call.function.name` aur `tool_call.function.arguments` ko extract karta hai.
   - Aapka code us tool function ko call karta hai (maslan, `my_tool(value)`).
3. **Exception Raises (Exception Ka Uthna):**
   - **Agar function tool ke andar koi ghalti hoti hai (maslan, invalid input, network issue, division by zero), to woh aik exception raise kar deta hai.**
4. **Handling the Exception in Your Application (Aapki Application Mein Exception Ko Handle Karna):**
   - **Yeh sabse aham qadam hai.** Aapke code ko tool function call ko **`try-except` block** mein wrap karna chahiye.

- Agar exception catch ho jati hai, to aapko us exception ki maloomat ko **tool output** ke tor par format karna chahiye aur Assistant API ko `submit_tool_outputs` ke zariye wapas bhejna chahiye.
- **Important:** Aapko tool output mein clear message dena chahiye ke tool execution fail ho gayi hai aur uski wajah kya thi.

5. **Assistant (LLM) Receives Error (Assistant (LLM) Ko Ghalti Milti Hai):**
   - Jab Assistant tool output receive karta hai (jismein error message hota hai), to LLM usay parse karta hai.
   - **LLM Is Error Ko Kaise Handle Karta Hai:**
     - **User Ko Notify Karna:** LLM aam taur par user ko polite tareeqay se batayega ke tool kaam nahin kar saka aur uski wajah kya thi.
     - **Alternative Strategy (Mutbadil Hikmat-e-Amli):** LLM koshish kar sakta hai ke us error ko handle karne ke liye koi mutbadil tareeqa apnaye, maslan, agar aik API fail ho jaye, to doosre tool ko try kare, ya user se mazeed maloomat mange.
     - **Task Abandonment (Task Ko Chhod Dena):** Bohat shadeed ghalti ki surat mein, LLM task ko mukammal karne se inkar kar sakta hai.
     - **Debugging Information:** Developer ke liye, tool outputs mein complete error message (traceback nahin, sirf summary) provide karna debugging mein madadgar hota hai.

Kyun Yeh Tareeqa Behtar Hai?

- **Transparency (Shiffafiyat):** LLM ko pata hota hai ke tool kyun fail hua, jisse woh zyada intelligent aur relevant jawab de sakta hai.
- **Robustness (Mazbooti):** System unexpected tool failures ko graceful tareeqay se handle karta hai, bajaye iske ke poora conversation crash ho jaye.
- **User Experience (Sarif Ka Tajurba):** User ko aik clear error message milta hai, bajaye iske ke system khamosh ho jaye ya ghalat jawab de.
- **Debugging:** Developer ke liye tool output mein error details dekhna debugging mein asani paida karta hai.

Example Code ()

Chaliye aik misal dekhte hain jahan aik function tool exception raise karta hai aur hum use kaise handle karte hain. Hum OpenAI Assistants API ke structure ko simulate karenge.

```
from openai import OpenAI
import os
import time
import json
from dotenv import load_dotenv

load_dotenv()

# Initialize OpenAI client (replace with your actual key if running)
# client = OpenAI(api_key=os.getenv("OPENAI_API_KEY")) # Uncomment for real API calls

# --- Simplified Tool Definition ---
def divide_numbers(numerator: float, denominator: float) -> Dict[str, Any]:
    """
    Divides two numbers. Raises an error if the denominator is zero.
    """
    print(f"\n[Tool]: Attempting to divide {numerator} by {denominator}...")
    if denominator == 0:
        # Simulate an error condition that would raise an exception
        raise ValueError("Cannot divide by zero!")

    result = numerator / denominator
```

```
    return {"result": result, "status": "success"}

# --- Helper to register and manage tools ---
available_functions = {
    "divide_numbers": divide_numbers
}

# --- Simulate Tool Execution and Error Handling in Your Application ---
def execute_tool_call_with_error_handling(tool_call_obj):
    function_name = tool_call_obj.function.name
    function_args = json.loads(tool_call_obj.function.arguments)

    print(f"\n[Your Application]: Executing tool: {function_name} with args: {function_args}")

    if function_name in available_functions:
        function_to_call = available_functions[function_name]
        try:
            # THIS IS THE KEY PART: Wrap the tool call in a try-except block
            tool_result_output = function_to_call(**function_args)
            return {
                "tool_call_id": tool_call_obj.id,
                "output": json.dumps(tool_result_output) # Output should be a string (JSON or
plain text)
            }
        except Exception as e:
            # If an exception occurs, capture it and send it as tool output
            error_message = f"Tool execution failed: {type(e).__name__}: {str(e)}"
            print(f"[Your Application]: Caught exception during tool execution: {error_message}")
            return {
                "tool_call_id": tool_call_obj.id,
                "output": json.dumps({"status": "error", "message": error_message}) # Send
structured error
            }
    else:
        error_message = f"Error: Tool '{function_name}' not found in available functions."
        print(f"[Your Application]: {error_message}")
        return {
            "tool_call_id": tool_call_obj.id,
            "output": json.dumps({"status": "error", "message": error_message})
        }

# --- Simulate OpenAI Assistant API Interactions ---
class MockAssistantClient:
    def __init__(self):
        self.threads = {}
        self.assistants = {}
        self._next_id = 1

    def create_assistant(self, name, instructions, model, tools):
        assistant_id = f"asst_mock_{self._next_id}"
        self._next_id += 1
        self.assistants[assistant_id] = {
            "name": name, "instructions": instructions, "model": model, "tools": tools
        }
        return type('Assistant', (object,), {'id': assistant_id})() # Mock object

    def create_thread(self):
        thread_id = f"thread_mock_{self._next_id}"
        self._next_id += 1
        self.threads[thread_id] = {"messages": []}
        return type('Thread', (object,), {'id': thread_id})()

    def create_message(self, thread_id, role, content):
        message_id = f"msg_mock_{self._next_id}"
```

```
        self._next_id += 1
        self.threads[thread_id]["messages"].append({"id": message_id, "role": role, "content":
content})
        return type('Message', (object,), {'id': message_id, 'role': role, 'content': content})()

    def create_run(self, thread_id, assistant_id, tool_choice="auto"):
        run_id = f"run_mock_{self._next_id}"
        self._next_id += 1
        self.threads[thread_id]["current_run_id"] = run_id

        # Simplified LLM logic: if user asks to divide and tool is available,
        # LLM decides to call the tool.
        last_message_content = self.threads[thread_id]["messages"][-1]["content"]
        assistant_tools = self.assistants[assistant_id]["tools"]

        if "divide" in last_message_content.lower() and "divide_numbers" in [t['function']['name']
for t in assistant_tools]:
            parts = last_message_content.lower().split("divide")
            num_str = parts[0].strip().split("by")[0].replace("please", "").replace("calculate",
"").strip()
            den_str = parts[1].strip()

            try:
                numerator = float(num_str)
                denominator = float(den_str)
            except ValueError:
                return type('Run', (object,), {'id': run_id, 'status': 'completed', 'output':
'Please provide valid numbers for division.'})()

            # Simulate LLM generating tool_call
            mock_tool_call_id = f"tool_call_mock_{self._next_id}"
            self._next_id += 1
            mock_tool_call = type('ToolCall', (object,), {
                'id': mock_tool_call_id,
                'function': type('Function', (object,), {
                    'name': 'divide_numbers',
                    'arguments': json.dumps({"numerator": numerator, "denominator": denominator})
                })
            })()

            return type('Run', (object,), {
                'id': run_id,
                'status': 'requires_action',
                'required_action': type('RequiredAction', (object,), {
                    'submit_tool_outputs': type('SubmitToolOutputs', (object,), {
                        'tool_calls': [mock_tool_call]
                    })
                })
            })()
        else:
            return type('Run', (object,), {'id': run_id, 'status': 'completed', 'output': 'I can
help with division questions.'})()

    def retrieve_run(self, thread_id, run_id):
        # In a real scenario, this would query the OpenAI API
        # For mock, we assume run status is already determined by create_run
        # and subsequent submit_tool_outputs will change it
        pass # Not fully implemented for simplicity

    def submit_tool_outputs(self, thread_id, run_id, tool_outputs):
        # After tool outputs, LLM processes and gives final response
        # Here we simulate the LLM's final response based on tool output
        output_data = json.loads(tool_outputs[0]['output'])
        if output_data.get("status") == "error":
```

```
            response_content = f"I encountered an error while performing the calculation:
{output_data['message']}. Please try again with valid numbers."
        else:
            response_content = f"The result of your division is: {output_data['result']}."

        # Simulate adding assistant's final response message
        message_id = f"msg_mock_{self._next_id}"
        self._next_id += 1
        self.threads[thread_id]["messages"].append({"id": message_id, "role": "assistant",
"content": response_content, "run_id": run_id})

        return type('Run', (object,), {'id': run_id, 'status': 'completed'})()

    def list_messages(self, thread_id, order="asc"):
        # Returns mock messages. In real API, this would return proper Message objects.
        return type('MessagesList', (object,), {'data': [
            type('Message', (object,), {'role': m['role'], 'content': [type('TextContent',
(object,), {'text': type('Text', (object,), {'value': m['content']})})], 'run_id':
m.get('run_id')})
            for m in self.threads[thread_id]["messages"]
        ]})()

# --- Main Flow Simulation ---
async def main_simulation():
    # Use the Mock client for demonstration
    client = MockAssistantClient()

    # 1. Create Assistant
    my_assistant = client.create_assistant(
        name="Calculator Assistant",
        instructions="You are a calculator assistant that can perform division.",
        model="gpt-4o",
        tools=[
            {"type": "function", "function": {
                "name": "divide_numbers",
                "description": "Divides two numbers.",
                "parameters": {
                    "type": "object",
                    "properties": {
                        "numerator": {"type": "number"},
                        "denominator": {"type": "number"}
                    },
                    "required": ["numerator", "denominator"]
                }
            }}
        ]
    )
    print(f"Assistant created with ID: {my_assistant.id}")

    # 2. Create Thread
    thread = client.create_thread()
    print(f"Thread created with ID: {thread.id}")

    print("\n--- Scenario 1: Tool execution SUCCESS ---")
    user_query_success = "Calculate 10 divided by 2."
    print(f"User: {user_query_success}")
    client.create_message(thread_id=thread.id, role="user", content=user_query_success)

    run_success = client.create_run(thread_id=thread.id, assistant_id=my_assistant.id)

    if run_success.status == "requires_action":
        tool_outputs_success = []
        for tool_call in run_success.required_action.submit_tool_outputs.tool_calls:
```

```
            output_data = execute_tool_call_with_error_handling(tool_call) # Your application
executes tool
            tool_outputs_success.append(output_data)

        client.submit_tool_outputs(thread_id=thread.id, run_id=run_success.id,
tool_outputs=tool_outputs_success)

    messages_success = client.list_messages(thread_id=thread.id, order="asc")
    final_response_success = [m for m in messages_success.data if m.role == "assistant" and
m.run_id == run_success.id]
    print(f"\nAssistant: {final_response_success[0].content[0].text.value if
final_response_success else 'No response.'}")


    print("\n" + "="*80 + "\n")

    print("--- Scenario 2: Tool execution FAILURE (Divide by Zero) ---")
    user_query_failure = "Calculate 10 divided by 0."
    print(f"User: {user_query_failure}")
    client.create_message(thread_id=thread.id, role="user", content=user_query_failure)

    run_failure = client.create_run(thread_id=thread.id, assistant_id=my_assistant.id)

    if run_failure.status == "requires_action":
        tool_outputs_failure = []
        for tool_call in run_failure.required_action.submit_tool_outputs.tool_calls:
            output_data = execute_tool_call_with_error_handling(tool_call) # Your application
executes tool, catches error
            tool_outputs_failure.append(output_data)

        client.submit_tool_outputs(thread_id=thread.id, run_id=run_failure.id,
tool_outputs=tool_outputs_failure)

    messages_failure = client.list_messages(thread_id=thread.id, order="asc")
    final_response_failure = [m for m in messages_failure.data if m.role == "assistant" and
m.run_id == run_failure.id]
    print(f"\nAssistant: {final_response_failure[0].content[0].text.value if
final_response_failure else 'No response.'}")

# Run the asynchronous simulation
if __name__ == "__main__":
    import asyncio
    asyncio.run(main_simulation())
```

Code Example ki Wazahat ():

1. **`divide_numbers(numerator, denominator)` Tool:**
    - Yeh hamara function tool hai.
    - Ismein aik condition hai: `if denominator == 0: raise ValueError("Cannot divide by zero!")`. Agar `denominator` zero hai, to yeh function aik `ValueError` exception raise kar dega.
2. **`execute_tool_call_with_error_handling(tool_call_obj):`**
    - **Yahi woh aham hissa hai jo exception ko handle karta hai.**
    - Is function ke andar, `function_to_call(**function_args)` (yani `divide_numbers` ka actual call) ko aik **`try...except Exception as e:` block** mein wrap kiya gaya hai.
    - **Agar `divide_numbers` exception raise karta hai (jaise `ValueError`), to `except` block catch kar leta hai.**
    - `error_message` banayi jati hai jismein exception ka type aur message shamil hota hai (`f"Tool execution failed: {type(e).__name__}: {str(e)}"`).
    - **Output Format:** Is error message ko Assistant API ko wapas bhejne ke liye, hum isay `submit_tool_outputs` ke `output` field mein JSON format mein dalte hain. `{"status": "error", "message": error_message}` jaisi structured output LLM ke liye asani se parse karne wali hoti hai.

3. **`MockAssistantClient`:** Yeh class OpenAI Assistants API ke kuch buniyadi calls ko simulate karti hai (`create_assistant`, `create_thread`, `create_run`, `submit_tool_outputs`, `list_messages`). Iska maqsad asal API calls ki zaroorat ke baghair flow ko samjhana hai. `create_run` method mein, simplified LLM logic hai jo user query ke mutabiq tool call generate karta hai. `submit_tool_outputs` method tool ke output ko simulate karta hai aur uske base par Assistant ka final jawab banata hai.
4. **`main_simulation()`:**
    - **Scenario 1 (Success):** Jab user 10 ko 2 se divide karne ko kehta hai, to `divide_numbers` tool successfully execute hota hai, aur Assistant sahi jawab deta hai.
    - **Scenario 2 (Failure - Exception):** Jab user 10 ko 0 se divide karne ko kehta hai:
        - `MockAssistantClient.create_run` tool call generate karta hai.
        - `execute_tool_call_with_error_handling` function `divide_numbers` ko call karta hai.
        - `divide_numbers` **`ValueError`** raise karta hai.
        - `execute_tool_call_with_error_handling` ka `except` block is `ValueError` ko catch karta hai.
        - Error message ko `{"status": "error", "message": "Tool execution failed: ValueError: Cannot divide by zero!"}` format mein `output` ke tor par wapas bheja jata hai.
        - **Assistant (LLM) is error output ko receive karta hai aur uski bunyad par user ko aik polite aur informative jawab deta hai** ("I encountered an error while performing the calculation: ... Please try again with valid numbers.").

Summary ()

Agar aik **function tool** exception raise karta hai, to OpenAI Assistants API ke context mein:

1. **Aapki Application Exception Ko Catch Karti Hai:** Tool function ke call ko `try-except` block mein wrap karna zaroori hai.
2. **Error Ko Tool Output Mein Faraham Kiya Jata Hai:** Caught exception ki maloomat (maslan, error message, type) ko tool output ke tor par format kiya jata hai (aam taur par JSON string mein `{"status": "error", "message": "..."}` jaisa).
3. **Output OpenAI API Ko Submit Hota Hai:** Yeh formatted error output `client.beta.threads.runs.submit_tool_outputs()` method ke zariye Assistant API ko wapas bheja jata hai.
4. **Assistant (LLM) Error Receive Karta Hai:** LLM is error output ko parhta hai.
5. **LLM Error Ko Handle Karta Hai:** LLM apni intelligence ka istemal karte hue:
    - User ko safai ke sath batata hai ke tool kyun fail hua.
    - Agar mumkin ho to, koi mutbadil tareeqa apnata hai.
    - User se mazeed maloomat talab karta hai.

Buniyadi maqsad yeh hai ke tool ki ghalti ko system ko crash hone diye baghair **safai ke sath Assistant ko wichaar-vimarsh ke liye faraham kiya jaye**, taake woh user ko ek mazboot aur informative tajurba faraham kar sake.

21. What does the clone() method do ?

`clone()` Method Kya Karta Hai?

Definition ()

`clone()` method programming mein aik common pattern hai, khaas kar ke object-oriented programming mein. OpenAI SDK ke context mein, `clone()` method ka matlab hai ke yeh aik **maujooda object ki mukammal ya adhoori copy banata hai**. Yeh naya object asal object ki tarah ki properties aur configuration rakhta hai, lekin yeh aik alag instance hota hai, jis se aap baghair asal object ko mutasir kiye naye object mein tabdeeliyan kar sakte hain.

Explanation ()

Jab aap kisi object ka `clone()` banate hain, to aap us object ki **aik duplicate copy** hasil karte hain. Is duplicate copy ka maqsad yeh hota hai ke aap us mein tabdeeliyan kar saken ya usay mukhtalif maqasid ke liye istemal kar saken, jabke asal object apni pehle wali halat mein barqarar rahe.

`clone()` method ka **buniyadi maqsad** hai:

"Aik Maujooda Object Ki Nayi, Azadaana Copy Banani Taake Aap Us Nayi Copy Mein Tabdeeliyan Kar Saken Baghair Asal Object Ko Mutasir Kiye."

(To Create a New, Independent Copy of an Existing Object So You Can Make Changes to the New Copy Without Affecting the Original Object.)

OpenAI SDK (aur aam programming) mein iski wajahat:

1. **Immutability Ya Configuration Management (Immutability Ya Configuration Ki Nigrani):**
   - Bohat se frameworks mein, objects ki configurations (jaise client settings, Assistant settings) ko immutable (na badalna) banana behtar hota hai. Agar aapko un settings mein thodi si tabdeeli chahiye, to asal object ko badalne ke bajaye, aap uski `clone()` bana kar clone mein tabdeeli karte hain.
   - **Misal:** Agar aapke paas aik `OpenAIClient` object hai jiski default timeout 30 seconds hai, aur aapko sirf aik khaas request ke liye timeout 60 seconds karni hai, to aap client ko `clone()` kar ke clone ki timeout badal sakte hain, jabke asal client ki timeout 30 seconds hi rahegi.
2. **State Preservation (Halat Ka Tahaffuz):**
   - Agar aapke paas aik object hai jis mein koi state (current halat) mojood hai, aur aap us state ko save karte hue usi object ke base par aik naya object banana chahte hain, to `clone()` useful hota hai.
   - **Misal:** Aik `Assistant` object ki settings (instructions, tools, model) ko clone kar ke aap naya Assistant banate hain jo buniyadi taur par aik jaisa ho lekin kuch mukhtalif tools ya instructions ke sath.
3. **Reducing Initialization Overhead (Initialization Ke Bojh Ko Kam Karna):**

- Baaz auqat objects ko initialize karna aik resource-intensive operation ho sakta hai. Agar aapko aik hi tarah ke kai objects chahiye, to har aik ko naye sire se initialize karne ke bajaye, aik object ko `clone()` karna zyada muassir ho sakta hai.

4. **Deep Copy vs. Shallow Copy (Deep Copy vs. Shallow Copy):**
   - Jab `clone()` method ki baat hoti hai, to yeh concept bhi aata hai ke kya yeh **shallow copy** hai ya **deep copy** hai.
     - **Shallow Copy:** Sirf object ki top-level properties ko copy karta hai. Agar object ke andar doosre objects ka reference hai, to copy aur asal dono unhi references ko share karenge. Agar aap nested object mein tabdeeli karenge, to woh asal object mein bhi nazar aayegi.
     - **Deep Copy:** Object aur uske andar mojood tamam nested objects ko recursive tareeqay se copy karta hai. Is tarah clone aur asal object mukammal taur par alag alag hote hain. `clone()` method aam taur par use case ke mutabiq deep ya shallow copy karta hai. OpenAI SDK mein, jab aap configuration objects ko clone karte hain, to aam taur par woh deep copy jaisa behave karte hain, taake aap azadi se tabdeeliyan kar saken.

OpenAI SDK mein `clone()` ka istemal:

OpenAI Python SDK mein, `clone()` method aksar client configuration objects ya models par paya jata hai. Maslan, agar aapki default client settings hain, aur aapko temporarily un settings mein tabdeeli karni hai (jaise aik khaas request ke liye timeout badalna), to aap client object ko `clone()` kar sakte hain.

Example Code ()

Chaliye OpenAI Python SDK ke `Client` object ke context mein **`clone()`** method ka istemal dekhte hain.

```
from openai import OpenAI
import os

# Assuming OPENAI_API_KEY is set in your environment variables
# For demonstration, we'll use a mock client.
# In a real scenario, you'd initialize it like: client =
OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

# --- Mock OpenAI Client for Demonstration ---
class MockOpenAIClient:
    def __init__(self, api_key: str = "default_key", timeout: int = 60, max_retries: int = 2):
        self.api_key = api_key
        self.timeout = timeout
        self.max_retries = max_retries
        print(f"Initialized Mock Client: Timeout={self.timeout}, Retries={self.max_retries}")

    def clone(self, **kwargs):
        """
        Simulates the clone() method of an OpenAI Client object.
        It creates a new instance with existing settings and applies any kwargs overrides.
        """
        # Create a new instance with current settings
        cloned_client = MockOpenAIClient(
            api_key=self.api_key,
            timeout=self.timeout,
            max_retries=self.max_retries
        )
```

```
        # Apply any overrides from kwargs
        for key, value in kwargs.items():
            if hasattr(cloned_client, key):
                setattr(cloned_client, key, value)
                print(f"  Cloned Client: Setting '{key}' to '{value}'")
        return cloned_client

    def make_api_call(self, endpoint: str, data: str):
        """Simulates an API call using the client's current settings."""
        print(f"\n[API Call Simulation] Using client with Timeout={self.timeout},
Retries={self.max_retries}")
        print(f"  Calling '{endpoint}' with data: '{data}'")
        # In a real scenario, this would make the actual API request
        print("  API call simulated successfully.")

# --- Main Demonstration ---
if __name__ == "__main__":
    print("--- Original Client Setup ---")
    # 1. Create an original client with default settings (or specific ones)
    # This represents your main client instance used throughout your application.
    original_client = MockOpenAIClient(api_key="my_secure_api_key", timeout=30, max_retries=3)
    original_client.make_api_call("/chat/completions", "Hello world request")

    print("\n" + "="*50 + "\n")

    print("--- Cloned Client for a Specific Task (e.g., long-running operation) ---")
    # 2. Clone the original client and modify specific settings for the clone
    # We want a higher timeout for a specific, potentially long-running API call.
    # The clone() method allows us to change 'timeout' only for this new instance.
    specific_task_client = original_client.clone(timeout=120)

    # Now use the cloned client for a specific API call
    specific_task_client.make_api_call("/files/upload", "Large file upload data")

    print("\n" + "="*50 + "\n")

    print("--- Verify Original Client Remains Unchanged ---")
    # 3. Verify that the original client's settings are still the same
    # This demonstrates the primary purpose of clone(): non-mutating changes.
    original_client.make_api_call("/models", "List models request")

    print("\n" + "="*50 + "\n")

    print("--- Cloned Client for another Task (e.g., fewer retries for critical call) ---")
    # 4. Clone again for another scenario, demonstrating multiple independent clones
    critical_call_client = original_client.clone(max_retries=0, timeout=10)
    critical_call_client.make_api_call("/moderations", "Sensitive text moderation")

    print("\n" + "="*50 + "\n")

    print("--- Final Check of Original Client ---")
    # The original client is still unaffected.
    original_client.make_api_call("/usage", "Check usage data")
```

Code Example ki Wazahat ():

1. **`MockOpenAIClient` Class:**
    - Yeh class asal OpenAI `Client` object ke behavior ko simulate karti hai. Is mein `api_key`, `timeout`, aur `max_retries` jaisi properties hain.
    - **`clone(**kwargs)` Method:**
        - Yahi woh method hai jiske bare mein hum baat kar rahe hain.
        - Jab `clone()` call hota hai, to sabse pehle yeh `MockOpenAIClient` ka **naya instance** banata hai.

        - Yeh naya instance asal client ki **current settings (timeout, max_retries)** ke sath initialize hota hai.
        - Phir, `**kwargs` ke zariye jo bhi additional arguments (jaise `timeout=120`) diye jate hain, unhein naye `cloned_client` par apply kar diya jata hai. Is tarah, clone ki sirf woh settings badalti hain jo specifically override ki gayi hain.
        - Ahem baat yeh hai ke `cloned_client` aik **mukammal naya object** hai.
2. **Main Demonstration (`if __name__ == "__main__":`)**
    - **Original Client:** Hum aik `original_client` banate hain jiski default `timeout=30` aur `max_retries=3` hain.
    - **Cloning for Specific Task:** Hum `original_client.clone(timeout=120)` call karte hain. Is se aik `specific_task_client` banta hai jiski timeout **120** seconds hai, lekin `original_client` ki `timeout` **30** seconds hi rehti hai. `make_api_call` methods ke print statements se aap dekh sakte hain ke har client apni apni settings istemal kar raha hai.
    - **Verification:** Hum `original_client` ko dobara istemal karte hain taake yeh confirm ho sake ke uski settings mein koi tabdeeli nahin aayi, jo `clone()` method ka buniyadi fayda hai.
    - **Multiple Clones:** Hum aik aur `critical_call_client` banate hain jismein `max_retries=0` aur `timeout=10` hai. Yeh bhi `original_client` ki copy hai lekin apni tabdeel shuda settings ke sath.

Is misal se wazeh hota hai ke `clone()` method kis tarah aapko aik maujooda object ki mukammal copy bana kar us mein tabdeeliyan karne ki ijazat deta hai, jabke asal object ko mahfooz rakhta hai.

Summary ()

`clone()` method ka buniyadi maqsad **aik maujooda object ki aik nayi aur azadaana (independent) copy banana** hai.

OpenAI SDK jaise frameworks mein, yeh khaas taur par configuration objects (jaise `Client` settings) ke liye mufeed hai. Jab aap kisi object ko `clone()` karte hain:

- **Naya Instance:** Aapko usi type ka aik naya object instance milta hai.
- **Settings Ki Copy:** Naye object mein asal object ki tamam properties aur configurations copy ho jati hain.
- **Azad Tabdeeliyan:** Aap naye clone mein tabdeeliyan kar sakte hain (maslan, `timeout` ya `max_retries` badalna) baghair iske ke asal object mutasir ho.

Yeh aapko flexibility, code ki safai, aur state management mein madad karta hai, khaas kar ke un scenarios mein jahan aapko aik base configuration se shuru kar ke mukhtalif variations chahiye hoti hain.

22. What is ToolsToFinalOutputFunction in the OpenAI Agents SDK ?

OpenAI Agents SDK Mein `ToolsToFinalOutputFunction` Kya Hai?

Definition ()

`ToolsToFinalOutputFunction` (ya is se milta-julata concept jo dusre frameworks mein paya jata hai) OpenAI Agents SDK mein aik **callable entity (function ya object)** hai. Iska buniyadi maqsad **agent ke zariye execute kiye gaye tools ke output ko user ko samajh mein aane wale aur final jawab mein tabdeel karna** hai. Yeh agent ke **orchestration flow** ka aik aham hissa hai.

Explanation ()

Jab aik LLM-based agent tools ka istemal karta hai, to woh aik chain of thoughts ya actions mein kaam karta hai:

1. **User Query:** User aik sawal poochhta hai.
2. **LLM Thinking:** LLM user ki query ko samajhta hai aur faisla karta hai ke iska jawab dene ke liye kya tools chahiye.
3. **Tool Call:** LLM aik ya zyada tools ko call karne ka plan banata hai.
4. **Tool Execution:** Aapka application (SDK ke through) un tools ko execute karta hai.
5. **Tool Output:** Tools kuch output (data, status) wapas karte hain.

Ab sawal yeh hai ke is tool output ka kya kiya jaye? Yahan par `ToolsToFinalOutputFunction` ka kaam aata hai.

`ToolsToFinalOutputFunction` ka **buniyadi maqsad** hai:

"Raw Tool Outputs Ko Process Kar Ke User Ke Liye Aakhri, Samajhne Ke Qabil Jawab Mein Tabdeel Karna."

(To Process Raw Tool Outputs and Transform Them into a Final, Understandable Response for the User.)

Yeh ek bridge ka kaam karta hai **technical tool output** aur **human-readable final answer** ke darmiyan.

Iske key aspects:

1. **Input of Tool Outputs (Tool Outputs Ka Input):**
   - Is function ko executed tools ke outputs receive hotay hain. Yeh outputs mukhtalif formats mein ho sakte hain, maslan, JSON objects, dictionaries, ya plain strings. Yeh aksar aik list hoti hai har tool call aur uske corresponding output ki.
2. **Output Transformation (Output Ki Tabdeeli):**
   - `ToolsToFinalOutputFunction` ke andar ki logic in raw outputs ko leti hai aur unhein aik cohesive (juray hue), relevant, aur concise (mukhtasar) final response mein transform karti hai.
   - Is transformation mein shamil ho sakta hai:
     - **Summarization:** Multiple tool outputs ko summarize karna.
     - **Formatting:** Output ko user-friendly format (maslan, bullet points, structured text) mein tarteeb dena.

- **Contextualization:** Tool output ko user ki original query aur conversation ke context mein rakh kar jawab dena.
- **Error Handling:** Agar tool output mein error messages hain, to unhein user ko samajh mein aane wale tareeqay se bayan karna.
- **Filtering/Extraction:** Sirf zaroori maloomat ko nikalna aur ghair zaroori details ko chhod dena.
- **Natural Language Generation:** Tool data ko natural language mein convert karna.

3. **Human-Readability (Insani Qabil-e-Mutala):**
   - Maqsad yeh hai ke user ko aik aisa jawab mile jo usay samajh mein aaye, bajaye iske ke usay technical JSON data ya raw API responses milen.
4. **Flexibility (Lachak):**
   - Yeh function custom logic define karne ki ijazat deta hai ke aap apne tools ke outputs ko kis tarah final response mein tabdeel karna chahte hain. Har agent ya use case ke liye yeh function mukhtalif ho sakta hai.
5. **Orchestration Flow Mein Position (Orchestration Flow Mein Position):**
   - Yeh function aam taur par **tool execution ke baad aur LLM ke final response generate karne se pehle** ya **LLM ke final response generate karne ke hisse ke taur par** istemal hota hai. Kuch frameworks mein, LLM khud hi tool outputs ko dekhta hai aur `ToolsToFinalOutputFunction` LLM ke jawab ko "polish" karne ke liye istemal hota hai. Doosre frameworks mein, yeh function LLM ke baghair hi tool output ko final jawab mein tabdeel kar deta hai.

Example Code ()

Chaliye aik hypothetical **`ToolsToFinalOutputFunction`** ka istemal dekhte hain OpenAI Agents SDK ke context mein. Hum ek simple weather agent ki misal lenge.

```
from typing import List, Dict, Any, Callable

# --- Mock Tool Output ---
# This is what a tool might return after execution
def mock_weather_tool_output(city: str, unit: str) -> Dict[str, Any]:
    """Simulates raw output from a weather tool."""
    if city.lower() == "karachi":
        if unit == "celsius":
            return {"city": city, "temp": 30, "unit": "celsius", "condition": "Sunny", "humidity":
65}
        else:
            return {"city": city, "temp": 86, "unit": "fahrenheit", "condition": "Sunny",
"humidity": 65}
    elif city.lower() == "london":
        if unit == "celsius":
            return {"city": city, "temp": 15, "unit": "celsius", "condition": "Cloudy",
"humidity": 80}
        else:
            return {"city": city, "temp": 59, "unit": "fahrenheit", "condition": "Cloudy",
"humidity": 80}
    else:
        return {"city": city, "temp": "N/A", "condition": "Unknown", "humidity": "N/A", "error":
"City data not found"}

# --- The ToolsToFinalOutputFunction ---
# This function processes the raw tool outputs and creates a final user-friendly message.
def weather_summary_function(tool_outputs: List[Dict[str, Any]]) -> str:
    """
    Processes the raw outputs from weather tools and generates a human-readable summary.
    This acts as our ToolsToFinalOutputFunction.
    """
    print(f"\n[ToolsToFinalOutputFunction]: Processing tool outputs: {tool_outputs}")
```

```
    if not tool_outputs:
        return "I didn't receive any information from the tools to generate a weather report."

    # Assuming a single tool output for simplicity in this example
    output = tool_outputs[0]

    if output.get("status") == "error" or "error" in output:
        # Handle tool errors gracefully
        return f"I'm sorry, I couldn't get the weather for {output.get('city', 'the requested
location')}. Reason: {output.get('message', output.get('error', 'An unknown error occurred'))}.
Please check the city name."

    city = output.get("city", "the specified city")
    temp = output.get("temp", "N/A")
    unit = "°C" if output.get("unit") == "celsius" else "°F" if output.get("unit") == "fahrenheit"
else ""
    condition = output.get("condition", "unknown conditions")
    humidity = output.get("humidity", "N/A")

    if temp != "N/A":
        return (
            f"The current weather in **{city}** is {temp}{unit} with {condition}. "
            f"Humidity is around {humidity}%."
        )
    else:
        return f"I couldn't retrieve detailed weather information for **{city}**. The conditions
are {condition}."

# --- Agent Orchestration Simulation ---
# This simulates how an agent would use the ToolsToFinalOutputFunction
class WeatherAgentOrchestrator:
    def __init__(self, output_formatter: Callable[[List[Dict[str, Any]]], str]):
        self.output_formatter = output_formatter # This is our ToolsToFinalOutputFunction
        self.available_tools = {
            "get_current_weather": mock_weather_tool_output # Map tool name to its mock function
        }

    def run_query(self, user_query: str):
        print(f"\n--- User Query: '{user_query}' ---")

        # --- Step 1: LLM decides to call a tool (simulated) ---
        # In a real SDK, the LLM would generate the tool call based on its understanding.
        if "weather in karachi" in user_query.lower():
            llm_tool_call = {"name": "get_current_weather", "parameters": {"city": "Karachi",
"unit": "celsius"}}
        elif "weather in london" in user_query.lower() and "fahrenheit" in user_query.lower():
            llm_tool_call = {"name": "get_current_weather", "parameters": {"city": "London",
"unit": "fahrenheit"}}
        elif "weather in unknown city" in user_query.lower():
            llm_tool_call = {"name": "get_current_weather", "parameters": {"city": "Unknown City",
"unit": "celsius"}}
        else:
            print("Assistant: I can only help with weather queries for specific cities.")
            return

        print(f"Assistant (simulated LLM): Decided to call tool: {llm_tool_call['name']} with args
{llm_tool_call['parameters']}")

        # --- Step 2: Your application executes the tool ---
        tool_function = self.available_tools.get(llm_tool_call['name'])
        if tool_function:
            try:
                raw_tool_output = tool_function(**llm_tool_call['parameters'])
```

```
                # In a real Assistants API, this would be submitted as tool_outputs
                # and then the LLM would continue.
                # For this example, we directly pass it to our formatter.
                print(f"Raw Tool Output: {raw_tool_output}")

                # --- Step 3: ToolsToFinalOutputFunction processes the raw output ---
                final_response = self.output_formatter([raw_tool_output]) # Pass as a list, as it
usually expects multiple outputs
                print(f"\nAssistant (Final Response): {final_response}")
            except Exception as e:
                print(f"Error during tool execution: {e}")
                print("Assistant: I'm sorry, I encountered an internal issue while trying to get
the weather.")
        else:
            print(f"Assistant: I don't have a tool named '{llm_tool_call['name']}'.")


# --- Run the Demonstration ---
if __name__ == "__main__":
    print("--- Initializing Weather Agent with weather_summary_function ---")
    weather_agent = WeatherAgentOrchestrator(output_formatter=weather_summary_function)

    # Scenario 1: Successful weather query
    weather_agent.run_query("What is the weather in Karachi?")

    print("\n" + "="*80 + "\n")

    # Scenario 2: Successful weather query with different unit
    weather_agent.run_query("How hot is it in London in Fahrenheit?")

    print("\n" + "="*80 + "\n")

    # Scenario 3: Tool returns an error (simulated city not found)
    weather_agent.run_query("What's the weather in Unknown City?")

    print("\n" + "="*80 + "\n")

    # Scenario 4: User asks a non-weather question (handled by orchestrator before tool call)
    weather_agent.run_query("Tell me a joke.")
```

Code Example ki Wazahat ():

1. **`mock_weather_tool_output(city, unit):`**
    - Yeh hamare **function tool** ki simulation hai.
    - Jab yeh tool execute hota hai, to woh raw data return karta hai (dictionaries jismein `city`, `temp`, `condition`, etc. hain).
    - Ismein aik `error` case bhi hai jahan `Unknown City` ke liye `error` key return ki jati hai.
2. **`weather_summary_function(tool_outputs: List[Dict[str, Any]]) -> str:`**
    - **Yahi hamara `ToolsToFinalOutputFunction` hai.**
    - Iska input `tool_outputs` ki list hai (humne is misal mein sirf aik output expect kiya hai).
    - Yeh function tool outputs ko leti hai aur unhein **parse, summarize, aur format** karti hai.
    - Ismein **error handling logic** bhi shamil hai. Agar tool output mein `error` key mojood hai, to yeh aik friendly error message banata hai.
    - Agar output theek hai, to yeh aik mukammal aur human-readable sentence banata hai ("The current weather in **Karachi** is 30°C with Sunny. Humidity is around 65%.").
3. **`WeatherAgentOrchestrator` Class:**
    - Yeh simulate karta hai ke aik agent framework kis tarah kaam karta hai.
    - `__init__` mein, `output_formatter` (jo hamara `weather_summary_function` hai) ko pass kiya jata hai.

- `run_query` method LLM ke tool call (simulated) ko process karta hai, tool ko execute karta hai, aur phir **`self.output_formatter([raw_tool_output])`** ko call karta hai. Yahan par `ToolsToFinalOutputFunction` asal mein kaam karta hai aur raw tool output ko final user-friendly jawab mein tabdeel karta hai.

Simulation:

- **Scenario 1 & 2 (Success):** User Karachi aur London ke mausam ke bare mein poochhta hai. Mock tool output data deta hai. `weather_summary_function` is data ko parh kar aik mukammal, grammatical, aur easy-to-understand sentence banata hai.
- **Scenario 3 (Tool Error):** User `Unknown City` ke mausam ke bare mein poochhta hai. Mock tool `error` key ke sath output deta hai. `weather_summary_function` is `error` ko detect karta hai aur user ko aik polite error message deta hai ("I'm sorry, I couldn't get the weather...").
- **Scenario 4 (Non-Weather Query):** User ek general sawal poochhta hai. Orchestrator mein hi yeh detect ho jata hai ke yeh weather query nahin, to tool call nahin hota, aur direct text response milta hai.

Is misal se wazeh hota hai ke **`ToolsToFinalOutputFunction`** kis tarah backend tool output ko front-end user-friendly message mein tabdeel karne mein aham kirdar ada karta hai.

Summary ()

OpenAI Agents SDK mein `ToolsToFinalOutputFunction` (ya uska equivalent concept) aik **khas function** hota hai jo **agent ke zariye execute kiye gaye tools ke raw outputs ko user ko samajh mein aane wale aur aakhri jawab mein tabdeel karta hai**.

Iska buniyadi maqsad yeh hai ke:

- **Raw Data Ko Tabdeel Karna:** Tool ke technical outputs ko insani qabil-e-mutala (human-readable) aur meaningful (maqsad-e-bakhsh) jawab mein badalna.
- **Jawab Ko Mukammal Karna:** Tool se milne wali mukhtalif maloomat ko ikattha kar ke aik cohesive aur comprehensive final response banana.
- **User Experience Behtar Banana:** User ko raw code output ya technical data ke bajaye, aik polish kiya hua aur relevant jawab faraham karna.
- **Error Handling:** Tool outputs mein mojood errors ko user-friendly tareeqay se bayan karna.

Mukhtasar yeh ke, **`ToolsToFinalOutputFunction`** **agent ke "communication specialist"** ka kaam karta hai, jo backend ki maloomat ko front-end ki zaban mein tarjuma karta hai.

23. What is the return type of Runner.run() ?

`Runner.run()` Ka Return Type Kya Hai?

Definition ()

OpenAI Assistants API ke context mein, `Runner.run()` method (jo LangChain ya doosre orchestration frameworks mein paya jata hai jo Assistants API ko wrap karte hain) aik asynchronous operation ko execute karta hai jismein aik Assistant (LLM) ko aik thread mein messages ko process karne aur mukammal task ko anjaam dene ki hidayat di jati hai. Is method ka **return type** woh object hota hai jo is poore "run" ka natija (result) ya uski final state ko represent karta hai.

Explanation ()

Jab aap `Runner.run()` call karte hain, to aap Assistant ko aik message thread mein kaam shuru karne ka hukm dete hain. Yeh method sirf aik kaam shuru nahin karta, balkay us kaam ke mukammal hone ke baad uska natija wapas karta hai.

`Runner.run()` ka **buniyadi return type** hai:

"Aik `Run` Object, Jo Asal Mein OpenAI Assistants API Ke `Run` Resource Ko Represent Karta Hai. Yeh Object Assistant Ke Poore Amal Ki Halat (Status), Natija, Aur Zaruri Iqdamat Ki Tafseelat Faraham Karta Hai."

(A `Run` Object, which fundamentally represents the OpenAI Assistants API's `Run` resource. This object provides details about the Assistant's entire operation's status, outcome, and required actions.)

Iska matlab hai ke `Runner.run()` aapko aik object deta hai jiske zariye aap jaan sakte hain ke Assistant ne apna kaam kaisa kiya hai, kya usay mazeed kuch karna hai (maslan, tools execute karna), ya woh mukammal ho gaya hai.

Key aspects of the `Run` object (jo `Runner.run()` return karta hai):

1. **`status` ():**
    - Yeh `Run` object ki sabse ahem property hai. Yeh batata hai ke Assistant is waqt kis marhale mein hai. Possible statuses mein shamil hain:
        - **`queued`**: Run ko line mein laga diya gaya hai.
        - **`in_progress`**: Run chal raha hai.
        - **`requires_action`**: Assistant ko tool outputs ki zarurat hai ya koi aur action perform karna hai (maslan, code interpreter input). **Yeh sabse common status hai jab tools istemal hote hain.**
        - **`cancelling`**: Run cancel ho raha hai.
        - **`cancelled`**: Run cancel ho gaya hai.
        - **`failed`**: Run kisi ghalti ki wajah se mukammal nahin ho saka.
        - **`completed`**: Run mukammal ho gaya hai aur Assistant ne apna kaam kar liya hai (aksar final jawab de diya hai).
        - **`expired`**: Run apni time limit poori kar chuka hai.
    - Aapko is `status` ko check karna hota hai taake pata chale ke agla qadam kya hai.

2. **`id` ():**
    - Run ka unique identifier. Yeh ID doosre API calls (jaise `runs.retrieve()` ya `runs.submit_tool_outputs()`) mein istemal hota hai.
3. **`thread_id` ():**
    - Jis thread par yeh run kiya gaya tha, uska ID.
4. **`assistant_id` ():**
    - Jis Assistant ne yeh run kiya tha, uska ID.
5. **`required_action` ():**
    - Agar `status` `"requires_action"` hai, to yeh property mojood hoti hai. Yeh batati hai ke Assistant ko kya action chahiye. Is mein further details hoti hain, maslan:
        - `submit_tool_outputs`: Agar Assistant ne tools call kiye hain, to un tool calls ki details (`tool_calls`) yahan mojood hoti hain. Aapko un tools ko execute kar ke unka output yahan submit karna hota hai.
6. **Other Properties (Deegar Properties):**
    - `created_at`, `expires_at`, `started_at`, `completed_at`, `failed_at`, `cancelled_at`: Run ki lifecycle ke timings.
    - `last_error`: Agar run `failed` ho jaye, to ismein ghalti ki tafseel hoti hai.
    - `model`, `instructions`, `tools`: Run ke dauran istemal kiye gaye model, instructions, aur tools ki details.

Flow of a Run:

Aam taur par, **`Runner.run()`** se hasil shuda `Run` object ka `status` check kiya jata hai. Jab tak `status` `completed` ya `failed` nahin hota, aapko run ko poll karte rehna hota hai (yani bar bar `client.beta.threads.runs.retrieve()` ko call kar ke status update lena). Agar `status` `"requires_action"` ho jaye, to aap tool calls ko execute karte hain aur `client.beta.threads.runs.submit_tool_outputs()` ke zariye unka output wapas Assistant ko bhejte hain, jis se run resume ho jata hai.

Example Code ()

Chaliye OpenAI Python SDK mein **`client.beta.threads.runs.create()`** (jo **`Runner.run()`** ka equivalent hai) aur uske return type ka istemal dekhte hain.

```
from openai import OpenAI
import os
import time
import json
from dotenv import load_dotenv

load_dotenv()

# Initialize OpenAI client (replace with your actual key)
# Make sure to set OPENAI_API_KEY in your environment variables or .env file
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

# --- Simplified Tool Definition ---
def get_stock_price(symbol: str) -> str:
    """Returns the current stock price for a given ticker symbol."""
    print(f"\n[Tool Execution]: Fetching stock price for {symbol}...")
    if symbol.upper() == "AAPL":
        return json.dumps({"symbol": symbol.upper(), "price": 175.50, "currency": "USD"})
    elif symbol.upper() == "GOOG":
        return json.dumps({"symbol": symbol.upper(), "price": 1800.25, "currency": "USD"})
    else:
```

```
        return json.dumps({"symbol": symbol.upper(), "price": "N/A", "error": "Symbol not found"})

# --- Helper to manage tool functions ---
available_functions = {
    "get_stock_price": get_stock_price
}

def call_tool_function(tool_call):
    function_name = tool_call.function.name
    function_args = json.loads(tool_call.function.arguments)

    if function_name in available_functions:
        function_to_call = available_functions[function_name]
        return function_to_call(**function_args)
    else:
        return f"Error: Tool '{function_name}' not found."

# --- Main Assistant Logic ---
async def run_assistant_flow_with_polling():
    # 1. Create an Assistant (or retrieve an existing one)
    try:
        my_assistant = client.beta.assistants.retrieve("asst_your_stock_assistant_id") # Use an
existing ID
        print(f"Using existing Assistant: {my_assistant.id}")
    except Exception:
        print("Creating a new Assistant...")
        my_assistant = client.beta.assistants.create(
            name="Stock Price Assistant",
            instructions=(
                "You are a helpful assistant that can fetch stock prices using the provided tools.
"
                "Always be precise with the stock symbol."
            ),
            model="gpt-4o", # Or "gpt-4-turbo"
            tools=[
                {"type": "function", "function": {
                    "name": "get_stock_price",
                    "description": "Returns the current stock price for a given ticker symbol.",
                    "parameters": {
                        "type": "object",
                        "properties": {"symbol": {"type": "string", "description": "The stock
ticker symbol, e.g., AAPL"}},
                        "required": ["symbol"]
                    }
                }}
            ]
        )
        print(f"New Assistant created with ID: {my_assistant.id}")

    # 2. Create a Thread
    print("\nCreating a new Thread...")
    thread = client.beta.threads.create()
    print(f"New Thread created with ID: {thread.id}")

    # --- Scenario: User asks for a stock price ---
    user_message = "What is the current stock price of AAPL?"
    print(f"\nUser: {user_message}")
    client.beta.threads.messages.create(
        thread_id=thread.id,
        role="user",
        content=user_message,
    )

    # 3. Create a Run (this is where Runner.run() functionality comes in)
```

```
    print("\nCreating a Run...")
    # The return type of client.beta.threads.runs.create() is an OpenAI Run object.
    current_run = client.beta.threads.runs.create(
        thread_id=thread.id,
        assistant_id=my_assistant.id,
    )
    print(f"Run created with ID: {current_run.id}, initial status: {current_run.status}")

    # 4. Poll the Run status until it completes or requires action
    print("\nPolling Run status...")
    while current_run.status == "queued" or current_run.status == "in_progress":
        time.sleep(0.5) # Wait a bit before polling again
        current_run = client.beta.threads.runs.retrieve(thread_id=thread.id,
run_id=current_run.id)
        print(f"Run status: {current_run.status}")

    # 5. Handle different Run statuses
    if current_run.status == "requires_action":
        print("\nRun requires action: Assistant wants to call a tool.")
        tool_outputs = []
        for tool_call in current_run.required_action.submit_tool_outputs.tool_calls:
            print(f"  Tool Call ID: {tool_call.id}")
            print(f"  Function Name: {tool_call.function.name}")
            print(f"  Arguments: {tool_call.function.arguments}")

            output = call_tool_function(tool_call) # Execute your tool function
            tool_outputs.append({
                "tool_call_id": tool_call.id,
                "output": output, # Output should be a string
            })
            print(f"  Tool Output: {output}")

        # Submit the tool outputs back to the Assistant
        print("\nSubmitting tool outputs...")
        current_run = client.beta.threads.runs.submit_tool_outputs(
            thread_id=thread.id,
            run_id=current_run.id,
            tool_outputs=tool_outputs,
        )
        print(f"Run resumed with status: {current_run.status}")

        # Continue polling until the run completes
        while current_run.status == "queued" or current_run.status == "in_progress":
            time.sleep(0.5)
            current_run = client.beta.threads.runs.retrieve(thread_id=thread.id,
run_id=current_run.id)
            print(f"Run status: {current_run.status}")

    if current_run.status == "completed":
        print("\nRun completed. Retrieving Assistant's final message...")
        messages = client.beta.threads.messages.list(thread_id=thread.id, order="asc")
        # Find the latest assistant message related to this run
        final_assistant_message = None
        for msg in messages.data:
            if msg.role == "assistant" and msg.run_id == current_run.id:
                final_assistant_message = msg

        if final_assistant_message and final_assistant_message.content and
final_assistant_message.content[0].type == "text":
            print(f"\nAssistant: {final_assistant_message.content[0].text.value}")
        else:
            print("\nAssistant did not provide a text response or response is in unexpected
format.")
    elif current_run.status == "failed":
```

```
        print(f"\nRun failed. Last error: {current_run.last_error}")
    else:
        print(f"\nRun ended with unexpected status: {current_run.status}")

# To run this, replace "asst_your_stock_assistant_id" with a valid Assistant ID you've created
# and ensure your OPENAI_API_KEY environment variable is set.
if __name__ == "__main__":
    import asyncio
    asyncio.run(run_assistant_flow_with_polling())
```

Code Example ki Wazahat ():

1. **Assistant & Thread Setup:** Hum aik Assistant (LLM) aur aik conversation thread banate hain. Assistant `get_stock_price` tool istemal kar sakta hai.
2. **`current_run = client.beta.threads.runs.create(...):`**
    - Yahan par `Runner.run()` jaisa functionality call hota hai.
    - Is call ka return type aik **`Run` object** hai (`current_run` variable).
    - Hum foran `current_run.id` aur `current_run.status` ko print karte hain, jo us `Run` object ki properties hain.
3. **Polling Loop (`while current_run.status == "queued" or ...`):**
    - Yeh loop `Run` object ke `status` ko bar bar check karta hai (`client.beta.threads.runs.retrieve()` call ke zariye).
    - Jab tak status `completed` ya `failed` nahin hota, loop chalta rehta hai.
4. **`if current_run.status == "requires_action":`**
    - Agar `Run` ka status `"requires_action"` ho jaye, to iska matlab hai ke Assistant ne tool call kiya hai.
    - Hum `current_run.required_action.submit_tool_outputs.tool_calls` se tool calls ki details nikalte hain.
    - Hum apne `call_tool_function` (jo `get_stock_price` ko call karega) ko execute karte hain.
    - Phir hum `client.beta.threads.runs.submit_tool_outputs()` ko call karte hain, jisko `Run` object ka ID aur tool outputs ki list chahiye hoti hai. Yeh call `Run` object ka status `in_progress` par wapis kar deta hai.
5. **Final Response Retrieval:** Jab `Run` ka status `completed` ho jata hai, to hum `client.beta.threads.messages.list()` se messages retrieve karte hain aur Assistant ka final response print karte hain.

Is misal se wazeh hota hai ke **`Runner.run()`** (ya iska SDK equivalent) aik `Run` object return karta hai, jo Assistant ke task ki progression, zaruri actions, aur final outcome ko track karne ke liye markazi zarya hai.

Summary ()

`Runner.run()` method (OpenAI Assistants API ke SDK wrappers mein) ka return type aik **`Run` object** hota hai.

Yeh `Run` object Assistant ke aik khaas task (user messages ko process karna aur jawab dena) ki poori lifecycle ko represent karta hai. Iski aham properties mein shamil hain:

- **`status`:** Run ki maujooda halat (`queued`, `in_progress`, `requires_action`, `completed`, `failed`, etc.).
- **`id`:** Run ka unique ID.
- **`required_action`:** Agar Assistant ko tools execute karne ki zarurat ho, to ismein un tool calls ki tafseelat hoti hain.

Aap is `Run` object ke `status` ko poll kar ke (bar bar retrieve kar ke) Assistant ke kaam ki progress ko track karte hain, aur jab `status` `"requires_action"` ho jaye, to aap tool calls ko execute kar ke unka output Assistant ko wapas bhejte hain, jis se run dobara shuru ho jata hai. Jab `status` `"completed"` ho jaye, to Assistant ne apna jawab de diya hota hai.

24. What happens if a custom tool behavior function returns is_final_output=False ?

Kya Hota Hai Agar Aik Custom Tool Behavior Function `is_final_output=False` Return Karta Hai?

Definition ()

OpenAI Agents SDK mein, jab aap custom tool behavior define karte hain, to aap aksar aik function banate hain jo tool ke execution aur uske output ko handle karta hai. Is function ka **`is_final_output`** parameter (ya return value) ek boolean flag hai. Jab aapka **custom tool behavior function** **`is_final_output=False`** return karta hai, to iska matlab hai ke **tool ki execution ke baad bhi agent ka task (ya "run") mukammal nahin hua hai aur usay mazeed processing ya LLM ko mazeed input ki zarurat hai.**

Explanation ()

Aik typical agent workflow mein, LLM tools ko use karta hai taake koi task mukammal kar sake. Jab LLM aik tool call karta hai, to aapki application us tool ko execute karti hai, aur phir tool ka output wapas Assistant ko bheja jata hai.

`is_final_output` **ka maqsad** hai:

"Agent Ko Yeh Batana Ke Tool Ka Output Milne Ke Baad Bhi Kya Usay Mazeed Iqdamat Karne Hain Ya Task Abhi Tak Mukammal Nahin Hua Hai."

(To Inform the Agent Whether, Even After Receiving the Tool's Output, It Needs to Take Further Actions or If the Task Is Not Yet Complete.)

Jab aik **custom tool behavior function** **`is_final_output=False`** return karta hai, to buniyadi taur par yeh hota hai:

1. **Run Remains Active (Run Active Rehta Hai):**
   - **`is_final_output=False`** ka matlab hai ke Assistant system ko abhi bhi conversation (ya "run") ko active rakhna chahiye. Task abhi "completed" status tak nahin pahuncha hai.
   - Iske bar'aks, agar **`is_final_output=True`** hota, to Assistant system samjhta ke tool output ke baad task mukammal ho gaya hai aur LLM ko final response dena chahiye ya run ko end karna chahiye.
2. **Tool Output Is Processed by LLM (Tool Output LLM Ke Zariye Process Hota Hai):**
   - Tool se milne wala output (jo **`is_final_output=False`** ke sath aaya hai) LLM ke context mein feed kiya jata hai.
   - LLM is output ko dekhta hai aur phir apni reasoning (soch-vichar) aur instructions ki bunyad par agla qadam uthata hai.
3. **LLM's Next Action (LLM Ka Agla Qadam):**
   - Kyunki **`is_final_output=False`** hai, LLM samjhta hai ke usay abhi mazeed kaam karna hai. Agla qadam ho sakta hai:
     - **Mazeed Tools Call Karna:** Agar pehle tool ka output kisi doosre tool ko call karne ke liye zaroori maloomat provide karta hai. Maslan, aik tool ne `user_id` diya, aur ab doosra tool us `user_id` se `user_details` fetch karega.
     - **User Se Clarifying Question Poochna:** Agar tool output se LLM ko mukammal jawab nahin milta ya use mazeed maloomat chahiye hoti hai user se.

- **Apni Internal State Update Karna:** Tool output ki bunyad par apni internal understanding ko update karna.
- **Partial Response Dena:** Agar LLM ke paas kuch maloomat aa gayi hain aur woh user ko partial update dena chahta hai, jabke complete answer ke liye mazeed processing ki zarurat hai.

4. **No Immediate Final Response (Forced):**
   - Is setting se system LLM ko final, mukammal jawab dene par majboor nahin karta. Yeh LLM ko orchestration flow ko control karne ki ijazat deta hai.

`is_final_output` Kab Use Hota Hai?

`is_final_output` ka concept us waqt aham ho jata hai jab aapka tool flow multiple steps ya conditions par munhasir ho. Misal ke taur par:

- **Multi-Step Operations:** Aik order processing system jismein pehle `check_inventory` tool call hota hai, phir `calculate_price`, aur phir `place_order`. Har step ke baad `is_final_output=False` return hoga jab tak order successfully place na ho jaye.
- **User Confirmation:** Tool ne kuch data fetch kiya, aur ab LLM user se confirm karna chahta hai ke woh proceed kare ya nahin. Tool output aa gaya, lekin final jawab ke liye user input chahiye.
- **Data Validation:** Tool ne input liya, lekin woh valid nahin nikla. Tool output mein error aya, aur LLM ko ab user se re-enter karne ke liye kehna hai.

Example Code ()

Chaliye aik hypothetical agent framework ka flow dekhte hain jahan `is_final_output=False` ka istemal hota hai. Hum aik order fulfillment scenario lenge.

```
from typing import Dict, Any, List, Tuple, Callable

# --- Mock Tools ---
def check_inventory(item_id: str) -> Dict[str, Any]:
    """Checks the inventory level for a given item."""
    print(f"\n[Tool: check_inventory] Checking inventory for {item_id}...")
    if item_id == "LAPTOP":
        return {"item_id": item_id, "available": True, "stock_count": 5}
    elif item_id == "MOUSE":
        return {"item_id": item_id, "available": False, "stock_count": 0, "reason": "Out of
stock"}
    else:
        return {"item_id": item_id, "available": False, "reason": "Item not found"}

def calculate_shipping(item_id: str, quantity: int) -> Dict[str, Any]:
    """Calculates shipping cost based on item and quantity."""
    print(f"[Tool: calculate_shipping] Calculating shipping for {item_id}, qty {quantity}...")
    if item_id == "LAPTOP" and quantity > 0:
        return {"item_id": item_id, "quantity": quantity, "shipping_cost": 25.0, "status":
"calculated"}
    return {"item_id": item_id, "quantity": quantity, "status": "error", "message": "Cannot
calculate shipping for invalid item/quantity."}

# --- Custom Tool Behavior Function ---
# This simulates the part of the SDK that wraps your tool execution
# and decides whether the output is final.
def process_order_tool_behavior(
    tool_name: str,
    tool_args: Dict[str, Any]
```

```
) -> Tuple[Dict[str, Any], bool]: # Returns (tool_output_data, is_final_output)
    """
    Executes a tool and determines if its output is the final one for the overall task.
    """
    print(f"\n[Custom Tool Behavior]: Processing {tool_name}...")
    raw_output = {}

    # Execute the actual tool function
    if tool_name == "check_inventory":
        raw_output = check_inventory(tool_args["item_id"])
        # If item is available, we'll need to calculate shipping next.
        # So, this is NOT the final output for the "order" task.
        if raw_output.get("available"):
            return raw_output, False # Not final
        else:
            # If not available, we can't proceed, so this might be the final (unsuccessful) output
            return raw_output, True # Final (failure)

    elif tool_name == "calculate_shipping":
        raw_output = calculate_shipping(tool_args["item_id"], tool_args["quantity"])
        # After shipping is calculated, we still need to confirm with the user or place order.
        # So, this is NOT the final output.
        if raw_output.get("status") == "calculated":
            return raw_output, False # Not final
        else:
            return raw_output, True # Final (failure)

    else:
        return {"error": "Unknown tool"}, True # If tool not recognized, it's a final error

# --- Simulate LLM Response Generation based on Tool Output and is_final_output ---
def simulate_llm_response(
    tool_outputs_with_final_flag: List[Tuple[Dict[str, Any], bool]]
) -> Tuple[str, bool]: # Returns (response_text, is_task_completed)
    """
    Simulates how LLM would react based on tool output and is_final_output flag.
    """
    if not tool_outputs_with_final_flag:
        return "No tool outputs received.", True

    # Assuming one tool call for simplicity
    last_output, is_final_output = tool_outputs_with_final_flag[0]

    if is_final_output:
        # If is_final_output is True, LLM generates a conclusive response.
        if "error" in last_output:
            return f"I'm sorry, I couldn't complete the request: {last_output.get('message',
last_output.get('error', 'An unknown error occurred'))}. Please try again.", True
        elif "available" in last_output and not last_output["available"]:
            return f"Unfortunately, {last_output['item_id']} is {last_output.get('reason', 'not
available')}.", True
        else:
            return f"Task completed based on last tool output: {last_output}.", True # Should not
happen with current logic for is_final_output=True
    else:
        # If is_final_output is False, LLM analyzes and plans next steps/questions.
        if "available" in last_output and last_output["available"]:
            item_id = last_output["item_id"]
            stock_count = last_output["stock_count"]
            # LLM might now decide to call calculate_shipping or ask user for quantity.
            return (
                f"Okay, {item_id} is in stock ({stock_count} units available). "
                f"Now, I need to calculate shipping. What quantity would you like to order?"
            ), False # Task not completed, waiting for more info/next tool
```

```
        elif "shipping_cost" in last_output:
            cost = last_output["shipping_cost"]
            return (
                f"Shipping cost calculated: ${cost}. "
                f"Would you like to proceed with the order for {last_output['quantity']}x
{last_output['item_id']}?"
            ), False # Not final, needs user confirmation
        else:
            return "Processing in progress, more steps required.", False

# --- Main Orchestration Flow ---
class AgentOrchestrator:
    def __init__(self, tool_behavior_func: Callable[[str, Dict[str, Any]], Tuple[Dict[str, Any],
bool]]):
        self.tool_behavior_func = tool_behavior_func
        self.conversation_state = {} # To store intermediate states if needed

    def process_user_request(self, user_input: str):
        print(f"\n--- User: '{user_input}' ---")

        # Simulate LLM's initial tool call decision
        tool_call_info = None
        if "order laptop" in user_input.lower():
            tool_call_info = {"name": "check_inventory", "args": {"item_id": "LAPTOP"}}
        elif "order mouse" in user_input.lower():
            tool_call_info = {"name": "check_inventory", "args": {"item_id": "MOUSE"}}
        elif "calculate shipping for laptop 2" in user_input.lower():
            tool_call_info = {"name": "calculate_shipping", "args": {"item_id": "LAPTOP",
"quantity": 2}}
        else:
            print("Assistant: I can help with ordering items. Please specify the item.")
            return

        if tool_call_info:
            print(f"Assistant (LLM): Decided to call {tool_call_info['name']} with
{tool_call_info['args']}")

            # Execute the tool behavior function
            tool_output_data, is_final_output = self.tool_behavior_func(
                tool_call_info["name"], tool_call_info["args"]
            )

            # LLM processes the output and the is_final_output flag
            llm_response, task_completed = simulate_llm_response([(tool_output_data,
is_final_output)])

            print(f"\nAssistant: {llm_response}")
            print(f"Task Completed: {task_completed}")
            if not task_completed:
                print("--- Waiting for next user input or further internal action ---")


# --- Run Simulation ---
if __name__ == "__main__":
    orchestrator = AgentOrchestrator(tool_behavior_func=process_order_tool_behavior)

    print("\n--- Scenario 1: Item in stock, requires more steps (is_final_output=False) ---")
    orchestrator.process_user_request("I want to order a Laptop.")

    print("\n" + "="*80 + "\n")

    print("--- Scenario 2: Item out of stock, task ends (is_final_output=True) ---")
    orchestrator.process_user_request("Can I order a Mouse?")
```

```
    print("\n" + "="*80 + "\n")

    print("--- Scenario 3: Shipping calculated, requires confirmation (is_final_output=False) ---
")
    # This simulates a follow-up step after check_inventory
    orchestrator.process_user_request("Calculate shipping for Laptop 2.")
```

Code Example ki Wazahat ():

1. **`check_inventory` & `calculate_shipping` (Mock Tools):**
    - Yeh hamare do hypothetical tools hain jo inventory aur shipping check karte hain.
2. **`process_order_tool_behavior(tool_name, tool_args)` (Custom Tool Behavior Function):**
    - **Yahi woh aham function hai jo `is_final_output` ko return karta hai.**
    - Ismein har tool ke liye logic hai jo yeh faisla karti hai ke us tool ka output final hai ya nahin.
    - **`check_inventory` tool ke liye:**
        - Agar `LAPTOP` available hai (`available=True`), to yeh `False` return karta hai (`return raw_output, False`). Iska matlab hai ke agent ko abhi shipment calculate karni hai, yani task mukammal nahin hua.
        - Agar `MOUSE` available nahin hai (`available=False`), to yeh `True` return karta hai (`return raw_output, True`). Iska matlab hai ke task yahin khatam ho gaya (failure ke sath).
    - **`calculate_shipping` tool ke liye:**
        - Shipment calculate hone ke baad bhi, user ki confirmation ya actual order place karna baqi hai. Isliye, yeh `False` return karta hai (`return raw_output, False`).
3. **`simulate_llm_response(tool_outputs_with_final_flag)`:**
    - Yeh function simulate karta hai ke LLM (ya agent orchestrator) `is_final_output` flag ko kaise read karta hai.
    - Agar `is_final_output` `True` hai, to LLM aik definitive (qatai) jawab deta hai aur `task_completed=True` return karta hai.
    - Agar `is_final_output` `False` hai, to LLM mazeed steps ki nishandahi karta hai (`"Okay, LAPTOP is in stock... Now, I need to calculate shipping."` ya `"Shipping cost calculated... Would you like to proceed?"`) aur `task_completed=False` return karta hai.
4. **`AgentOrchestrator`:**
    - Yeh simulate karta hai ke poora agent flow kaise chalta hai.
    - Yeh `process_order_tool_behavior` function ko call karta hai, aur phir uske result (`tool_output_data` aur `is_final_output`) ko `simulate_llm_response` ko pass karta hai.
    - `task_completed` flag ki bunyad par, yeh print karta hai ke "Task Completed" hai ya "Waiting for next user input".

Simulation:

- **Scenario 1 (`LAPTOP`):**
    - `check_inventory` call hota hai aur `available=True` deta hai.
    - `process_order_tool_behavior` iske liye `is_final_output=False` return karta hai.
    - LLM samajhta hai ke task mukammal nahin hua aur user ko mazeed maloomat ke liye poochhta hai ("What quantity would you like to order?"). `Task Completed: False` print hota hai.
- **Scenario 2 (`MOUSE`):**
    - `check_inventory` call hota hai aur `available=False` deta hai.
    - `process_order_tool_behavior` iske liye `is_final_output=True` return karta hai.
    - LLM samajhta hai ke task mukammal ho gaya (failure ke sath) aur user ko batata hai ("Unfortunately, MOUSE is out of stock."). `Task Completed: True` print hota hai.

- **Scenario 3 (`calculate_shipping`):**
  - `calculate_shipping` call hota hai aur cost deta hai.
  - `process_order_tool_behavior` iske liye `is_final_output=False` return karta hai.
  - LLM samajhta hai ke task mukammal nahin hua (confirmation baqi hai) aur user se poochhta hai ("Would you like to proceed...?"). `Task Completed: False` print hota hai.

Summary ()

Agar aik **custom tool behavior function `is_final_output=False`** return karta hai, to iska matlab hai ke **tool ki execution ke baad bhi agent ka task ya overall goal mukammal nahin hua hai.**

Iske natije mein:

1. **Run Active Rehta Hai:** Assistant ka "run" `completed` status par nahin jata, balkay `in_progress` ya `requires_action` (agar mazeed tool calls hain) jaisi states mein rehta hai.
2. **LLM Mazeed Iqdamat Karta Hai:** Tool se milne wala output LLM ke context mein shamil kiya jata hai, aur LLM apni intelligence ki bunyad par agla munasib qadam uthata hai. Yeh agla qadam mazeed tools call karna, user se sawal poochhna, ya apni internal state ko update karna ho sakta hai.
3. **No Immediate Final Response:** System LLM ko final, qatai jawab dene par majboor nahin karta, balkay usay orchestration flow ko jari rakhne ki ijazat deta hai.

Mukhtasar yeh ke, **`is_final_output=False`** agent ko multi-step workflows aur complex conversations ko manage karne ki lachak faraham karta hai, jahan aik tool call ka natija poore task ko mukammal karne ke liye kafi nahin hota.

25. When would you use StopAtTools instead of stop_on_first_tool ?

`StopAtTools` Ki Bajaye `stop_on_first_tool` Kab Istemal Karenge?

Agent frameworks (jaise LangChain ya custom agent orchestrators jo OpenAI API ko istemal karte hain) mein, `StopAtTools` aur `stop_on_first_tool` dono tareeqe hain jo agent ki execution ko pause karte hain jab woh koi tool istemal karne ka faisla karta hai. Dono mukhtalif scenarios mein istemal hotay hain, jo aapke **tool-calling process par control ki granularity** par munhasir hain.

`stop_on_first_tool` Kya Hai?

`stop_on_first_tool` aam taur par aik seedha-saadha boolean flag hota hai.

- **Definition ():**

  Jab ise `True` par set kiya jata hai, to yeh agent ko hidayat deta hai ke woh **Language Model (LLM) ke pehle tool call ka faisla karne ke foran baad apni execution rokay**. Is se koi farq nahin parta ke LLM aik hi turn mein kitne tools call karna chahta hai; agent sirf pehle tool call par ruk jayega.

- **Explanation ():**

  Tasawwur karen ke LLM soch raha hai: "User ne X poocha hai. Iska jawab dene ke liye, mujhe `tool_A` ko call karna hoga. Oh, aur `tool_A` ke baad, shayad mujhe `tool_B` ki bhi zarurat padegi." Agar `stop_on_first_tool` `True` hai, to agent `tool_A` ke liye call generate karega aur phir **ruk jayega**. Woh isi output turn mein `tool_B` ke liye call generate karne ki koshish bhi nahin karega. Aapki application ko phir `tool_A` ka call milega, usay execute karegi, aur output submit karegi. Uske baad hi agent dobara shuru hoga.

- **Use Cases ():**

  Aap **`stop_on_first_tool`** tab istemal karenge jab:

  - Aap har **individual tool execution par ziyada se ziyada control** chahte hain. Aap pehle tool call ko dekhna, usay process karna, aur shayad custom logic ya user ki dakhal-andazi karna chahte hain, is se pehle ke LLM agle tool calls par ghaur kare.
  - Aapke tools ke **side effects** hain jo baad ke tool choices ko mutasir kar sakte hain. Maslan, `tool_A` koi resource bana sakta hai, aur `tool_B` ko us resource ki ID chahiye. `tool_A` ke baad rokne se aapko woh ID mil jati hai is se pehle ke `tool_B` ko plan bhi kiya jaye.
  - Aap **agent ki tool-calling logic ko step-by-step debug** kar rahe hain.
  - Aapke environment mein **sakht rate limits** ya execution costs hain, aur aap unhein har tool call par tightly manage karna chahte hain.
  - Aapko har tool call ke baad **user se tasdeeq** karne ki zarurat hai (maslan, "Agent aapki email fetch karna chahta hai. Kya yeh theek hai?").

`StopAtTools` Kya Hai?

`StopAtTools` (ya is se milte-julte concepts jinhein run states mein "tool invocation" ya "action steps" kaha jata hai) aik ziyada mushkil concept hai, khaas taur par OpenAI Assistants API mein relevant.

- **Definition ():**

  `StopAtTools` aam taur par `stop_on_first_tool` jaisa boolean flag nahin hota. Iske bajaye, yeh agent ki execution mein aik **state ya strategic point** ko bayan karta hai jahan woh khaas taur par **LLM ne aik single generation step ke andar call karne ka faisla kiye gaye *tamam* tools ko execute karne ki ijazat dene ke liye rukta hai**. Agent tamam zaroori tool calls generate karega aur phir aik state (maslan, Assistants API mein `requires_action`) mein tabdeel ho jayega jahan aapki application unhein execute kar sake.

- **Explanation ():**

  Is scenario mein, LLM ka sochne ka tareeqa yeh ho sakta hai: "X ka jawab dene ke liye, mujhe data hasil karne ke liye `tool_A` ko call karna hoga, aur phir us data ko process karne ke liye `tool_B` ko." **`StopAtTools`** ke sath (aik structured workflow jaise Assistants API ke hisse ke taur par), LLM apne `required_action` block mein `tool_A` aur `tool_B` dono calls ko output karega. Agent tab tak nahin rukega jab tak yeh tamam plan shuda tool calls pesh na kar diye jayen. Aapki application phir `tool_A` aur `tool_B` (ya un sabhi) calls ki list receive karti hai, un sabhi ko execute karti hai, aur unke outputs ko agent ko wapas submit karti hai. Agent phir in tamam outputs ko receive karne ke *baad* processing dobara shuru karta hai.

- **Use Cases ():**

  Aap **`StopAtTools`** approach ko tab istemal karenge ya use tarjeeh denge jab:

  - Aap chahte hain ke LLM aik hi turn mein **kayi tool calls ko aik sath plan** kare, jis se mutaliqa tools ko parallel ya chained tareeqay se execute kiya ja sake. Yeh aksar mutalaq Assistant frameworks jaise OpenAI Assistants API ka **default behavior** hota hai jab woh `requires_action` state mein dakhil hota hai.
  - Tools ka **sequence bahut tightly coupled** hai aur aik tool ka output agle ko call karne ke *faisle* ko lazmi nahin badalta, balkay sirf uske liye input faraham karta hai (maslan, data fetch karna, phir us data ka تجزیہ karna).
  - Aap **kam granular rukawat** aur aik ziyada sayyal (fluid), khudmukhtar agent execution chahte hain. Agent aik step ke liye apna poora tool-calling plan banata hai, aur aap us batch ko execute karte hain.
  - Aap OpenAI Assistants jaisi API ke sath kaam kar rahe hain, jahan `Run` object qudrati taur par `requires_action` state mein dakhil hota hai aur aapko execute karne ke liye `tool_calls` ki list faraham karta hai.
  - **Efficiency aik ahmiyat rakhti hai**, kyunki yeh aapki application aur LLM ke darmiyan API round-trips ki tadaad ko kam karta hai. LLM ke tool planning ke liye aik trip, aur phir tamam tool outputs submit karne ke liye aik trip.

Kab Kisko Chunein ()

`StopAtTools` (ya APIs jaisi Assistants mein iska qudrati behavior) aur **`stop_on_first_tool`** (custom loops ya saade frameworks mein aik ziyada wazeh control) ke darmiyan intikhab **orchestration control** ki aapki mutlooba satah aur **aapke tool interactions ki pechidgi** par munhasir hai:

- **`stop_on_first_tool` ko chunein agar:**
  - Aapko har **individual tool call** ke baad uski janch-partaal, tarmeem, ya user ki manzoori ki zarurat hai.
  - Tool calls ke **significant side effects** hain jo LLM ke baad ke reasoning path ko badal dete hain.
  - Aap aik bahut **interactive ya guided** agent experience bana rahe hain.
  - Aap costs ya rate limits ko **har tool call ki bunyad par** manage karna chahte hain.
- **`StopAtTools` ko chunein (ya use by default istemal karen) agar:**
  - Aap LLM par bharosa karte hain ke woh **aik hi go mein mutaliqa tool calls ki series plan** karega.
  - Tools aam taur par **mustaqil** hain ya aik qabil-e-peshan-goi chain banate hain, aur aap unhein baghair rukawat ke **batch mein execute** karna chahte hain.
  - Aap **kam API round-trips** aur aik ziyada sayyal (fluid), khudmukhtar agent execution chahte hain.
  - Aap aik aisi API (jaise OpenAI Assistants) ke sath kaam kar rahe hain jo tamam generate kiye gaye tool calls ko aik sath `requires_action` state mein pesh karne ke liye design ki gayi hai.

Mukhtasar mein, **`stop_on_first_tool`** aapko fine-grained, sequential control deta hai, jabke **`StopAtTools`** (tamam generate kiye gaye tools ke liye rukne ka concept) LLM ko aik diye gaye step ke liye aik ziyada mukammal plan banane ki ijazat deta hai, jis se ziyada kargar batch execution hoti hai.

Code Example: Dono Behaviors Ki Simulation

Chunkay **`StopAtTools`** OpenAI Assistants API ke `requires_action` state jaisi cheezon ke andarunee ravaiye ke baray mein ziyada hai, aur `stop_on_first_tool` aik wazeh flag hai jo aap aik custom agent loop mein set karte hain, to aaiye dekhte hain ke aap aik custom agent ko kis tarah implement karenge jo in donon ravaiyon ke darmiyan switch kar sake.

```
import time
import json
from typing import List, Dict, Any, Tuple, Callable

# --- Mock Tool Definitions ---
def get_user_profile(user_id: str) -> Dict[str, Any]:
    """User ki profile details hasil karta hai."""
    print(f"[Tool]: User_id: {user_id} ke liye profile hasil ki ja rahi hai...")
    if user_id == "user123":
        return {"name": "Alice", "email": "alice@example.com", "phone_verified": True}
    return {"error": "User nahin mila"}

def send_email(to_email: str, subject: str, body: str) -> Dict[str, Any]:
    """Aik email bhejta hai."""
    print(f"[Tool]: {to_email} ko '{subject}' subject ke sath email bheja ja raha hai...")
    if "@" in to_email:
        return {"status": "email_sent", "recipient": to_email}
    return {"error": "Invalid email format"}

# Tool names ko unke asal functions se map karna
available_tools = {
    "get_user_profile": get_user_profile,
    "send_email": send_email,
}

# --- Mock LLM (Simplified Decision Logic) ---
class MockLLM:
    def __init__(self):
        self.conversation_history = [] # Guftagu ke context ko simulate karta hai
```

```
    def generate_tool_calls(self, prompt: str, stop_on_first_tool: bool = False) ->
Tuple[List[Dict[str, Any]], bool]:
        """
        Prompt aur control flag ki bunyad par LLM ke tool calls generate karne ko simulate karta
hai.
        Returns (tool_calls ki list, llm_ka_tools_ke_liye_rukne_ya_jari_rakhne_ka_faisla).
        """
        self.conversation_history.append(prompt)
        tool_calls = []
        should_continue_generating_tools = True

        if "get user profile for user123 and email them" in prompt.lower():
            # LLM faisla karta hai ke usay do tools chahiye
            tool_calls.append({
                "id": "tool_call_001",
                "function": {"name": "get_user_profile", "arguments": json.dumps({"user_id":
"user123"})}
            })
            if stop_on_first_tool:
                # Agar stop_on_first_tool True hai, to LLM pehle tool plan ke baad ruk jata hai
                print("  [MockLLM]: stop_on_first_tool=True, pehle tool call ke baad ruk raha
hai.")
                should_continue_generating_tools = False # External orchestrator ko rukne ka
ishara dena
            else:
                # Warna, yeh isi turn mein agle tool call ko plan karna jari rakhta hai
                tool_calls.append({
                    "id": "tool_call_002",
                    "function": {"name": "send_email", "arguments": json.dumps({"to_email":
"user@example.com", "subject": "Hello", "body": "Profile details attached."})}
                })
                print("  [MockLLM]: stop_on_first_tool=False, aik sath kayi tools plan kar raha
hai.")
                should_continue_generating_tools = False # Is turn ke liye mazeed tools plan nahin
kiye gaye

        elif "fetch user profile for user999" in prompt.lower():
             tool_calls.append({
                "id": "tool_call_003",
                "function": {"name": "get_user_profile", "arguments": json.dumps({"user_id":
"user999"})}
            })
             should_continue_generating_tools = False # Sirf aik tool chahiye

        else:
            return [], True # Koi tools nahin, LLM normal text generation jari rakhta hai

        return tool_calls, should_continue_generating_tools

    def generate_text_response(self, prompt: str, tool_outputs: List[Dict[str, Any]]) -> str:
        """LLM ke final text response generate karne ko simulate karta hai."""
        response = "Mai aapki request samajh nahin paya."
        if tool_outputs:
            output = tool_outputs[0] # Saadi misal ke liye aik output suppose kiya
            if output.get("name") == "get_user_profile":
                data = json.loads(output["output"])
                if data.get("error"):
                    response = f"User profile hasil nahin kar saka: {data['error']}. "
                else:
                    response = f"User profile hasil ho gayi: Naam {data['name']} hai, Email
{data['email']} hai. "
            elif output.get("name") == "send_email":
                 data = json.loads(output["output"])
                 if data.get("error"):
```

```
                    response += f"Email bhejne mein nakaam raha: {data['error']}. "
                else:
                    response += f"Email {data['recipient']} ko bhej di gayi hai. "

        # Prompt ki bunyad par saadi continuation
        if "email them" in prompt.lower() and not tool_outputs:
             response += " Email bhejne ke liye mujhe user ki email maloom honi chahiye."

        return response + " Aur kis tarah madad kar sakta hoon?"

# --- Agent Orchestrator (Runner ko Simulate Karna) ---
def run_agent_flow(user_query: str, control_flag: str):
    llm = MockLLM()
    print(f"\n--- Control flag ke sath chal raha hai: {control_flag} ---")

    current_prompt = user_query

    # Assistants API ke Run concept ko simulate karta hai
    run_status = "in_progress"

    while run_status != "completed" and run_status != "failed":
        print(f"\n[Orchestrator]: Maujooda Run Status: {run_status}")

        if control_flag == "stop_on_first_tool":
            # LLM tool calls generate karta hai, lekin hum use pehle ke baad rukne ko kehte hain.
            tool_calls_from_llm, llm_wants_to_continue = llm.generate_tool_calls(current_prompt,
stop_on_first_tool=True)
        elif control_flag == "stop_at_tools":
            # LLM is turn ke liye plan kiye gaye tamam tool calls generate karta hai.
            tool_calls_from_llm, llm_wants_to_continue = llm.generate_tool_calls(current_prompt,
stop_on_first_tool=False)
        else:
            print("[Orchestrator]: Invalid control flag.")
            run_status = "failed"
            break

        if tool_calls_from_llm:
            print(f"[Orchestrator]: LLM ne {len(tool_calls_from_llm)} tool calls ki darkhwast
ki.")
            run_status = "requires_action" # OpenAI Assistants API state ke mutabiq

            executed_tool_outputs = []
            for tool_call in tool_calls_from_llm:
                print(f"  [Orchestrator]: Tool '{tool_call['function']['name']}' execute kiya ja
raha hai...")
                func = available_tools.get(tool_call['function']['name'])
                if func:
                    try:
                        args = json.loads(tool_call['function']['arguments'])
                        output = func(**args)
                        executed_tool_outputs.append({
                            "tool_call_id": tool_call["id"],
                            "name": tool_call['function']['name'], # MockLLM ko output process
karne ke liye
                            "output": json.dumps(output)
                        })
                    except Exception as e:
                        print(f"    [Orchestrator]: Tool execution mein ghalti hui: {e}")
                        executed_tool_outputs.append({
                            "tool_call_id": tool_call["id"],
                            "name": tool_call['function']['name'],
                            "output": json.dumps({"error": str(e), "status": "failed"})
                        })
                else:
```

```
                    print(f"    [Orchestrator]: Tool '{tool_call['function']['name']}' nahin
mila.")
                    executed_tool_outputs.append({
                        "tool_call_id": tool_call["id"],
                        "name": tool_call['function']['name'],
                        "output": json.dumps({"error": "Tool nahin mila", "status": "failed"})
                    })

            # Tools execute karne ke baad, outputs ko wapas LLM context mein feed karna
            # Asal OpenAI Assistants mein, aap submit_tool_outputs karte aur run jari rehta
            # Yahan, hum LLM ke outputs ki bunyad par response generate karne ko simulate karte
hain
            current_prompt = "Tool outputs receive hue: " + json.dumps(executed_tool_outputs)
            final_response = llm.generate_text_response(user_query, executed_tool_outputs) # LLM
ko original prompt + naye tool outputs milte hain
            print(f"\n[Assistant ka Final Response is segment ke liye]: {final_response}")
            run_status = "completed" # Is saadi misal ke liye, hum tool execution aur response ke
baad complete karte hain

        else: # LLM ne koi tool calls nahin kiye
            print("[Orchestrator]: LLM ne koi tools ki darkhwast nahin ki.")
            final_response = llm.generate_text_response(user_query, [])
            print(f"\n[Assistant ka Final Response]: {final_response}")
            run_status = "completed"


# --- Demonstrations Chalana ---
if __name__ == "__main__":
    # Scenario 1: stop_on_first_tool ka istemal (ziyada granular control)
    # Agent get_user_profile ko execute karega, phir ruk jayega.
    # Yeh usi turn mein send_email ko plan bhi nahin karega.
    run_agent_flow("Please get user profile for user123 and email them a welcome message.",
"stop_on_first_tool")

    print("\n" + "="*100 + "\n")

    # Scenario 2: StopAtTools ka istemal (plan kiye gaye tools ki batch execution)
    # Agent get_user_profile aur send_email dono ko usi turn mein plan karega,
    # phir rukega, dono ko execute karne ki ijazat dega, aur phir jari rakhega.
    run_agent_flow("Please get user profile for user123 and email them a welcome message.",
"stop_at_tools")

    print("\n" + "="*100 + "\n")

    # Scenario 3: Single tool call jahan dono aik jaisa behave karenge
    run_agent_flow("What is the profile for user999?", "stop_on_first_tool")
    print("\n" + "="*100 + "\n")
    run_agent_flow("What is the profile for user999?", "stop_at_tools")
```

Code Example Ki Wazahat ():

1. **`MockTool Definitions`**: Yeh asal tools ki simulation hain (`get_user_profile` aur `send_email`). Yeh asal APIs ko call karne ke bajaye simple dictionary returns karte hain.
2. **`MockLLM` Class**:
    - Yeh aik **simulated Language Model (LLM)** hai. Yeh asal mein LLM nahin hai, balkay LLM ke tool-calling behavior ko mimic karta hai.
    - **`generate_tool_calls(prompt, stop_on_first_tool=False)`**: Yahi woh **aham method** hai jahan dono flags (`stop_on_first_tool` aur `StopAtTools` ka behavior) ka farq dekha jata hai.

        - Agar `stop_on_first_tool=True` hai, aur LLM ko aik se ziyada tools chahiye (jaise "get user profile and email them" ke liye `get_user_profile` aur `send_email`), to yeh sirf **pehla tool call generate karega** aur `should_continue_generating_tools = False` (yani bahar wale orchestrator ko "abhi ruko") set kar dega.
        - Agar `stop_on_first_tool=False` hai (jo **`StopAtTools`** ke default behavior ko simulate karta hai), aur LLM ko aik se ziyada tools chahiye, to yeh **tamam zaroori tool calls generate karega aik hi baar mein**.
    - **`generate_text_response`**: Yeh LLM ke final text response generate karne ko simulate karta hai jab usay tool outputs mil chukay hotay hain.
3. **`run_agent_flow(user_query, control_flag)` (Agent Orchestrator Simulation):**
    - Yeh function aik **Runner** (jaise LangChain mein hota hai ya OpenAI Assistants API ke under-the-hood logic) jaisa kaam karta hai. Yeh LLM ko invoke karta hai aur tool execution ko manage karta hai.
    - `control_flag` (`"stop_on_first_tool"` ya `"stop_at_tools"`) LLM ke `generate_tool_calls` method ko pas kiya jata hai.
    - **Main `while` loop**: Yeh `run_status` ko poll karta hai, jo asal OpenAI Assistants API ke `Run` object status (`in_progress`, `requires_action`, `completed`) ko emulate karta hai.
    - **Tool Execution**: Agar LLM tool calls request karta hai, to `orchestrator` un tools ko `available_tools` dictionary se dhund kar execute karta hai. Asal mein, yeh APIs call kar sakta hai.

Har Scenario Ki Tafseel ():

- **Scenario 1: `stop_on_first_tool` ka istemal (for "get user profile for user123 and email them")**
    - LLM ko do tools (`get_user_profile`, `send_email`) chahiye.
    - Lekin `stop_on_first_tool=True` ki wajah se, `MockLLM` sirf `get_user_profile` ka call generate karta hai.
    - `Orchestrator` sirf `get_user_profile` ko execute karta hai.
    - Is simplified example mein, `Run completed` ho jata hai, aur LLM jawab deta hai, lekin woh email send nahin karta kyunki `send_email` ka plan abhi tak bana hi nahin. Aapko next turn mein dobara prompt karna hoga ya mazeed orchestration karni hogi.
    - **Natija:** LLM **sirf pehla tool** (`get_user_profile`) plan karta hai aur ruk jata hai. Aapke paas har **individual tool call** ko rokne aur use manage karne ka mauqa hota hai.
- **Scenario 2: `stop_at_tools` ka istemal (for "get user profile for user123 and email them")**
    - LLM ko do tools chahiye.
    - `stop_on_first_tool=False` (yani **`StopAtTools`** ka behavior) ki wajah se, `MockLLM` **dono** `get_user_profile` aur `send_email` calls ko **aik hi baar mein generate karta hai**.
    - `Orchestrator` phir dono tools ko execute karta hai.
    - `Run completed` ho jata hai, aur LLM jawab deta hai jo dono tools ke outputs ko cover karta hai.
    - **Natija:** LLM **aik hi baar mein tamam zaroori tool calls** (`get_user_profile` aur `send_email`) ki planning karta hai. Aap unhein aik **batch mein execute** kar sakte hain, jis se API round-trips kam ho jati hain.
- **Scenario 3: Single Tool Call (`user999` profile)**
    - Is scenario mein, user sirf aik tool call request karta hai. Chahe `stop_on_first_tool` `True` ho ya `False`, LLM sirf aik tool call generate karega. Isliye, in dono control flags ka behavior is case mein **aik jaisa** hoga. Yeh batata hai ke asal farq tab wazeh hota hai jab LLM ko aik hi turn mein kayi tools call karne ki zarurat ho.

Summary ()

`stop_on_first_tool` aapko **har fard-e-wahid tool call par qabza (control)** deta hai. Aap use tab istemal karte hain jab aapko har tool execution ke baad dakhal-andazi karne, side effects ko manage karne, ya debugging karne ki zarurat ho. Yeh ziyada granular aur sequential control faraham karta hai.

`StopAtTools` (jo aksar OpenAI Assistants API ke `requires_action` jaisi states ka default behavior hai) is baat ko bayan karta hai ke LLM **aik hi baar mein tamam mutaliqa tools ko plan karega**. Aap ise tab istemal karte hain jab aap LLM par bharosa karte hain ke woh related tools ko sahih tareeqay se plan karega aur aap un tools ko batch mein execute kar ke efficiency hasil karna chahte hain. Yeh kam granular lekin ziyada sayyal (fluid) execution faraham karta hai.

Intikhab aapke agent ke workflow ki complexity aur aapko kitni dafa agent ki tool-calling process mein dakhal-andazi karni hai, is par munhasir hai. Kya aap is simulation ko kisi aur scenario ke sath mazeed explore karna chahenge?

26. What is the default value for tool_use_behavior in an Agent ?

Agent Mein `tool_use_behavior` Ki Default Value Kya Hai?

Definition ()

OpenAI Agents SDK (ya is se milte-julte agent orchestration frameworks) mein, **`tool_use_behavior`** aik configuration setting hai jo is baat ko control karti hai ke Agent (yani underlying LLM) tools ko kis tarah istemal karta hai aur aapki application ke sath interact karta hai jab tool calls ki zarurat hoti hai. Is setting ki **default value** yeh tay karti hai ke agar aap khud se koi value specify nahin karte to Agent tools ko kis tareeqay se handle karega.

Explanation ()

`tool_use_behavior` aik crucial parameter hai jo Agent ke tool-calling workflow ko define karta hai. Yeh LLM aur aapke code ke darmiyan coordination ka tareeqa batata hai. Iski default value Agent framework par munhasir ho sakti hai, lekin aam taur par iska maqsad LLM ko tools istemal karne ki mukammal azadi dena hota hai, jab tak ke kisi insani dakhal-andazi ki zarurat na pade.

OpenAI Assistants API ke Context Mein Default Behavior:

OpenAI Assistants API ke mutabiq, jab aap aik **Run** create karte hain aur usmein **`tool_use_behavior`** ko explicitly specify nahin karte, to **default behavior** yeh hota hai ke Assistant (LLM) **automatically tools ko execute karne ki koshish karega agar usay zarurat ho**. Agar tools ka output chahiye to yeh `requires_action` state mein rukay ga, jahan aapke code ko tools execute karke unke results wapas submit karne honge. Asal mein, LLM ko khud se tools ko run karne ki "permission" hoti hai.

Tafseelan:

1. **Autonomous Tool Execution (Khud Mukhtar Tool Execution):** Default taur par, Agent ka maqsad yeh hota hai ke woh apne task ko mukammal karne ke liye zaroorat ke mutabiq tools ko khudmukhtari se istemal kare. Yeh LLM ko `tool_code` (Python interpreter, ya any other code interpreter for example) aur `functions` (custom tools) ko call karne ki ijazat deta hai.
2. **`requires_action` State:** Jab LLM koi tool call karta hai (maslan, aik `function` tool ya `code_interpreter` tool), to Assistant ka **Run `requires_action`** state mein chala jata hai. Yeh woh default "pause" point hai. Aapki application ko is state ko detect karna hoga, LLM ke plan kiye gaye tools ko execute karna hoga, aur phir un tool outputs ko Assistant ko wapas **submit** karna hoga. Iske baad, Run dobara `in_progress` state mein chala jata hai aur LLM mazeed processing karta hai.
3. **No Explicit Approval Needed for Each Tool Call:** Default behavior mein, LLM ko har tool call ke liye aapki explicit "manzoori" ki zarurat nahin hoti. Woh apna plan banata hai aur `requires_action` mein chala jata hai, aapke code se umeed karta hai ke woh tools execute karke output provide karega.

Kisi Aur Framework (Maslan LangChain) Mein Default Values:

Agar aap koi dusra agent framework (jaise LangChain) use kar rahe hain jo OpenAI Assistants API ke upar bana hai, to wahan `tool_use_behavior` jaisa parameter ho sakta hai jiska default behavior slightly mukhtalif ho, lekin buniyadi tor par woh isi concept ko emulate karega:

- **Most Autonomous Option:** Default value aam taur par sabse ziyada khudmukhtar option hoti hai jahan agent apne task ko pura karne ke liye tools ko khud se manage karta hai, aur sirf us waqt rukta hai jab insani dakhal-andazi (jaise tool output ki zarurat) zaroori ho.
- Misal ke taur par, LangChain ke kuch agents `AgentExecutor` mein `handle_parsing_errors` ya `return_intermediate_steps` jaisi settings hoti hain jo tool behavior se related ho sakti hain, lekin `tool_use_behavior` directly OpenAI Assistants API se aaya hai.

Default Value Ka Khulasa:

Asal mein, OpenAI Assistants API mein **`tool_use_behavior`** (jab explicitly set na kiya jaye) ka default ye hota hai ke Agent "**autonomously decides to call tools and transitions to `requires_action` state for your application to execute them.**" Is mein koi explicit **`tool_use_behavior`** parameter value nahi di jati, balkay ye API ka andaruni ravaiya hai.

Agar API mein koi enum value hoti to woh shayad "auto" ya "continue_on_tool_call" jaisi hoti.

Example Code ()

OpenAI Python SDK mein, aap `tool_use_behavior` ko directly **`client.beta.threads.runs.create()`** mein specify kar sakte hain. Jab aap ise nahin dete, to default behavior lagoo hota hai. Aaiye dekhte hain iska matlab kya hai code mein:

```
from openai import OpenAI
import os
import time
import json
from dotenv import load_dotenv

load_dotenv()

# Initialize OpenAI client
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

# --- Mock Tool ---
def get_current_weather(location: str, unit: str = "fahrenheit") -> str:
    """Gets the current weather in a given location."""
    print(f"\n[Tool Execution]: Fetching weather for {location} ({unit})...")
    if "karachi" in location.lower():
        return json.dumps({"location": location, "temperature": "32", "unit": unit, "forecast":
"sunny"})
    elif "london" in location.lower():
        return json.dumps({"location": location, "temperature": "18", "unit": unit, "forecast":
"cloudy"})
    else:
        return json.dumps({"location": location, "temperature": "N/A", "error": "Location not
found"})

# Tool mapping for execution
available_functions = {
    "get_current_weather": get_current_weather
}

# --- Main Assistant Flow ---
async def demonstrate_default_tool_behavior():
    # 1. Assistant create ya retrieve karein
    try:
        # Apne Assistant ID ko yahan dalen
```

```
        my_assistant = client.beta.assistants.retrieve("asst_your_weather_assistant_id")
        print(f"Existing Assistant istemal ho raha hai: {my_assistant.id}")
    except Exception:
        print("Naya Assistant banaya ja raha hai...")
        my_assistant = client.beta.assistants.create(
            name="Weather Checker Assistant",
            instructions="You are a helpful assistant that can provide current weather information
using tools.",
            model="gpt-4o", # Ya koi aur tool-enabled model
            tools=[
                {"type": "function", "function": {
                    "name": "get_current_weather",
                    "description": "Get the current weather in a given location",
                    "parameters": {
                        "type": "object",
                        "properties": {
                            "location": {"type": "string", "description": "The city and state,
e.g. San Francisco, CA"},
                            "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]}
                        },
                        "required": ["location"]
                    }
                }}
            ]
        )
        print(f"Naya Assistant ban gaya ID ke sath: {my_assistant.id}")

    # 2. Thread create karein
    print("\nNaya Thread banaya ja raha hai...")
    thread = client.beta.threads.create()
    print(f"Naya Thread ban gaya ID ke sath: {thread.id}")

    # 3. User message add karein
    user_message = "What's the weather like in Karachi?"
    print(f"\nUser: {user_message}")
    client.beta.threads.messages.create(
        thread_id=thread.id,
        role="user",
        content=user_message,
    )

    # 4. Run create karein WITHOUT specifying tool_use_behavior
    # Yeh is baat ka saboot hai ke hum default behavior istemal kar rahe hain.
    print("\nRun banaya ja raha hai (default tool_use_behavior ke sath)...")
    current_run = client.beta.threads.runs.create(
        thread_id=thread.id,
        assistant_id=my_assistant.id,
    )
    print(f"Run ban gaya ID ke sath: {current_run.id}, ibtedai status: {current_run.status}")

    # 5. Run status ko poll karein
    print("\nRun status check kiya ja raha hai...")
    while current_run.status == "queued" or current_run.status == "in_progress":
        time.sleep(0.5)
        current_run = client.beta.threads.runs.retrieve(thread_id=thread.id,
run_id=current_run.id)
        print(f"Run status: {current_run.status}")

    # 6. Default behavior mein, hum expects karte hain ke yeh "requires_action" mein jaye ga
    if current_run.status == "requires_action":
        print("\nRun requires_action state mein hai: Assistant tool call karna chahta hai.")
        tool_outputs_to_submit = []
        for tool_call in current_run.required_action.submit_tool_outputs.tool_calls:
            print(f"  Tool Call ID: {tool_call.id}")
```

```
            print(f"  Function Name: {tool_call.function.name}")
            print(f"  Arguments: {tool_call.function.arguments}")

            # Asal tool function ko execute karein
            function_name = tool_call.function.name
            function_args = json.loads(tool_call.function.arguments)

            output = ""
            if function_name in available_functions:
                output = available_functions[function_name](**function_args)
            else:
                output = json.dumps({"error": f"Function {function_name} not found"})

            tool_outputs_to_submit.append({
                "tool_call_id": tool_call.id,
                "output": output,
            })
            print(f"  Tool Output: {output}")

        # Tool outputs ko Assistant ko wapas submit karein
        print("\nTool outputs submit kiye ja rahe hain...")
        current_run = client.beta.threads.runs.submit_tool_outputs(
            thread_id=thread.id,
            run_id=current_run.id,
            tool_outputs=tool_outputs_to_submit,
        )
        print(f"Run dobara shuru ho gaya status ke sath: {current_run.status}")

        # Submit karne ke baad dobara poll karein completion tak
        while current_run.status == "queued" or current_run.status == "in_progress":
            time.sleep(0.5)
            current_run = client.beta.threads.runs.retrieve(thread_id=thread.id,
run_id=current_run.id)
            print(f"Run status: {current_run.status}")

    if current_run.status == "completed":
        print("\nRun mukammal ho gaya. Assistant ka final message hasil kiya ja raha hai...")
        messages = client.beta.threads.messages.list(thread_id=thread.id, order="desc", limit=1)
        if messages.data and messages.data[0].role == "assistant" and
messages.data[0].content[0].type == "text":
            print(f"\nAssistant: {messages.data[0].content[0].text.value}")
        else:
            print("\nAssistant ne text response nahin diya ya response ghair-mutawaqqo format mein
hai.")
    elif current_run.status == "failed":
        print(f"\nRun failed. Last error: {current_run.last_error}")
    else:
        print(f"\nRun ghair-mutawaqqo status par khatam hua: {current_run.status}")

# Ensure you have your Assistant ID set or remove the try-except to create a new one every time.
if __name__ == "__main__":
    import asyncio
    asyncio.run(demonstrate_default_tool_behavior())
```

Code Example Ki Wazahat ():

1. **Assistant Aur Thread Banana:** Hum aik Assistant (jiske paas `get_current_weather` tool hai) aur aik naya conversation thread banate hain.

2. **`client.beta.threads.runs.create(...)` Call:**
    - **Aham Nuqta:** Is function call mein, humne `tool_use_behavior` parameter ko **specify nahin kiya** hai. Yahi woh jagah hai jahan **default behavior** lagoo hota hai.
    - Iska matlab hai ke hum **OpenAI Assistants API ke built-in, khudmukhtar tool-calling mechanism** par inhesar kar rahe hain.
3. **Polling For `requires_action`:**
    - Jab Run shuru hota hai (`in_progress` status), Assistant LLM weather maloomat hasil karne ke liye `get_current_weather` tool ko call karne ka faisla karega.
    - Jab LLM tool call generate kar chuka hoga, to **Run ka status `requires_action` mein badal jayega**. Yeh is baat ka saboot hai ke default `tool_use_behavior` ne kaam kiya hai, aur LLM ne khud se tool call karne ka faisla kiya hai aur ab woh tool output ka intizar kar raha hai.
    - Hamara code is `requires_action` state ko detect karta hai.
4. **Tool Execution And Output Submission:**
    - Code `current_run.required_action.submit_tool_outputs.tool_calls` se tool call ki tafseelat nikalta hai.
    - Phir, yeh hamare `available_functions` dictionary se asal `get_current_weather` function ko execute karta hai.
    - Tool ka output hasil karne ke baad, yeh `client.beta.threads.runs.submit_tool_outputs()` function ko call karta hai, jo outputs ko Assistant ko wapas bhej deta hai. Isse Run dobara `in_progress` state mein chala jata hai.
5. **Final Response:** Jab Run `completed` ho jata hai, to hum Assistant ka final jawab retrieve aur print karte hain.

Is misal se wazeh hota hai ke **`tool_use_behavior` ki default value Assistant ko tools call karne ki ijazat deti hai, aur jab woh tools call karta hai to Run `requires_action` state mein chala jata hai, jahan aapke code ko un tools ko execute karna hota hai.**

Summary ()

Agent mein **`tool_use_behavior`** ki default value (khaas taur par OpenAI Assistants API mein) yeh hai ke **Agent tools ko istemal karne ka khudmukhtarana faisla karega aur jab use tool outputs ki zarurat hogi to `requires_action` state mein chala jayega.**

Iska matlab hai:

- LLM ko apne task ko pura karne ke liye `code_interpreter` ya `function` tools ko call karne ki **mukammal azadi** hoti hai.
- Jab Agent koi tool call karta hai, to `Run` ka status **`requires_action`** ho jata hai.
- Aapki application ko is `requires_action` state ko monitor karna, LLM ke plan kiye gaye tools ko **execute** karna, aur phir un tools ke outputs ko Assistant ko **submit** karna hota hai taake Run dobara jari reh sake.
- Is default behavior mein, har tool call ke liye **explicit user approval ki zarurat nahin** hoti; Agent khud faisla karta hai ke kab aur kaun sa tool call karna hai, aur aapka code uske baad ke execution ka zimmedar hota hai.

---

27. What's the key difference b/w run_llm_again and stop_on_first_tool in terms of performance ?

`run_llm_again` aur `stop_on_first_tool` Ke Darmiyan Performance Ka Bunyadi Farq Kya Hai?

Agent frameworks (jaise LangChain ya custom agent orchestrators) mein, `run_llm_again` aur `stop_on_first_tool` do mukhtalif tareeqe hain jo agent ke control flow aur LLM (Large Language Model) ke saath interaction ko mutasir karte hain. Inka bunyadi farq performance par seedha asar dalta hai, khaas taur par **API calls ki tadaad aur overall latency** (deri) ke lehaz se.

`stop_on_first_tool` Kya Hai?

`stop_on_first_tool` aik configuration setting ya flag hai.

- **Definition ():**

  Jab ise `True` par set kiya jata hai, to yeh agent ko hidayat deta hai ke woh **LLM ke pehle tool call ka faisla karne ke foran baad apni execution rokay**. Yani, LLM apni soch-vichar mein agar aik se ziyada tools call karna chahe bhi, agent sirf pehle tool call ki tafseelat (specifications) ko retrieve karega aur ruk jayega.

- **Explanation ():**
  - **Control Flow:** Yeh bahut **granular control** faraham karta hai. Har tool call ke baad, agent ruk jata hai. Aapki application ko tool ko execute karna hoga, uske output ko hasil karna hoga, aur phir nakiye agent ko dobara LLM ke paas bhejna hoga.
  - **LLM Interactions:** Is mein LLM ke saath **ziyada API round-trips** shamil hotay hain. Har tool call ke liye aik alag API call ya interaction cycle hoti hai.
  - **Performance Impact:**
    - **Latency (Deri):** Ziyada hoti hai. Har tool call ke liye network latency, API processing time, aur LLM ke dobara sochne ka waqt shamil hota hai.
    - **Cost:** Har round-trip mein LLM ko dobara invoke karna padta hai, jiska matlab ziyada token consumption (agar previous context ko har baar bheja jaye) aur is tarah ziyada costs ho sakti hain.
    - **Use Case:** Yeh debugging, user approval (har tool action se pehle), ya complex multi-step processes ke liye behtar hai jahan har tool ke output par LLM ka agla faisla mukhtalif ho sakta hai.

`run_llm_again` Kya Hai?

`run_llm_again` directly OpenAI Assistants API ka parameter nahin hai, balkay yeh LangChain jaise frameworks mein aik **design pattern** ya internal mechanism ki taraf ishara karta hai, jahan LLM ko `stop_on_first_tool` ke bar-aks **multiple tool calls ko aik hi baar mein plan aur execute karne** (ya `required_action` state mein jama karne) aur phir un tamam outputs ke baad LLM ko dobara chalane ki ijazat di jati hai. Iska maqsad aam taur par **LLM ko us waqt tak chalate rehna hai jab tak woh khud faisla na kare ke use mazeed tools ki zarurat nahin ya woh final jawab de sakta hai.**

- **Definition ():**

  Yeh aik agent ke us ravaiye ko bayan karta hai jahan LLM ko **ek hi turn mein multiple tools ko call karne ka mauqa diya jata hai** (ya unhein plan karne ka), aur Agent tab tak LLM ko dobara invoke karta rehta hai jab tak ke LLM koi final text response na de ya saaf tor par bataye ke use mazeed tools ki zarurat nahin.

- **Explanation ():**
  - **Control Flow:** Yeh **autonomous (khud-mukhtar)** control faraham karta hai. LLM ko ijazat hoti hai ke woh aik hi go mein kayi tools ko chain (aik ke baad aik) ya parallel (aik sath) call kare. Agent un sab tool calls ko ek hi bar mein execute karta hai aur phir un sab ke outputs ko LLM ko wapas bhej deta hai.
  - **LLM Interactions:** Is mein LLM ke saath **kam API round-trips** shamil hotay hain. Agar LLM ko 3 tools ki zarurat hai, to woh un 3 calls ko aik hi baar mein plan karega, aapki application unko execute karegi, aur phir un 3 outputs ko aik hi baar mein LLM ko wapas bheja jayega. Yeh aik "batch processing" jaisa hai.
  - **Performance Impact:**
    - **Latency (Deri):** Kam hoti hai. Multiple tool calls ke liye sirf aik initial LLM invocation aur aik final LLM invocation (outputs submit karne ke baad) hoti hai. Darmiyan mein tool execution aapke local environment mein hoti hai.
    - **Cost:** Aam taur par kam hoti hai kyunki LLM ko dobara invoke karne ki tadaad kam ho jati hai. Context management bhi behtar ho sakti hai.
    - **Use Case:** Complex tasks ke liye behtar hai jahan LLM ko multiple related actions ki planning karne ki zarurat hoti hai (maslan, "Ek user ki profile fetch karo, uske orders dekho, aur phir usay recent order status ka email bhejo"). Yeh production environments mein ziada pasand kiya jata hai jahan performance aur efficiency aham hai.

Key Difference in Performance

Khususiyat (Feature)	`stop_on_first_tool`	`run_llm_again` **(Implicit behavior)**
API Round-Trips	**Ziyada** (har tool call ke liye aik trip)	**Kam** (LLM ki planning ke baad aik ya do trip, phir tool execution ke baad wapis)
Latency (Deri)	**Ziyada** (har round-trip par network/LLM delay)	**Kam** (fewer network/LLM delays)
Cost Efficiency	**Kam** (ziyada LLM invocations)	**Ziyada** (kam LLM invocations)
Control Granularity	**Bahut Ziyada** (har tool par control)	**Kam** (LLM ki planning par bharosa)
Best For	Debugging, User Confirmation, Critical Side Effects, Step-by-step guidance	Autonomous multi-step tasks, Production, Efficiency, Complex workflows with internal chaining

Export to Sheets

Example Code (Simulating Both Behaviors)

Is misal mein, hum aik agent orchestrator banayenge jo `stop_on_first_tool` ko explicitly use kar sakega aur `run_llm_again` ke behavior ko simulate karega (yaani LLM ko tab tak chalne dega jab tak woh final jawab na de).

```
import time
import json
from typing import List, Dict, Any, Tuple, Callable

# --- Mock Tool Definitions ---
def get_stock_price(symbol: str) -> Dict[str, Any]:
    """Given a stock symbol, returns its current price."""
    print(f"\n[Tool]: Fetching stock price for {symbol}...")
    time.sleep(0.1) # Simulate network delay
    if symbol.upper() == "GOOG":
        return {"symbol": "GOOG", "price": 175.50, "currency": "USD"}
    elif symbol.upper() == "MSFT":
        return {"symbol": "MSFT", "price": 420.10, "currency": "USD"}
```

```
    return {"symbol": symbol, "error": "Symbol not found"}

def send_notification(user_id: str, message: str) -> Dict[str, Any]:
    """Sends a notification to a specific user."""
    print(f"[Tool]: Sending notification to {user_id}: '{message}'...")
    time.sleep(0.1) # Simulate network delay
    if user_id.startswith("user_"):
        return {"status": "success", "user_id": user_id}
    return {"status": "failed", "reason": "Invalid user ID"}

# Mapping of tool names to their actual functions
available_tools = {
    "get_stock_price": get_stock_price,
    "send_notification": send_notification,
}

# --- Mock LLM (Simplified Decision Logic) ---
class MockLLM:
    def __init__(self):
        self.context = [] # Simulates conversation history and previous tool outputs

    def _generate_response_logic(self, query: str) -> Tuple[List[Dict[str, Any]], str]:
        """Internal logic to decide tool calls or text response."""
        tool_calls = []
        text_response = ""

        query_lower = query.lower()

        if "stock price for goog" in query_lower and "notify me" in query_lower:
            # LLM decides it needs two tools in a chained fashion
            tool_calls.append({
                "id": "call_goog_price",
                "function": {"name": "get_stock_price", "arguments": json.dumps({"symbol":
"GOOG"})}
            })
            # It expects the orchestrator to process this first, then it might plan the next
            # For simplicity, if we are in 'run_llm_again' mode, LLM might plan the second tool
directly
            # This is where the core difference is simulated
            tool_calls.append({
                "id": "call_notify",
                "function": {"name": "send_notification", "arguments": json.dumps({"user_id":
"user_abc", "message": "GOOG price is [price_placeholder]"})}
            })
            text_response = "Fetching stock price and preparing notification." # Intermediate
thought/response

        elif "stock price for msft" in query_lower:
            tool_calls.append({
                "id": "call_msft_price",
                "function": {"name": "get_stock_price", "arguments": json.dumps({"symbol":
"MSFT"})}
            })
            text_response = "Fetching MSFT stock price."

        else:
            text_response = "I can help with stock prices and notifications. Please ask."

        return tool_calls, text_response

    def invoke(self, messages: List[Dict[str, Any]], stop_on_first_tool: bool = False) ->
Tuple[List[Dict[str, Any]], str]:
        """
        Simulates one LLM invocation.
```

```
        Returns (generated_tool_calls, generated_text_response_if_no_tools_or_final).
        """
        print(f"\n[LLM Invocation]: Messages received. Stop on first tool: {stop_on_first_tool}")

        # Get the latest user query or tool output as the current prompt
        latest_message = messages[-1]["content"] if messages else ""

        # Simulate LLM's thinking process to decide on tool calls
        all_planned_tool_calls, text_output_if_no_tools =
self._generate_response_logic(latest_message)

        if all_planned_tool_calls:
            if stop_on_first_tool:
                # If stop_on_first_tool is True, return only the first planned tool call
                print("  [LLM]: Decided to stop after planning the first tool.")
                return [all_planned_tool_calls[0]], "" # No text response if tool is called
            else:
                # If run_llm_again is the strategy (stop_on_first_tool=False), return all planned
tool calls
                print(f"  [LLM]: Decided to plan {len(all_planned_tool_calls)} tools in this
turn.")
                return all_planned_tool_calls, "" # No text response if tools are called

        print("  [LLM]: Decided not to use tools. Generating text response.")
        return [], text_output_if_no_tools # No tool calls, only text

# --- Agent Orchestrator ---
class AgentOrchestrator:
    def __init__(self, llm: MockLLM):
        self.llm = llm
        self.conversation_messages = []

    def run_flow(self, user_query: str, strategy: str):
        print(f"\n--- Running Agent with Strategy: {strategy} ---")
        self.conversation_messages.append({"role": "user", "content": user_query})

        run_completed = False
        while not run_completed:
            print(f"\n[Orchestrator]: Current messages for LLM: {self.conversation_messages[-
1]['content'][:50]}...")

            # --- Key Difference Logic ---
            if strategy == "stop_on_first_tool":
                # Only allows LLM to plan one tool at a time
                tool_calls, llm_text_response = self.llm.invoke(self.conversation_messages,
stop_on_first_tool=True)
            elif strategy == "run_llm_again":
                # Allows LLM to plan multiple tools in one go for efficiency
                tool_calls, llm_text_response = self.llm.invoke(self.conversation_messages,
stop_on_first_tool=False)
            else:
                print("Invalid strategy.")
                break

            if tool_calls:
                print(f"[Orchestrator]: LLM requested {len(tool_calls)} tool call(s).")
                tool_outputs_to_add_to_context = []

                # Execute all requested tools
                for tool_call in tool_calls:
                    function_name = tool_call["function"]["name"]
                    function_args = json.loads(tool_call["function"]["arguments"])
```

```
                    print(f"  [Orchestrator]: Executing tool:
{function_name}({function_args})...")
                    func = available_tools.get(function_name)
                    if func:
                        output = func(**function_args)
                        tool_output_content = json.dumps(output)
                        print(f"  [Orchestrator]: Tool '{function_name}' output: {output}")
                    else:
                        tool_output_content = json.dumps({"error": "Tool not found"})
                        print(f"  [Orchestrator]: Tool '{function_name}' not found.")

                    tool_outputs_to_add_to_context.append({
                        "role": "tool",
                        "content": tool_output_content,
                        "tool_call_id": tool_call["id"]
                    })

                # Add tool outputs to conversation context for next LLM invocation
                self.conversation_messages.extend(tool_outputs_to_add_to_context)

                # In 'run_llm_again' scenario, LLM will be invoked again automatically here
                # In 'stop_on_first_tool' scenario, this is where you'd typically stop and wait
for another user input/explicit resume
                # For this simulation, we'll continue the loop to represent "running LLM again"
after outputs
                print("[Orchestrator]: Tool outputs added to context. Invoking LLM again (next
loop iteration).")

            elif llm_text_response:
                print(f"\n[Assistant]: {llm_text_response}")
                run_completed = True # LLM gave a final text response, so task is completed

            else:
                print("[Orchestrator]: No tools or text response. Something went wrong or task
completed ambiguously.")
                run_completed = True # Break loop to prevent infinite loop

# --- Run Demonstrations ---
if __name__ == "__main__":
    orchestrator = AgentOrchestrator(llm=MockLLM())

    # Scenario 1: stop_on_first_tool (less performance, more control)
    # LLM will plan get_stock_price, stop. Then you feed output. Then LLM plans send_notification.
    print("***** Scenario 1: Using 'stop_on_first_tool' (Higher Latency/Cost, More Control)
*****")
    orchestrator.run_flow("What's the GOOG stock price and notify me?", "stop_on_first_tool")

    print("\n" + "="*120 + "\n")
    orchestrator = AgentOrchestrator(llm=MockLLM()) # Reset orchestrator for clear demo

    # Scenario 2: run_llm_again (better performance, less granular control)
    # LLM will plan BOTH get_stock_price and send_notification in one go.
    # Orchestrator executes both, then LLM is run *once* with both outputs.
    print("***** Scenario 2: Using 'run_llm_again' behavior (Lower Latency/Cost, Less Granular
Control) *****")
    orchestrator.run_flow("What's the GOOG stock price and notify me?", "run_llm_again")
```

Code Example Ki Wazahat ():

1. **`MockTool Definitions`**: Do saade tools - `get_stock_price` aur `send_notification` - banaye gaye hain, jin mein thodi si `time.sleep` delay shamil ki gayi hai takay network latency ka ehsaas ho.
2. **`MockLLM` Class**:
    - Yeh LLM ki taraf se tool-calling aur response generation ko simulate karta hai.
    - **`_generate_response_logic`**: Yeh internal method hai jo LLM ki asli "planning" ko simulate karti hai. Humne use is tarah code kiya hai ke agar query mein "stock price" aur "notify" dono shamil hon, to woh **dono tools** ko call karne ka plan banata hai.
    - **`invoke(messages, stop_on_first_tool=False)`** : Yeh woh aham method hai jahan `stop_on_first_tool` flag ka istemal hota hai.
        - Agar `stop_on_first_tool` `True` hai, to `invoke` method LLM ke plan kiye gaye tools mein se **sirf pehla tool call** return karta hai.
        - Agar `stop_on_first_tool` `False` hai (jo `run_llm_again` ke behavior ko mimic karta hai), to yeh LLM ke plan kiye gaye **tamam tool calls** (agar hon) return karta hai.
3. **`AgentOrchestrator` Class**:
    - Yeh hamare agent ka main runner hai. Yeh user query ko `MockLLM` ke paas bhejta hai aur phir LLM ke jawab (tool calls ya text) ki bunyad par agla qadam uthata hai.
    - **`run_flow(user_query, strategy)`**: Yeh method `strategy` parameter (`"stop_on_first_tool"` ya `"run_llm_again"`) ki bunyad par `self.llm.invoke` ko call karta hai.
    - **Loop (`while not run_completed`):** Yeh loop LLM invocations aur tool executions ke poore cycle ko simulate karta hai jab tak LLM final text response na de.
        - **Performance Farq Yahan Nazar Aata Hai:**
            - **`stop_on_first_tool` strategy mein:** LLM har baar sirf aik tool call generate karta hai. Iska matlab hai ke har tool call ke baad loop dobara chalega aur `self.llm.invoke` dobara call hoga (jo aik LLM API call ko simulate karta hai). Yani, 2 tools ke liye 2 LLM invocations ki zaroorat padegi (plus final response ke liye aik aur).
            - **`run_llm_again` strategy mein:** LLM aik hi `self.llm.invoke` call mein dono tools ko plan kar leta hai. `Orchestrator` un dono ko execute karta hai, aur phir LLM ko un dono outputs ke sath **aik hi baar** dobara invoke karta hai (ya is case mein, loop ka next iteration). Isse LLM invocations ki tadaad kam ho jati hai.

Scenarios ():

- **Scenario 1: `stop_on_first_tool` (`"What's the GOOG stock price and notify me?"`)**
    1. **LLM Invocation 1:** `MockLLM` ko query milti hai. `stop_on_first_tool=True` hone ki wajah se, yeh sirf `get_stock_price("GOOG")` ko plan karta hai aur return karta hai.
    2. **Tool Execution 1:** Orchestrator `get_stock_price` ko execute karta hai.
    3. **LLM Invocation 2:** Orchestrator `MockLLM` ko dobara invoke karta hai (pehle tool output ke sath). Ab LLM ko `send_notification` plan karna hoga.
    4. **Tool Execution 2:** Orchestrator `send_notification` ko execute karta hai.
    5. **LLM Invocation 3:** Orchestrator `MockLLM` ko dobara invoke karta hai (doosre tool output ke sath). Ab LLM final text response generate karta hai.
    - **Natija: 3 LLM invocations** (API calls) hotay hain, jo ziyada latency aur cost ka sabab ban sakte hain.
    -

Scenario 2: `run_llm_again` (implicit, by `stop_on_first_tool=False`) (`"What's the GOOG stock price and notify me?"`)

0. **LLM Invocation 1:** `MockLLM` ko query milti hai. **`stop_on_first_tool=False`** hone ki wajah se, yeh `get_stock_price("GOOG")` aur `send_notification(...)` **dono** ko plan karta hai aur return karta hai.
1. **Tool Execution (Batch):** Orchestrator `get_stock_price` aur `send_notification` **dono** ko execute karta hai.

2. **LLM Invocation 2:** Orchestrator `MockLLM` ko dobara invoke karta hai (dono tools ke outputs ke sath). Ab LLM final text response generate karta hai.
   - **Natija:** Sirf **2 LLM invocations** (API calls) hotay hain, jo kam latency aur cost ka sabab bante hain.

Summary ()

`run_llm_again` aur **`stop_on_first_tool`** ke darmiyan performance ka buniyadi farq **LLM (API) invocations ki tadaad** mein hai.

- **`stop_on_first_tool`**: Har tool call ke baad agent ruk jata hai, jis se **ziyada LLM invocations** (yani ziyada API calls) hotay hain. Iska natija **ziyada latency (deri) aur ziyada cost** hota hai. Yeh har qadam par ziyada control faraham karta hai.
- **`run_llm_again`** **(implicit behavior ya batch processing)**: LLM ko ijazat deta hai ke woh aik hi baar mein kayi tools ko plan kare. Agent un sab ko execute karta hai aur phir unke outputs ko aik hi baar mein LLM ko wapas bhej deta hai. Is se **LLM invocations ki tadaad kam** ho jati hai, jis ka natija **kam latency aur kam cost** hota hai. Yeh autonomous (khud-mukhtar) aur efficient multi-step tasks ke liye behtar hai.

Chunkay har LLM invocation (ya API call) mein network latency aur processing time shamil hota hai, `run_llm_again` ka ravaiya (ya **`stop_on_first_tool`** ko `False` set karna) aam taur par **behtar performance** deta hai production environments mein jahan efficiency aham hai.

28. Which Pydantic v2 decorator is used for field validation ?

Pydantic v2 Mein Field Validation Ke Liye Konsa Decorator Istemal Hota Hai?

Pydantic v2 mein **field validation** ke liye istemal hone wala buniyadi decorator `Annotated` ke saath `BeforeValidator` ya `AfterValidator` hai, jinhein `pydantic.functional_validators` module se import kiya jata hai.

Definition ()

`BeforeValidator` aur `AfterValidator` Pydantic v2 mein khaas decorators hain jo aapko fields ki values ko **model mein parse hone se pehle (Before)** ya **parse hone ke baad (After)** validate ya transform karne ki ijazat dete hain. Ye `Annotated` type ke andar istemal hote hain taake type hints ke zariye custom validation logic ko fields se joda ja sake.

- **`BeforeValidator`**: Yeh validator function input data ko model field mein convert karne se pehle (raw input par) chalaya jata hai. Agar aapko raw input ko saaf karna, validate karna ya transform karna ho to yeh behtareen hai.
- **`AfterValidator`**: Yeh validator function data ke model field mein convert hone ke baad (parsed value par) chalaya jata hai. Agar aapko value ki final form ko validate karna ho, ya us par mazeed checks lagane hon to yeh mufeed hai.

Explanation ()

Pydantic v1 mein, aap `validator` decorator (`@validator('field_name')`) istemal karte the. Pydantic v2 ne validation ke tareeqe ko mazeed flexible aur type-safe banane ke liye `Annotated` aur functional validators ka concept introduce kiya hai.

`Annotated` Python 3.9+ mein shamil kiya gaya aik feature hai jo aapko type hints mein metadata add karne ki ijazat deta hai. Pydantic is metadata ko custom validation logic attach karne ke liye istemal karta hai.

Kaam Karne Ka Tareeqa:

1. **`Annotated` ka Istemal:** Aap apni field ke type hint ko `Annotated` ke andar wrap karte hain. Pehla argument field ka asal type hota hai (maslan `str`, `int`, `float`).
2. **Validator Ko Import Karna:** Aap `pydantic.functional_validators` se `BeforeValidator`, `AfterValidator` (ya dusre functional validators jaise `PlainValidator`, `WrapValidator`) ko import karte hain.
3. **Validation Function Likhna:** Aap aik regular Python function likhte hain jo validation logic perform karta hai. Is function ko kam az kam aik argument (value) milta hai.
    - **`BeforeValidator` function:** Is function ko raw input value milti hai. Jo kuch bhi yeh function return karta hai woh agle validator ya parsing step ko pas kar diya jata hai.
    - **`AfterValidator` function:** Is function ko pehle se parse shuda value milti hai (jo `BeforeValidator` se guzar chuki ho aur Pydantic ke default parsing se bhi). Jo kuch bhi yeh function return karta hai woh field ki final value ban jati hai.
4. **Decorator Ke Zariye Link Karna:** Aap `BeforeValidator` ya `AfterValidator` ko apne validation function par decorator ke taur par istemal karte hain, aur phir is decorated function ko `Annotated` ke doosre argument ke taur par pas karte hain.

Pydantic v2 mein, har validator aik simple function hota hai, decorator nahin jo kisi class method se jura ho, jaisa ke v1 mein hota tha. Is se validation functions ko reusable aur testable banana asan ho jata hai.

Example Code ()

Aaiye in decorators ka practical istemal dekhte hain:

```
from typing import Annotated
from pydantic import BaseModel, ValidationError, Field
from pydantic.functional_validators import BeforeValidator, AfterValidator

# --- Validation Functions ---

# Example 1: BeforeValidator - Email ko lowercase karna aur basic format check
def validate_and_normalize_email(email: str) -> str:
    """
    Email ko lowercase karta hai aur basic format check karta hai.
    Agar invalid ho to ValueError raise karta hai.
    """
    if not isinstance(email, str):
        raise ValueError("Email must be a string")

    normalized_email = email.lower().strip()
    if "@" not in normalized_email or "." not in normalized_email:
        raise ValueError(f"Invalid email format: {email}")
    return normalized_email

# Example 2: AfterValidator - Age ki range validate karna
def validate_age_range(age: int) -> int:
    """
    Verify karta hai ke age 0 aur 120 ke darmiyan hai.
    """
    if not isinstance(age, int):
        raise ValueError("Age must be an integer")
    if not 0 <= age <= 120:
        raise ValueError(f"Age {age} must be between 0 and 120")
    return age # Hamesha validated value return karein

# Example 3: BeforeValidator & AfterValidator combination
def ensure_string_is_not_empty(v: str) -> str:
    """Ensure string is not empty after stripping whitespace."""
    stripped_v = v.strip()
    if not stripped_v:
        raise ValueError("String cannot be empty or just whitespace")
    return stripped_v

def validate_min_length(s: str) -> str:
    """Ensure string has a minimum length after parsing."""
    if len(s) < 5:
        raise ValueError("String must be at least 5 characters long")
    return s

# --- Pydantic Model Definition ---

class User(BaseModel):
    # 'email' field: BeforeValidator email ko normalize aur validate karega
    email: Annotated[str, BeforeValidator(validate_and_normalize_email)]

    # 'age' field: AfterValidator age ki range ko validate karega
    age: Annotated[int, AfterValidator(validate_age_range)]
```

```
    # 'username' field: BeforeValidator aur AfterValidator dono
    username: Annotated[str,
                        BeforeValidator(ensure_string_is_not_empty),
                        AfterValidator(validate_min_length)]

    # Optional field with default value and simple validation
    status: Annotated[str, Field(default="active")] = "active"


# --- Testing the Model ---

print("--- Valid Data Examples ---")
try:
    user1 = User(email="Test@EXAMPLE.com ", age=30, username="user123")
    print(f"User 1 (Valid): {user1.model_dump()}")
    # Output: User 1 (Valid): {'email': 'test@example.com', 'age': 30, 'username': 'user123',
'status': 'active'}

    user2 = User(email="ANOTHER@domain.ORG", age=1, username="my_user")
    print(f"User 2 (Valid): {user2.model_dump()}")
    # Output: User 2 (Valid): {'email': 'another@domain.org', 'age': 1, 'username': 'my_user',
'status': 'active'}

except ValidationError as e:
    print(f"Error (unexpected): {e}")


print("\n--- Invalid Data Examples ---")

# Invalid Email Format
try:
    User(email="invalid-email", age=25, username="testuser")
except ValidationError as e:
    print(f"\nError (Invalid Email): {e}")
    # Expected: Invalid email format: invalid-email

# Age out of range
try:
    User(email="test@example.com", age=150, username="testuser")
except ValidationError as e:
    print(f"\nError (Invalid Age): {e}")
    # Expected: Age 150 must be between 0 and 120

# Empty Username (BeforeValidator handles this)
try:
    User(email="test@example.com", age=30, username="   ")
except ValidationError as e:
    print(f"\nError (Empty Username): {e}")
    # Expected: String cannot be empty or just whitespace

# Username too short (AfterValidator handles this)
try:
    User(email="test@example.com", age=30, username="abc")
except ValidationError as e:
    print(f"\nError (Short Username): {e}")
    # Expected: String must be at least 5 characters long

# Invalid type for email (BeforeValidator handles type check)
try:
    User(email=123, age=30, username="validuser")
except ValidationError as e:
    print(f"\nError (Email wrong type): {e}")
    # Expected: Email must be a string
```

Code Example Ki Wazahat ():

1. **Validation Functions:**
    - `validate_and_normalize_email`: Yeh aik `BeforeValidator` ke liye function hai. Yeh email string ko lowercase aur strip karta hai (`.lower().strip()`). Agar email mein `@` ya `.` na ho to `ValueError` raise karta hai. Yeh raw input par kaam karta hai.
    - `validate_age_range`: Yeh aik `AfterValidator` ke liye function hai. Yeh is baat ki tasdeeq karta hai ke integer `age` ki value 0 aur 120 ke darmiyan hai. Yeh Pydantic ke default parsing ke baad chalta hai.
    - `ensure_string_is_not_empty`: Yeh `BeforeValidator` ke liye hai. Yeh string se whitespace hata kar check karta hai ke woh khali to nahin hai.
    - `validate_min_length`: Yeh `AfterValidator` ke liye hai. Yeh `ensure_string_is_not_empty` se guzarne aur Pydantic ke default parsing ke baad string ki lambai check karta hai.
2. **`User` Model:**
    - **`email` field:** `Annotated[str, BeforeValidator(validate_and_normalize_email)]` syntax use karta hai. Iska matlab hai ke `email` asal mein `str` type ka hai, lekin is par `validate_and_normalize_email` function `BeforeValidator` ke taur par apply hoga.
    - **`age` field:** `Annotated[int, AfterValidator(validate_age_range)]` use karta hai. Iska matlab hai ke `age` asal mein `int` type ka hai, aur `validate_age_range` function `AfterValidator` ke taur par apply hoga.
    - **`username` field:** Yahan aap dekh sakte hain ke `Annotated` ke andar **kayi validators** ko aik sath specify kiya ja sakta hai. Pydantic unhein usi tarteeb se chalayega jis tarteeb se woh `Annotated` mein diye gaye hain, aur `BeforeValidator` pehle chalenge, phir `AfterValidator`.
    - **`status` field:** Yeh `Annotated` ke saath `Field` ka istemal dikhata hai, jo Pydantic v2 mein default values aur metadata define karne ka naya tareeqa hai.
3. **Testing:** Code different valid aur invalid inputs ke sath `User` model ko instatiate karne ki koshish karta hai, aur `ValidationError` ko catch karta hai taake validate kiya ja sake ke validation sahih kaam kar raha hai.

Summary ()

Pydantic v2 mein **field validation** ke liye, aap **`Annotated`** type hint ka istemal karte hain, jiske andar aap **`BeforeValidator`** aur **`AfterValidator`** (jo `pydantic.functional_validators` module se import kiye jate hain) ko apni custom validation functions ke saath specify karte hain.

- **`BeforeValidator`:** Data ko **parse hone se pehle** transform ya validate karta hai (raw input par kaam karta hai).
- **`AfterValidator`:** Data ke **parse hone ke baad** uski final form ko validate karta hai (parsed value par kaam karta hai).

Yeh naya tareeqa Pydantic v1 ke `@validator` decorator se ziyada flexible, type-safe, aur reusable validation logic banane ki ijazat deta hai. Ab validation functions simple Python functions hain jinhein kisi class se jora hona zaroori nahin, isse unka test karna aur dusre contexts mein istemal karna asan ho jata hai.

29. What is the purpose of model_settings in an Agent ?

Agent Mein `model_settings` Ka Maqsad Kya Hai?

Definition ()

Agent frameworks mein, khaas taur par jo OpenAI ke models istemal karte hain, `model_settings` aik configuration object ya parameter hai jo **underlying Large Language Model (LLM)** ke behavior ko control aur customize karne ke liye istemal hota hai. Is ka maqsad Agent ko taqatwar LLM ki mukhtalif capabilities aur performance characteristics ko behtar tareeqay se istemal karne ki ijazat dena hai.

Explanation ()

Jab aap koi Agent banate hain, to woh aam taur par kisi na kisi LLM (jaise GPT-4, GPT-3.5) par munhasir hota hai takay reasoning, text generation, aur tool selection jaise kaam kar sake. `model_settings` aapko is LLM ke mukhtalif pehluon ko configure karne ki ijazat deta hai. Yeh settings LLM ki output quality, generation process, aur resource usage ko mutasir kar sakti hain.

`model_settings` ke andar aam taur par shamil hone wali settings:

1. **`model` (Model Name/ID):**
   - **Maqsad:** Yeh sabse bunyadi setting hai jo batati hai ke Agent konsa khaas LLM model istemal karega. Misal ke taur par, "gpt-4o", "gpt-4-turbo", "gpt-3.5-turbo". Mukhtalif models ki qeemat, speed, aur capability mukhtalif hoti hai.
   - **Example:** `model="gpt-4o"`
2. **`temperature`:**
   - **Maqsad:** Yeh output ki randomness ya creativity ko control karta hai. Ziyada `temperature` (maslan 0.8-1.0) ziyada mutanawwi aur creative (magar kam predictable) jawab paida karega, jabkay kam `temperature` (maslan 0.1-0.3) ziyada markooz aur mutawaqqo (predictable) jawab dega.
   - **Example:** `temperature=0.7` (creativity ke liye) ya `temperature=0.2` (accuracy ke liye)
3. **`max_tokens` (Ya `max_output_tokens`):**
   - **Maqsad:** Yeh LLM ke generate kiye gaye jawab mein tokens ki ziyada se ziyada tadaad ko control karta hai. Iska istemal response ki lambai ko limit karne aur cost ko control karne ke liye kiya jata hai.
   - **Example:** `max_tokens=500`
4. **`top_p`:**
   - **Maqsad:** Yeh "nucleus sampling" ke zariye output ki randomness ko control karta hai. LLM sirf un tokens par ghaur karta hai jin ka cumulative probability `top_p` value se barabar ya kam ho. Yeh `temperature` ki tarah creativity ko mutasir karta hai lekin mukhtalif tareeqay se.
   - **Example:** `top_p=0.9`
5. **`presence_penalty` / `frequency_penalty`:**
   - **Maqsad:** Yeh LLM ke jawab mein naye topics (presence penalty) ya baar baar alfaz (frequency penalty) ko kitna penalize karna hai, usay control karte hain. Inka istemal jawab ko ziyada mukhtalif aur kam dohrane wala banane ke liye kiya jata hai.
   - **Example:** `presence_penalty=0.5`, `frequency_penalty=0.3`

6. **`stop_sequences`:**
    - **Maqsad:** Yeh strings ki aik list hoti hai, jahan LLM ko jawab dena rok dena chahiye agar woh in strings ko generate kare. Misal ke taur par, agar aap agent se sirf aik sentence ka jawab chahte hain, to aap `["\n"]` ko stop sequence set kar sakte hain.
    - **Example:** `stop_sequences=["\n", "User:"]`

`model_settings` ka Maqsad:

- **Customization:** LLM ke behavior ko specific task ke mutabiq dhalna. Maslan, creative writing ke liye high temperature, factual summary ke liye low temperature.
- **Performance Optimization:** Token limits set karke cost control karna, ya tez models select karke latency kam karna.
- **Response Quality:** Penalties ya top_p ko adjust karke irrelevant ya dohrane wale content ko kam karna.
- **Resource Management:** Mukhtalif models ki resource demands ko manage karna.

Yeh setting Agent framework ke andar LLM interaction layer par apply hoti hai, is liye Agent ko maloom hota hai ke underlying model ko kis tarah configure karna hai jab woh API calls karta hai.

Example Code ()

OpenAI Assistants API mein **`model_settings`** jaisa koi direct named parameter nahin hota jahan aap dictionary pas karte hon. Iske bajaye, ye settings (jaise `temperature`, `top_p`) `client.beta.threads.runs.create()` function mein **seedhay parameters** ke taur par diye jate hain. Agent frameworks (jaise LangChain) mein, `model_settings` aik conceptual grouping ho sakta hai jise woh internally map karte hain.

Aaiye dekhte hain ke OpenAI Python SDK mein in settings ko kis tarah istemal kiya jata hai, jo **`model_settings`** ke maqsad ko wazeh karta hai:

```
from openai import OpenAI
import os
import time
import json
from dotenv import load_dotenv

load_dotenv()

client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

# --- Mock Tool ---
def get_current_time(timezone: str) -> str:
    """Gets the current time for a given timezone."""
    print(f"\n[Tool Execution]: Getting time for {timezone}...")
    # Simulate different times based on timezone
    if "karachi" in timezone.lower():
        return json.dumps({"time": "4:30 PM", "timezone": "PKT"})
    elif "london" in timezone.lower():
        return json.dumps({"time": "12:30 PM", "timezone": "GMT"})
    return json.dumps({"time": "N/A", "error": "Invalid timezone"})

# Tool mapping for execution
available_functions = {
    "get_current_time": get_current_time
}

# --- Main Assistant Flow with Custom Model Settings ---
```

```
async def demonstrate_model_settings():
    # 1. Assistant create ya retrieve karein
    try:
        # Apne Assistant ID ko yahan dalen
        my_assistant_id = "asst_your_time_assistant_id"
        my_assistant = client.beta.assistants.retrieve(my_assistant_id)
        print(f"Existing Assistant istemal ho raha hai: {my_assistant.id}")
    except Exception:
        print("Naya Assistant banaya ja raha hai...")
        my_assistant = client.beta.assistants.create(
            name="Time Teller Assistant",
            instructions="You are a helpful assistant that can tell the current time in different
timezones.",
            model="gpt-4o", # Ya koi aur tool-enabled model
            tools=[
                {"type": "function", "function": {
                    "name": "get_current_time",
                    "description": "Get the current time for a given timezone",
                    "parameters": {
                        "type": "object",
                        "properties": {
                            "timezone": {"type": "string", "description": "The city or timezone,
e.g. Karachi, London"},
                        },
                        "required": ["timezone"]
                    }
                }}
            ]
        )
        my_assistant_id = my_assistant.id
        print(f"Naya Assistant ban gaya ID ke sath: {my_assistant_id}")

    # 2. Thread create karein
    print("\nNaya Thread banaya ja raha hai...")
    thread = client.beta.threads.create()
    print(f"Naya Thread ban gaya ID ke sath: {thread.id}")

    # 3. User message add karein
    user_message = "What's the current time in Karachi? And be very precise and concise."
    print(f"\nUser: {user_message}")
    client.beta.threads.messages.create(
        thread_id=thread.id,
        role="user",
        content=user_message,
    )

    # 4. Run create karein WITH custom model settings
    # Yahan hum model_settings ka maqsad dekhte hain - LLM ke behavior ko adjust karna
    print("\nRun banaya ja raha hai (custom model settings ke sath)...")
    current_run = client.beta.threads.runs.create(
        thread_id=thread.id,
        assistant_id=my_assistant_id,
        # Yahan par 'model_settings' ke parameters directly diye jate hain
        # Yeh LLM ke response ko affect karega
        temperature=0.1,  # Kam temperature matlab ziyada precise aur kam creative
        max_tokens=50,    # Max output tokens ko limit karna
        # stop=["."]        # Jawab ko "." par rok denge agar zaroorat ho
    )
    print(f"Run ban gaya ID ke sath: {current_run.id}, ibtedai status: {current_run.status}")

    # 5. Run status ko poll karein
    print("\nRun status check kiya ja raha hai...")
    while current_run.status == "queued" or current_run.status == "in_progress":
        time.sleep(0.5)
```

```
        current_run = client.beta.threads.runs.retrieve(thread_id=thread.id,
run_id=current_run.id)
        print(f"Run status: {current_run.status}")

    # 6. Agar requires_action hai, tools execute karein
    if current_run.status == "requires_action":
        print("\nRun requires_action state mein hai: Assistant tool call karna chahta hai.")
        tool_outputs_to_submit = []
        for tool_call in current_run.required_action.submit_tool_outputs.tool_calls:
            print(f"  Tool Call ID: {tool_call.id}")
            print(f"  Function Name: {tool_call.function.name}")
            print(f"  Arguments: {tool_call.function.arguments}")

            function_name = tool_call.function.name
            function_args = json.loads(tool_call.function.arguments)

            output = ""
            if function_name in available_functions:
                output = available_functions[function_name](**function_args)
            else:
                output = json.dumps({"error": f"Function {function_name} not found"})

            tool_outputs_to_submit.append({
                "tool_call_id": tool_call.id,
                "output": output,
            })
            print(f"  Tool Output: {output}")

        print("\nTool outputs submit kiye ja rahe hain...")
        current_run = client.beta.threads.runs.submit_tool_outputs(
            thread_id=thread.id,
            run_id=current_run.id,
            tool_outputs=tool_outputs_to_submit,
        )
        print(f"Run dobara shuru ho gaya status ke sath: {current_run.status}")

        # Submit karne ke baad dobara poll karein completion tak
        while current_run.status == "queued" or current_run.status == "in_progress":
            time.sleep(0.5)
            current_run = client.beta.threads.runs.retrieve(thread_id=thread.id,
run_id=current_run.id)
            print(f"Run status: {current_run.status}")

    # 7. Final Response
    if current_run.status == "completed":
        print("\nRun mukammal ho gaya. Assistant ka final message hasil kiya ja raha hai...")
        messages = client.beta.threads.messages.list(thread_id=thread.id, order="desc", limit=1)
        if messages.data and messages.data[0].role == "assistant" and
messages.data[0].content[0].type == "text":
            print(f"\nAssistant: {messages.data[0].content[0].text.value}")
        else:
            print("\nAssistant ne text response nahin diya ya response ghair-mutawaqqo format mein
hai.")
    elif current_run.status == "failed":
        print(f"\nRun failed. Last error: {current_run.last_error}")
    else:
        print(f"\nRun ghair-mutawaqqo status par khatam hua: {current_run.status}")

if __name__ == "__main__":
    import asyncio
    asyncio.run(demonstrate_model_settings())
```

Code Example Ki Wazahat ():

1. **Assistant Aur Thread Banana:** Hum aik Assistant (jiske paas `get_current_time` tool hai) aur aik naya conversation thread banate hain.
2. **`client.beta.threads.runs.create(...)` Mein `model_settings` Ka Istemal:**
    - **Aham Nuqta:** Jab hum `Run` create karte hain, to hum `temperature=0.1` aur `max_tokens=50` jaisi values directly `runs.create()` function ko pas karte hain. OpenAI SDK mein, yehi parameters asal mein `model_settings` ka maqsad poora karte hain. LangChain jaise frameworks mein, aap inhein shayad `model_settings` dictionary ke andar define karenge, aur woh framework internally inhein sahih API parameters mein map karega.
    - `temperature=0.1` set karne ka matlab hai ke LLM ziyada **precise aur kam creative** jawab dega. Humne prompt mein "very precise and concise" bhi kaha hai, jo LLM ko is temperature setting ke sath mazeed control dega.
    - `max_tokens=50` set karne ka matlab hai ke LLM ka jawab **50 tokens se ziyada nahin** hoga, jo jawab ki lambai ko control karega.
3. **Run Processing:** Baqi flow pehle ki misal jaisa hi hai:
    - `Run` create hota hai.
    - Jab Assistant `get_current_time` tool ko call karne ka faisla karta hai, to `Run` `requires_action` state mein chala jata hai.
    - Hum tool ko execute karte hain (`get_current_time` function ko call karte hain).
    - Tool output ko `submit_tool_outputs` ke zariye Assistant ko wapas bhejte hain.
    - `Run` ke `completed` hone ka intizar karte hain.
    - Assistant ka final message retrieve aur print karte hain.

Is misal mein, aap dekhenge ke Assistant ka jawab `temperature` aur `max_tokens` ki settings ki wajah se **kam creative aur mukhtasar** hoga, jo **`model_settings`** ke maqsad ko wazeh karta hai.

Summary ()

Agent mein **`model_settings`** ka buniyadi maqsad **underlying LLM (Large Language Model) ke ravaiye ko customize aur control karna** hai. Yeh aapko LLM ki output generation, creativity, verbosity (baat ki lambai), aur resource usage jaisi cheezon ko apne task ya application ki zarooriyat ke mutabiq adjust karne ki ijazat deta hai.

Mukhtasaran, `model_settings` Agent ko is qabil banate hain ke woh LLM ki poori taqat ko control shuda tareeqay se istemal kar sake, jis se **behtar performance, cost-efficiency, aur mutawaqqo (predictable) results** hasil ho saken. OpenAI SDK mein, ye settings `runs.create()` jaise API calls mein directly parameters ke taur par diye jate hain, jabkay kuch Agent frameworks inhein aik **`model_settings`** dictionary mein group kar sakte hain.

30. How do you enable non-strict mode for flexible schemas ?

Pydantic Mein Flexible Schemas Ke Liye Non-Strict Mode Kaise Enable Karein?

Pydantic mein **non-strict mode** enable karne ka matlab hai ke aap apne data models ko **flexible schemas** ke sath kaam karne ki ijazat dete hain. Yeh us waqt mufeed hota hai jab aapka incoming data aapke model ki exact type definitions se bilkul match nahin karta, lekin aap phir bhi usay process karna chahte hain, bajaye iske ke validation error ho jaye.

Definition ()

Non-strict mode Pydantic models ke liye aik configuration setting hai jo Pydantic ko input data ko model fields ke types mein convert karne (type coercion) mein **ziyada liberal** banata hai. Jab non-strict mode enable hota hai, Pydantic type conversions ko allow karta hai jo `strict` mode mein errors produce karengay. Misal ke taur par, numeric strings ko integers mein convert karna, ya `True`/`False` strings ko booleans mein convert karna.

Explanation ()

Pydantic by default aik balance approach istemal karta hai. Yeh common sense type coercions ki ijazat deta hai (maslan, "123" ko `int(123)` mein convert karna), lekin kuch conversions par sakhti baratta hai. Non-strict mode is sakhti ko kam karta hai.

Non-strict mode enable karne ke liye, aap model ki configuration mein `strict=False` set karte hain.

Default Behavior (`strict=False` implicit): Pydantic ka default behavior already kafi non-strict hai. Misal ke taur par:

- `"1"` ko `int` field mein parse kar dega.
- `"true"` ya `"false"` ko `bool` field mein parse kar dega.
- `[1, 2, 3]` ko `Set[int]` mein parse kar dega.

`strict=True` Mode: Jab aap `strict=True` set karte hain, to Pydantic bahut sakht ho jata hai. Sirf woh values allow karta hai jo field ke type se bilkul match karti hon. Misal ke taur par:

- `int` field ke liye sirf asal `int` value qabool karega, `"1"` ko nahin.
- `bool` field ke liye sirf asal `True` ya `False` qabool karega, `"true"` ko nahin.
- `list` field ke liye sirf asal `list` qabool karega, `tuple` ko nahin.

Non-strict Mode (`strict=False` explicitly ya by default): `strict=False` set karna (ya default behaviour par inhesar karna) Pydantic ko type conversions mein ziyada lenient banata hai. Iska matlab hai ke Pydantic koshish karega ke incoming data ko aapki schema types mein fit kare, bhale hi woh asal mein match na karta ho.

Kab istemal karein:

- Jab aap **external APIs** se data receive kar rahe hon jahan data types hamesha consistent nahin hotay (maslan, JSON mein numbers strings ke taur par aa sakte hain).
- Legacy systems ke sath integrate karte waqt.

- Jab aap **flexible data ingestion** chahte hon aur sirf un errors par tawajjuh dena chahte hon jo conversion ke baad bhi theek nahin ho sakte.

Caveats (Ehtiyatein):

- **Data Integrity:** Ziyada flexibility data integrity ke issues paida kar sakti hai. Aapka model galat types ke data ko silently accept kar sakta hai jo aap expect nahin kar rahe the.
- **Unexpected Behavior:** Automatic conversions kabhi kabhi ghair-mutawaqqo tareeqay se kaam kar sakte hain.
- **Debugging:** Type conversion errors ko debug karna mushkil ho sakta hai agar aap bahut ziyada `strict=False` par inhesar karte hain.

Is liye, Pydantic mein flexibility ke liye `strict=False` mode (default behavior) istemal karna **data sources par inhesar karta hai jahan aapko maloom hai ke kuch type mismatches expected hain aur unhein gracefully handle karna chahiye.** Agar aap complete type safety aur debugging mein asani chahte hain, to `strict=True` istemal karna behtar ho sakta hai aur aapko explicit parsing functions likhni chahiye.

Example Code ()

Aaiye dekhte hain ke **`strict=True`** aur **`strict=False`** (default) kis tarah kaam karte hain:

```
from pydantic import BaseModel, ValidationError, Field, ConfigDict

# --- Strict Mode Example ---
# Model jo strict mode mein hai
class StrictUser(BaseModel):
    model_config = ConfigDict(strict=True) # Strict mode enable kiya

    id: int
    name: str
    is_active: bool
    data_list: list[int]

print("--- Strict Mode ---")
try:
    # Valid input (types perfectly match)
    user_strict_valid = StrictUser(id=1, name="Ali", is_active=True, data_list=[10, 20])
    print(f"StrictUser (Valid): {user_strict_valid.model_dump()}")
except ValidationError as e:
    print(f"StrictUser (Valid) Error (Unexpected): {e}")

try:
    # Invalid input (string for int) - strict mode mein fail hoga
    user_strict_invalid_int = StrictUser(id="2", name="Sara", is_active=True, data_list=[10, 20])
    print(f"StrictUser (Invalid int): {user_strict_invalid_int.model_dump()}")
except ValidationError as e:
    print(f"StrictUser (Invalid int) Error: {e}")
    # Expected: id: Input should be a valid integer, got type str

try:
    # Invalid input (string for bool) - strict mode mein fail hoga
    user_strict_invalid_bool = StrictUser(id=3, name="Ahmed", is_active="false", data_list=[10,
20])
    print(f"StrictUser (Invalid bool): {user_strict_invalid_bool.model_dump()}")
except ValidationError as e:
    print(f"StrictUser (Invalid bool) Error: {e}")
    # Expected: is_active: Input should be a valid boolean, got type str
```

```
try:
    # Invalid input (tuple for list) - strict mode mein fail hoga
    user_strict_invalid_list = StrictUser(id=4, name="Fatima", is_active=True, data_list=(1, 2,
3))
    print(f"StrictUser (Invalid list): {user_strict_invalid_list.model_dump()}")
except ValidationError as e:
    print(f"StrictUser (Invalid list) Error: {e}")
    # Expected: data_list: Input should be a valid list, got type tuple


# --- Non-Strict Mode (Default Behavior) Example ---
# Model jo non-strict mode mein hai (yaani strict=False by default)
class FlexibleUser(BaseModel):
    # strict=False explicit specify karna ya omit karna - dono ka result same hoga
    model_config = ConfigDict(strict=False) # Non-strict mode (default)

    id: int
    name: str
    is_active: bool
    data_list: list[int]

print("\n--- Non-Strict Mode (Default) ---")
try:
    # Valid input (types perfectly match)
    user_flexible_valid = FlexibleUser(id=1, name="Ali", is_active=True, data_list=[10, 20])
    print(f"FlexibleUser (Valid): {user_flexible_valid.model_dump()}")
except ValidationError as e:
    print(f"FlexibleUser (Valid) Error (Unexpected): {e}")

try:
    # Invalid input (string for int) - non-strict mode mein pass hoga
    user_flexible_invalid_int = FlexibleUser(id="2", name="Sara", is_active=True, data_list=[10,
20])
    print(f"FlexibleUser (String to int): {user_flexible_invalid_int.model_dump()}")
    # Output: FlexibleUser (String to int): {'id': 2, 'name': 'Sara', 'is_active': True,
'data_list': [10, 20]}
except ValidationError as e:
    print(f"FlexibleUser (String to int) Error (Unexpected): {e}")

try:
    # Invalid input (string for bool) - non-strict mode mein pass hoga
    user_flexible_invalid_bool = FlexibleUser(id=3, name="Ahmed", is_active="false",
data_list=[10, 20])
    print(f"FlexibleUser (String to bool): {user_flexible_invalid_bool.model_dump()}")
    # Output: FlexibleUser (String to bool): {'id': 3, 'name': 'Ahmed', 'is_active': False,
'data_list': [10, 20]}
except ValidationError as e:
    print(f"FlexibleUser (String to bool) Error (Unexpected): {e}")

try:
    # Invalid input (tuple for list) - non-strict mode mein pass hoga
    user_flexible_invalid_list = FlexibleUser(id=4, name="Fatima", is_active=True, data_list=(1,
2, 3))
    print(f"FlexibleUser (Tuple to list): {user_flexible_invalid_list.model_dump()}")
    # Output: FlexibleUser (Tuple to list): {'id': 4, 'name': 'Fatima', 'is_active': True,
'data_list': [1, 2, 3]}
except ValidationError as e:
    print(f"FlexibleUser (Tuple to list) Error (Unexpected): {e}")

try:
    # Mixed list items - non-strict mode mein fail hoga (nested type for list item is strict)
    # Note: List ke andar ke items ki validation hamesha strict hoti hai jab tak ke unhein
explicitly non-strict na kiya jaye.
    # Yani, [1, "2", 3] -> list[int] mein "2" fail karega.
```

```
    user_flexible_mixed_list = FlexibleUser(id=5, name="Zain", is_active=True, data_list=[1, "2",
3])
    print(f"FlexibleUser (Mixed List): {user_flexible_mixed_list.model_dump()}")
except ValidationError as e:
    print(f"FlexibleUser (Mixed List) Error: {e}")
    # Expected: data_list.1: Input should be a valid integer, got type str
```

Code Example Ki Wazahat ():

1. **`StrictUser` Model:**
    - Humne `ConfigDict(strict=True)` set kiya hai.
    - Aap dekhenge ke jab `id` ko string `"2"` ke taur par, `is_active` ko string `"false"` ke taur par, ya `data_list` ko `tuple` ke taur par diya jata hai, to `StrictUser` model **`ValidationError`** raise karta hai. Iska matlab hai ke yeh model sirf exact type matches qabool karta hai.
2. **`FlexibleUser` Model:**
    - Humne `ConfigDict(strict=False)` set kiya hai (ya is line ko omit bhi kar sakte hain, kyunki `strict=False` default hai).
    - Aap dekhenge ke is model mein:
        - `id="2"` (string) ko `id=2` (integer) mein **safalta se convert** kar diya gaya.
        - `is_active="false"` (string) ko `is_active=False` (boolean) mein **safalta se convert** kar diya gaya.
        - `data_list=(1, 2, 3)` (tuple) ko `data_list=[1, 2, 3]` (list) mein **safalta se convert** kar diya gaya.
    - Akhiri misal mein, `data_list=[1, "2", 3]` fail ho jata hai. Yeh is baat ko wazeh karta hai ke `strict=False` sirf **top-level container types** (jaise `list` aur `tuple` ke darmiyan) aur **simple primitive types** (string to int/bool) ke liye kaam karta hai. Nested types (list ke andar `int` ki jaga `str`) ab bhi default strictness rakhte hain, jab tak ke unhein explicitly `Annotated` aur custom validators se handle na kiya jaye.

Summary ()

Pydantic v2 mein flexible schemas ko enable karne ya **non-strict mode** mein kaam karne ke liye, aap apne `BaseModel` ki `model_config` mein **`strict=False`** set karte hain. Yaad rakhiye, `strict=False` Pydantic ka **default behavior** hai, isliye agar aap ise specify nahin bhi karte hain, to bhi aap non-strict mode mein honge.

Yeh setting Pydantic ko type coercion mein ziyada liberal banati hai, jahan woh incoming data ko aapke model fields ke expected types mein convert karne ki koshish karta hai, bhale hi woh asal mein thoda mukhtalif ho (maslan, numeric strings ko integers mein badalna). Jab aapko un data sources se deal karna ho jahan type consistency ki guarantee na ho, to yeh behtareen hai, lekin data integrity ke masail se bachne ke liye ehtiyat zaroori hai.

31. What does the handoff_description parameter do ?

`handoff_description` Parameter Kya Karta Hai?

Definition ()

`handoff_description` parameter Agent frameworks mein istemal hota hai (khaas taur par LangChain jaisay jo mukhtalif "Agents" ya "Assistants" ke darmiyan coordination ko allow karte hain). Yeh aik **mukhtasar, wazeh text description** hai jo is baat ki wazahat karta hai ke jab aik Agent apna kaam mukammal kar leta hai ya mazeed koi action nahin le sakta, to woh **aglay Agent ya insani user ko kis buniyad par control (ya "handoff") de raha hai.**

Explanation ()

Complex tasks mein, aik single Agent tamam kaam nahin kar sakta. Kabhi kabhi, aik Agent ko dosre specialized Agent ko task pas karna hota hai, ya insani dakhal-andazi ki zarurat padti hai. `handoff_description` is transition ko asan aur mufeed banata hai.

Maqsad:

1. **Context Transfer (Context Ki Muntaqili):** Jab aik Agent apna kaam mukammal karta hai aur aglay Agent ko control deta hai, to `handoff_description` aglay Agent ko yeh batata hai ke **pehle Agent ne kya achieve kiya hai, kya progress hui hai, aur agla qadam kya hona chahiye.** Yeh aik "briefing" jaisa kaam karta hai.
2. **Human Guidance (Insani Rehnumai):** Agar Agent ko insani review ya action ki zarurat hai (maslan, aik khareedari ki tasdeeq, ya kisi mushkil faisle ke liye), to `handoff_description` user ko batata hai ke **Agent ne kahan tak kaam kiya hai aur ab user se kya umeed ki ja rahi hai.** Yeh user experience ko behtar banata hai aur confusion ko kam karta hai.
3. **Error Handling (Ghaltiyon Ko Handle Karna):** Agar Agent kisi aisi mushkil mein phas jata hai jahan woh khud se aagay nahin badh sakta (maslan, tool fail ho gaya, ya user ki request clear nahin hai), to `handoff_description` is masle ko aglay handler ya user ko wazeh karta hai.
4. **Orchestration Clarity (Orchestration Ki Wazahat):** Multiple Agents ke darmiyan complex workflows mein, `handoff_description` yeh tay karta hai ke control kis Agent ko milega aur kis wajah se. Yeh orchestration logic ko mazeed robust aur samajhne mein asan banata hai.
5. **Audit Trail (Record Rakhna):** Yeh agent ke decision-making process ka aik useful hissa ban jata hai, jo baad mein debugging ya process ko samajhne mein madad karta hai.

Aam taur par `handoff_description` mein kya shamil hota hai:

- **Progress Summary:** Jo kaam ab tak kiya gaya hai uski mukhtasar summary.
- **Current State:** Task ki maujooda haalat.
- **Reason for Handoff:** Control transfer karne ki wajah (maslan, task mukammal ho gaya, mazeed tools ki zarurat hai, insani dakhal-andazi chahiye).
- **Next Steps/Expected Action:** Aglay Agent ya user se kya qadam uthane ki umeed hai.
- **Relevant Data (optional):** Handoff ke liye zaruri kuch data points.

OpenAI SDK mein Direct Parameter nahin:

`handoff_description` **OpenAI Python SDK ya Assistants API ka seedha parameter nahin hai.** Yeh **Agent orchestration frameworks** (jaise LangChain, AutoGen, ya custom Agent loops) mein ek conceptual parameter ya pattern hai jo Developers khud implement karte hain jab woh multiple Assistants ko chain karte hain ya user interaction ko manage karte hain. Jab ek Agent dusre Agent ko ya user ko `tool_output` submit karta hai ya final `message` deta hai, to us message ya `tool_output` ke content mein yeh `handoff_description` shamil kiya ja sakta hai.

Example Code ()

Chunkay `handoff_description` OpenAI SDK ka seedha parameter nahin hai, hum ise LangChain jaisay Agent framework ke conceptual implementation ya custom Agent flow mein simulate karenge.

Hum do **Mock Agents** banayenge:

1. **`DataFetchAgent`**: Jo user data fetch karega.
2. **`EmailSendAgent`**: Jo email bhejega.

Aur aik **Orchestrator** jo in Agents ke darmiyan `handoff_description` ko pass karega.

```
import time
import json
from typing import Dict, Any, Tuple, Optional

# --- Mock Tools ---
def get_user_data(user_id: str) -> Dict[str, Any]:
    """Fetches user profile data."""
    print(f"[Tool]: Fetching data for user: {user_id}")
    time.sleep(0.5)
    if user_id == "user123":
        return {"id": user_id, "name": "Alice", "email": "alice@example.com", "is_premium": True}
    return {"error": "User not found", "id": user_id}

def send_email_tool(to_email: str, subject: str, body: str) -> Dict[str, Any]:
    """Sends an email."""
    print(f"[Tool]: Sending email to {to_email} with subject: '{subject}'")
    time.sleep(0.5)
    if "@" in to_email:
        return {"status": "success", "recipient": to_email}
    return {"status": "failed", "recipient": to_email, "reason": "Invalid email"}

available_tools = {
    "get_user_data": get_user_data,
    "send_email_tool": send_email_tool,
}

# --- Mock LLM Simulation ---
class MockLLM:
    def decide_action(self, task_description: str, context: Dict[str, Any]) -> Tuple[str,
Optional[Dict[str, Any]], str]:
        """
        Simulates LLM's decision: (action_type, tool_call_details, final_response)
        action_type: "tool_call", "final_response", "handoff"
        """
        if "fetch user" in task_description.lower() and "data_fetched" not in context:
```

```
            print("[LLM]: Decided to fetch user data.")
            return "tool_call", {"name": "get_user_data", "args": {"user_id": "user123"}}, ""

        elif "send email" in task_description.lower() and "user_email" in context and "email_sent"
not in context:
            print("[LLM]: Decided to send email.")
            user_email = context.get("user_email", "default@example.com")
            email_body = f"Hello {context.get('user_name', 'User')},\n\n"
            email_body += f"Your data has been processed. Status: {context.get('data_status',
'N/A')}\n"
            if context.get("data_status") == "success":
                email_body += f"Your premium status: {context.get('is_premium', 'N/A')}.\n"

            return "tool_call", {"name": "send_email_tool", "args": {"to_email": user_email,
"subject": "Update on Your Request", "body": email_body}}, ""

        elif "data_fetched" in context and "email_sent" not in context:
            # If data is fetched but email not sent, it's a handoff scenario
            # This is where handoff_description is implicitly formed by the LLM's reasoning
            print("[LLM]: Data fetched. Handoff to EmailSendAgent needed.")
            handoff_reason = "User data has been successfully fetched and is ready for further
processing, specifically email notification. The fetched data includes email and name."
            return "handoff", None, handoff_reason # handoff_description in the third return value

        print("[LLM]: Task completed or no further action required.")
        return "final_response", None, "Task has been completed."

# --- Mock Agent Classes ---
class BaseAgent:
    def __init__(self, name: str):
        self.name = name
        self.llm = MockLLM()
        self.context = {} # Agent's internal context

    def execute_tool(self, tool_name: str, args: Dict[str, Any]) -> Dict[str, Any]:
        func = available_tools.get(tool_name)
        if func:
            try:
                return func(**args)
            except Exception as e:
                return {"error": str(e), "status": "tool_failure"}
        return {"error": "Tool not found", "status": "tool_failure"}

class DataFetchAgent(BaseAgent):
    def __init__(self):
        super().__init__("Data Fetch Agent")

    def run(self, initial_task: str, incoming_context: Dict[str, Any] = {}) -> Tuple[str,
Dict[str, Any], str]:
        """
        Runs the data fetching process.
        Returns: (status, context, handoff_description)
        """
        self.context.update(incoming_context)
        print(f"\n[{self.name}]: Starting with task: '{initial_task}'")

        action_type, tool_details, handoff_message = self.llm.decide_action(initial_task,
self.context)

        if action_type == "tool_call" and tool_details and tool_details["name"] ==
"get_user_data":
            user_data = self.execute_tool(tool_details["name"], tool_details["args"])
            if "error" not in user_data:
                self.context["data_fetched"] = True
```

```
                self.context["user_email"] = user_data["email"]
                self.context["user_name"] = user_data["name"]
                self.context["is_premium"] = user_data["is_premium"]
                self.context["data_status"] = "success"

                # --- HANDOFF DESCRIPTION IS FORMED HERE ---
                # This agent knows it needs to handoff for email sending
                handoff_description = (
                    f"User data for {user_data['name']} (ID: {user_data['id']}) has been
successfully retrieved. "
                    f"Email: {user_data['email']}. This data is now available for sending a
notification email."
                )
                print(f"[{self.name}]: Data fetched. Handoff to next agent or user required.")
                return "handoff", self.context, handoff_description
            else:
                handoff_description = f"Failed to fetch user data for
{tool_details['args']['user_id']}: {user_data['error']}"
                print(f"[{self.name}]: Failed to fetch data. Handoff to user for review.")
                return "failed_handoff", self.context, handoff_description

        print(f"[{self.name}]: Completed or no relevant action.")
        return "completed", self.context, "No further action from Data Fetch Agent."

class EmailSendAgent(BaseAgent):
    def __init__(self):
        super().__init__("Email Send Agent")

    def run(self, incoming_task_context: Dict[str, Any], handoff_description_from_prev: str) ->
Tuple[str, Dict[str, Any], str]:
        """
        Runs the email sending process based on context from previous agent.
        Returns: (status, context, final_message)
        """
        self.context.update(incoming_task_context)
        print(f"\n[{self.name}]: Received handoff. Previous Agent's note:
'{handoff_description_from_prev}'")

        # LLM decides action based on full context including previous handoff
        action_type, tool_details, final_response = self.llm.decide_action("send email",
self.context) # Use a generic task for LLM

        if action_type == "tool_call" and tool_details and tool_details["name"] ==
"send_email_tool":
            # Update placeholder in body
            body_with_price = tool_details["args"]["body"].replace("[price_placeholder]",
str(self.context.get("stock_price", "N/A")))
            tool_details["args"]["body"] = body_with_price

            email_result = self.execute_tool(tool_details["name"], tool_details["args"])
            if email_result.get("status") == "success":
                self.context["email_sent"] = True
                print(f"[{self.name}]: Email successfully sent to {email_result['recipient']}.")
                return "completed", self.context, f"Email successfully sent to
{email_result['recipient']}."
            else:
                return "failed", self.context, f"Failed to send email: {email_result['reason']}.
Please review."

        return "failed", self.context, "Email Send Agent could not complete its task."


# --- Orchestrator ---
def main_orchestrator(initial_user_request: str):
```

```
    print(f"Orchestrator: Starting with user request: '{initial_user_request}'")

    current_context = {}

    # Step 1: Data Fetch Agent
    data_agent = DataFetchAgent()
    status, updated_context, handoff_msg_data_agent = data_agent.run(initial_user_request,
current_context)

    if status == "handoff":
        current_context.update(updated_context)
        print(f"\nOrchestrator: Handoff from DataFetchAgent. Handoff Description:
'{handoff_msg_data_agent}'")

        # Step 2: Email Send Agent - receiving handoff_description
        email_agent = EmailSendAgent()
        final_status, final_context, final_message = email_agent.run(current_context,
handoff_msg_data_agent)

        print(f"\nOrchestrator: Final Status: {final_status}")
        print(f"Orchestrator: Final Message: {final_message}")
    else:
        print(f"\nOrchestrator: DataFetchAgent did not complete successfully or did not handoff.
Status: {status}")
        print(f"Orchestrator: Message: {handoff_msg_data_agent}")

# --- Run the demonstration ---
if __name__ == "__main__":
    main_orchestrator("I need user data for user123 and then please send them an email.")
```

Code Example Ki Wazahat ():

1. **`MockTools`**: `get_user_data` aur `send_email_tool` asal tools ko simulate karte hain.
2. **`MockLLM`**:
   - Yeh LLM ke **decision-making** ko simulate karta hai.
   - `decide_action` method `action_type` return karta hai (`"tool_call"`, `"final_response"`, `"handoff"`).
   - **Aham Nuqta:** Jab yeh `handoff` ka action decide karta hai (maslan, `data_fetched` context mein hai lekin email nahin bheja gaya), to yeh **`handoff_message`** ko third return value ke taur par deta hai. Yeh message `handoff_description` ka maqsad poora karta hai.
3. **`BaseAgent`**: Aik buniyadi class hai jo agents ke liye common functionality (LLM aur tool execution) faraham karti hai.
4. **`DataFetchAgent`**:
   - Yeh `get_user_data` tool ko istemal karta hai.
   - **`run` method mein:** Jab yeh successfully user data fetch kar leta hai, to yeh aik **`handoff_description` string** banata hai (`"User data for Alice (ID: user123) has been successfully retrieved..."`).
   - Yeh `handoff_description` phir `return "handoff", self.context, handoff_description` ke zariye Orchestrator ko wapas bhej diya jata hai.
5. **`EmailSendAgent`**:
   - Yeh `send_email_tool` ko istemal karta hai.
   - **`run` method mein:** Yeh `handoff_description_from_prev` parameter ko receive karta hai. Yeh is baat ko simulate karta hai ke aglay Agent ko maloom hai ke pichle Agent ne kya kaam kiya hai. Yeh is information ko apne internal reasoning ya logs mein istemal kar sakta hai.
6. **`main_orchestrator`**:
   - Yeh overall workflow ko control karta hai.
   - Pehle `DataFetchAgent` ko chalata hai.

- Agar `DataFetchAgent` `handoff` status return karta hai, to `orchestrator` **`handoff_msg_data_agent` (jo asal mein hamara `handoff_description` hai)** ko capture karta hai.
- Phir, yeh `EmailSendAgent` ko invoke karta hai aur is `handoff_msg_data_agent` ko `EmailSendAgent` ke `run` method mein **pass kar deta hai**.

Is misal se wazeh hota hai ke **`handoff_description`** aik structured message hai jo Agents ke darmiyan ya Agent aur user ke darmiyan **context aur irade (intent) ko wazeh karne** ke liye istemal hota hai jab control transfer hota hai.

Summary ()

`handoff_description` parameter (ya is jaisa koi concept) Agent frameworks mein aik **maloomati string** hoti hai. Iska maqsad yeh hai ke jab aik Agent apna task mukammal karta hai, kisi mushkil mein phas jata hai, ya mazeed action ke liye kisi dosre Agent ya insani user ko control dena chahta hai, to woh **aglay Agent ya user ko is transition ke bare mein mukhtasar, wazeh aur relevant tafseelat faraham karta hai.**

Yeh description aik Agent ke zariye ki gayi progress, maujooda haalat, control transfer karne ki wajah, aur aglay qadam ke bare mein guidelines faraham karta hai. Is se multi-agent systems mein **orchestration mein wazahat aati hai, context ka loss kam hota hai, aur user experience behtar hota hai.** Yaad rahe ke yeh OpenAI SDK ka seedha parameter nahin hai, balkay Agent frameworks mein aik design pattern hai.

32. How do you implement **schema evolution** while maintaining backward compatibility ?

Schema Evolution Ko Backward Compatibility Ke Sath Kaise Implement Karein?

Schema evolution se murad hai waqt ke sath data schema mein tabdeeliyan karna jabkay **backward compatibility** is baat ko yaqeeni banati hai ke naye schema ke mutabiq data (jo naye format mein store ya process kiya gaya ho) ko purane code ya systems ke zariye bhi parha (read) aur interpret (samjha) ja sake. Yeh distributed systems, microservices, aur long-lived applications mein aik bohat ahem challenge hai.

Definition ()

Schema Evolution: Kisi bhi structured data format (jaise database tables, API payloads, message queues, configuration files) mein hone wali tabdeeliyan, maslan fields ka add karna, remove karna, rename karna, ya types ko modify karna.

Backward Compatibility (): Yeh is baat ki salahiyat hai ke naya software (ya naye schema ke mutabiq data) purane software (ya purane schema ke mutabiq data) ke sath baghair kisi masle ke kaam kar sake. Yani, purana code naye data ko theek tarah se handle kar sake.

Explanation ()

Schema evolution ko backward compatibility ke sath implement karna zaroori hai takay aap apne systems ko update karte waqt disruption ko kam se kam kar saken. Agar aap backward compatibility maintain nahin karte, to aapko aksar "big bang" deployments karne padte hain jahan aapko aik hi waqt mein poore system (database, microservices, frontends) ko update karna padta hai, jo ke bohat mushkil aur risky hota hai.

Buniyadi Usool Aur Tareeqe (Key Principles and Techniques):

1. **Additive Changes (Izafi Tabdeeliyan):**
    - **New Fields Are Optional:** Jab aap koi naya field add karte hain, to usay **optional** (ya default value ke sath) banayein. Is tarah, purana code jo is naye field se waqif nahin hai, woh isay ignore kar dega, aur naya code isay istemal kar sakega.
    - **Example:** Agar `User` schema mein `phone_number` add kar rahe hain, to usay `Optional[str]` ya `str | None` banayein.
    - **Impact on Old Code:** Purana code naye field ko nahin dekhega. Naya code naye field ko istemal kar sakta hai, aur agar purane data mein woh field mojood nahin hai to `None` ya default value milegi.
2. **Non-Breaking Changes (Ghair-Tootne Wali Tabdeeliyan):**
    - **Adding New Enum Values:** Enum mein naye values add karna aam taur par non-breaking hota hai, jab tak ke purana code naye values ko handle karne ki ummeed na rakhta ho.
    - **Impact on Old Code:** Purana code naye enum values ko ignore kar sakta hai (ya fallback mechanism use kar sakta hai).
3. **Deprecation Strategy (Niskh Karne Ki Hikmat-e-Amali):**
    - **Soft Deletion / Deprecation:** Kisi field ko fori taur par remove karne se gurez karein. Pehle usay deprecated mark karein (documentation mein, ya code mein annotations ke sath).
    - **Grace Period:** Aik munafiq waqt dein takay dependent services aur clients us field ka istemal band kar saken. Is dauran, data ko dono formats mein support kiya ja sakta hai (ya naye format mein migrate kiya ja sakta hai).

- **Impact on Old Code:** Purana code abhi bhi deprecated field ko istemal kar sakta hai, lekin developers ko maloom ho jayega ke yeh field mustaqbil mein remove ho jayega.

4. **Field Removal (Fields Ko Hatana):**
   - Yeh aik **backward-incompatible** change hai. Is se bachna chahiye jab tak ke aap "big bang" deployment na kar saken.
   - **Solution:** Agar hatana zaroori hai, to pehle use deprecated karein, aur phir usay remove karne se pehle poore system ko upgrade kar lein. Ya phir, **data migration** aur **versioning** ka istemal karein.
5. **Field Renaming (Fields Ka Naam Badlna):**
   - Yeh bhi aik **backward-incompatible** change hai.
   - **Solution:** Naye naam ke sath naya field add karein (additive change), purane field ko deprecated karein, aur naye code ko naye field par migrate karein. Aik "shim" (temporary compatibility layer) istemal kar sakte hain jo purane aur naye naam ke darmiyan mapping provide kare.
   - **Example:** `user_name` se `full_name` tak. `full_name` add karein, `user_name` ko deprecated karein.
6. **Type Changes (Types Ki Tabdeeli):**
   - **Narrowing Type (Type ko tang karna):** `str` se `int` tak - yeh aam taur par **backward-incompatible** hai kyunki purana data naye type mein fit nahin hoga.
   - **Widening Type (Type ko wasee karna):** `int` se `str` tak - yeh aam taur par **backward-compatible** hai (agar numbers ko strings mein convert kiya ja sake), lekin purane code ko naye type ke mutabiq data handle karna mushkil ho sakta hai.
   - **Solution:** Is se bachna chahiye. Agar zaroori hai, to proper data migration aur transformation ki zaroorat padegi.
7. **Data Versioning (Data Versioning):**
   - **Concept:** Apne data ke sath schema version number shamil karein (maslan, API response mein `version: 1` ya database table mein `schema_version` column).
   - **Implementation:** Jab data read kiya jata hai, to software version number ko check karta hai aur us version ke mutabiq parsing ya transformation logic apply karta hai.
   - **Example:**

```JSON
// Version 1
{ "id": 1, "name": "Ali" }

// Version 2
{ "version": 2, "id": 1, "name": "Ali", "email": "ali@example.com" }
```

   - **Impact:** Yeh aik powerful tareeqa hai jo aapko mutiple schema versions ko aik hi waqt mein support karne ki ijazat deta hai.
8. **Default Values (Default Values):**
   - Naye non-optional fields ke liye default values set karein. Jab purana data (jis mein naya field nahin hai) load hoga, to Pydantic ya database default value assign kar dega.
   - **Example:** `age: int = 0` (agar purane data mein age nahin thi, to 0 set ho jayegi).

Example Code (Pydantic ke Hawale Se) ()

Hum Pydantic models ka istemal karte hue schema evolution ko backward compatibility ke sath dikhayenge.

```
from pydantic import BaseModel, Field, ValidationError
from typing import Optional, List, Dict, Any, Literal
import json

# --- Version 1 Schema ---
print("--- Version 1 Schema (Original) ---")

class UserV1(BaseModel):
    id: int
    name: str # Old field: name

try:
    user_v1_data = UserV1(id=1, name="Ali Khan")
    print(f"UserV1 (Valid): {user_v1_data.model_dump()}")
except ValidationError as e:
    print(f"UserV1 Error: {e}")

# --- Version 2 Schema Evolution ---
# Tabdeelian:
# 1. `name` field ko `first_name` aur `last_name` mein alag kiya (breaking change agar direct
rename karein).
#    Solution: `name` ko optional/deprecated rakha, naye fields add kiye.
# 2. `email` field add kiya (additive change).
# 3. `status` field add kiya with default value (additive change).
# 4. `roles` field add kiya (additive change).
# 5. Data versioning add kiya.

class UserV2(BaseModel):
    version: Literal[2] = 2 # Data versioning field
    id: int
    first_name: str
    last_name: str

    # Old field 'name' for backward compatibility. Use Optional.
    # Purana code jo 'name' ki umeed rakhta hai, woh isay read kar sakega.
    # Naya code ya to isay ignore kar dega ya migration logic use karega.
    name: Optional[str] = None # Deprecated/legacy field

    email: Optional[str] = Field(None, description="User's email address") # New optional field
    status: Literal["active", "inactive", "pending"] = "active" # New field with default
    roles: List[str] = Field(default_factory=list, description="List of user roles") # New list
field

    # Pydantic v2 ka model_validator istemal kar ke data migration handle karein
    # This acts as a bridge for backward compatibility
    from pydantic import model_validator

    @model_validator(mode='before')
    @classmethod
    def handle_v1_data(cls, data: Any) -> Any:
        if isinstance(data, dict):
            # Agar purana 'name' field mojood hai aur 'first_name' ya 'last_name' nahin hain,
            # to 'name' ko split karke naye fields mein map karein.
            if 'name' in data and 'first_name' not in data and 'last_name' not in data:
                name_parts = data['name'].split(' ', 1)
                data['first_name'] = name_parts[0]
                data['last_name'] = name_parts[1] if len(name_parts) > 1 else ''
                # 'name' ko bhi rakhein taake purana code isay read kar sake,
                # lekin hamare model mein ab primary fields first/last_name hain.
                # data.pop('name', None) # Agar aap name ko remove karna chahte hain after
migration

            # Agar email ya roles field purane data mein nahin hain, Pydantic default provide
karega
```

```
            # Agar 'version' field nahin hai, to assume V1 data
            if 'version' not in data:
                print("    [Schema Evolution]: Converting V1 data to V2 format...")
                data['version'] = 2 # Default to current version
        return data

print("\n--- Version 2 Schema (Evolved with Backward Compatibility) ---")

# Example 1: Loading new data (naturally compatible with V2)
try:
    user_v2_new_data = UserV2(
        id=2,
        first_name="Sara",
        last_name="Ali",
        email="sara@example.com",
        status="active",
        roles=["admin", "editor"]
    )
    print(f"UserV2 (New Data): {user_v2_new_data.model_dump()}")
except ValidationError as e:
    print(f"UserV2 (New Data) Error: {e}")

# Example 2: Loading V1 data (Backward Compatibility Test)
# Is data mein 'version', 'first_name', 'last_name', 'email', 'status', 'roles' nahin hain.
# Pydantic ka model_validator aur default values isay theek karengay.
v1_raw_data = {"id": 3, "name": "Usman Tariq"}
try:
    user_v2_from_v1_data = UserV2(**v1_raw_data)
    print(f"UserV2 (From V1 Data): {user_v2_from_v1_data.model_dump()}")
    # Expected Output: {'version': 2, 'id': 3, 'first_name': 'Usman', 'last_name': 'Tariq',
'name': 'Usman Tariq', 'email': None, 'status': 'active', 'roles': []}
except ValidationError as e:
    print(f"UserV2 (From V1 Data) Error: {e}")

# Example 3: Loading V1 data with just one name part
v1_single_name_data = {"id": 4, "name": "Zain"}
try:
    user_v2_from_v1_single_name = UserV2(**v1_single_name_data)
    print(f"UserV2 (From V1 Single Name): {user_v2_from_v1_single_name.model_dump()}")
    # Expected: 'first_name': 'Zain', 'last_name': ''
except ValidationError as e:
    print(f"UserV2 (From V1 Single Name) Error: {e}")

# Example 4: Loading V2 data that only provides 'first_name' and 'last_name' (no 'name')
v2_without_old_name = {"id": 5, "first_name": "Maria", "last_name": "Hussain", "email":
"maria@example.com"}
try:
    user_v2_no_old_name = UserV2(**v2_without_old_old_name)
    print(f"UserV2 (V2 without old name field): {user_v2_no_old_name.model_dump()}")
    # Expected: 'name': None
except ValidationError as e:
    print(f"UserV2 (V2 without old name field) Error: {e}")
```

Code Example Ki Wazahat ():

1. **`UserV1` (Original Schema):**
    - Yeh hamara purana schema hai jismein sirf `id` aur `name` fields hain.
2. **`UserV2` (Evolved Schema):**
    - **`version: Literal[2] = 2`**: Humne aik `version` field add kiya hai. `Literal[2]` yeh ensure karta hai ke iski value 2 hi hogi, aur `= 2` default value provide karta hai. Yeh data versioning ka aik simple tareeqa hai.
    - **`first_name: str, last_name: str`**: Humne `name` field ko `first_name` aur `last_name` mein split kiya hai.
    - **`name: Optional[str] = None`**: Yeh bahut ahem hai **backward compatibility** ke liye. Humne purane `name` field ko model mein **barqarar** rakha hai, lekin usay `Optional` banaya hai aur default `None` diya hai. Is tarah, purana code jo `name` field ko expect karta hai, woh abhi bhi data ko read kar sakega, aur naya data `name` ke baghair bhi theek se parse ho jayega.
    - **`email: Optional[str] = Field(None, ...)`**: Naya field jo optional hai. Purane data mein yeh field nahin hoga, to Pydantic isay `None` set kar dega.
    - **`status: Literal[...] = "active"`**: Naya field default value ke sath. Purane data mein `status` nahin hoga, to `active` set ho jayega.
    - **`roles: List[str] = Field(default_factory=list, ...)`**: Naya list field default empty list ke sath.
    - **`@model_validator(mode='before') handle_v1_data`**: Yeh Pydantic v2 ka aik powerful feature hai.
        - `mode='before'` ka matlab hai ke yeh validator **raw input data (dictionary)** par chalega, Pydantic ke parsing se pehle.
        - Is validator mein, hum check karte hain ke kya `name` field mojood hai aur `first_name` ya `last_name` nahin hain (jo ke V1 data ki nishani hai).
        - Agar aisa hai, to hum `name` ko split karte hain aur `first_name` aur `last_name` fields ko dynamically add karte hain `data` dictionary mein.
        - Hum `version` field ko bhi set karte hain agar woh mojood nahin hai.
        - Yeh validator **incoming V1 data ko V2 format mein "migrate" karta hai on-the-fly**, usay load karte waqt.
3. **Testing Examples:**
    - **`UserV2 (New Data)`**: Naya data jo seedhay V2 schema ke mutabiq hai, theek parse hota hai.
    - **`UserV2 (From V1 Data)`**: Yeh wahan hai jahan backward compatibility test hoti hai. Hamari raw `v1_raw_data` mein sirf `id` aur `name` hain. Jab hum `UserV2(**v1_raw_data)` karte hain, to `handle_v1_data` validator chalega, `name` ko split karega, `first_name` aur `last_name` banayega, `version` ko 2 set karega, aur baqi naye fields ko default values (ya `None`) mil jayengi. Resulting model V2 schema ke mutabiq hoga, lekin purane data se bana hai.
    - `UserV2 (From V1 Single Name)`: Dikhata hai ke `name` ko kaise handle kiya jata hai agar is mein sirf aik part ho.
    - `UserV2 (V2 without old name field)`: Yeh dikhata hai ke naya data (jo purana `name` field nahin deta) bhi theek se parse hota hai, aur `name` field `None` set ho jayega.

Is misal mein, humne **additive changes**, **default values**, **optional fields**, aur **`model_validator` ke zariye on-the-fly data migration** ka istemal kiya hai takay schema evolution ko backward compatibility ke sath handle kiya ja sake.

Summary ()

Schema evolution ko backward compatibility ke sath implement karne ka matlab hai ke aap apne data ke structure mein tabdeeliyan is tarah karein ke purana code naye data ko baghair kisi masle ke parh aur samajh sake. Iska buniyadi tareeqa **additive changes** (naye optional fields ya default values ke sath fields add karna) aur **non-breaking changes** ko tarjeeh dena hai.

Pydantic v2 mein, aap yeh strategies asani se implement kar sakte hain:

- Naye fields ko `Optional` ya **default values** ke sath define karein.
- Purane fields ko `Optional` ya `None` ke default ke sath **barqarar** rakhein jab tak ke aap unhein mukammal taur par hata na lein (deprecation policy ke mutabiq).
- `@model_validator(mode='before')` ka istemal karte hue **on-the-fly data migration logic** likhein. Yeh aapko incoming purane version ke data ko naye schema ke mutabiq transform karne ki ijazat deta hai parse hone se pehle.
- **Data versioning** (maslan, `version` field add karna) aik robust tareeqa hai jo aapko runtime par data version ko pehchanne aur uske mutabiq handle karne mein madad karta hai.

In tareeqon ko apnakar, aap apne applications ko smoothly update kar sakte hain, deployment risks ko kam kar sakte hain, aur mukhtalif components (jo mukhtalif speeds par evolve hote hain) ke darmiyan compatibility maintain kar sakte hain.

33. Can dynamic instructions be Async functions ?

Kya Dynamic Instructions Async Functions Ho Sakti Hain?

Haan, **dynamic instructions async functions** **ho sakti hain**, khaas taur par jab aap Agent frameworks mein ya complex conversational AI systems mein kaam kar rahe hon jahan instruction generation mein network calls, database lookups, ya dusre time-consuming operations shamil hon.

Definition ()

Dynamic Instructions: Yeh woh instructions ya guidelines hain jo run-time par generate ya retrieve ki jati hain, na ke hard-code ki jati hain. Yeh Agent ya LLM ke behavior ko current context, user input, ya system state ki bunyad par modify karti hain.

Async Functions (Asynchronous Functions): Python mein `async def` key-word se define kiye gaye functions jo `await` key-word ka istemal kar sakte hain takay non-blocking operations perform kar saken. Yeh I/O-bound tasks (network requests, file operations, database queries) ke liye behtar hain kyunkay yeh operation complete hone ka intizar karte waqt program ke doosre hisson ko chalne ki ijazat dete hain.

Explanation ()

Jab Agent frameworks (maslan, LangChain Agents ya custom agent orchestrators) kaam karte hain, to unhein aksar dynamic instructions ki zarurat padti hai. Misal ke taur par:

- **User ki Permissions ki Bunyad Par Instructions:** Agar user "admin" hai, to Agent ko ziyada powerful tools ya instructions mil sakti hain. In permissions ko database se fetch karna ya authentication service se check karna aik async operation ho sakta hai.
- **Real-time Data Fetching:** Agent ko stock price ki bunyad par advise karna ho, ya current weather ke mutabiq plan karna ho. Yeh data fetch karna aik async network request hoga.
- **Configuration Retrieval:** Agent ki instructions kisi remote configuration service (maslan, Firebase Remote Config, AWS AppConfig) se fetch karna jo run-time par update ho sakti hain.
- **Complex Reasoning:** Agar instructions khud aik doosre LLM call ke zariye generate ki ja rahi hain (meta-prompting), to woh LLM call bhi async ho sakta hai.

Async instructions ke fawaid:

1. **Responsiveness:** Non-blocking I/O operations ki wajah se aapka application responsive rehta hai. Jab instruction fetch ho rahi ho, to main event loop block nahin hota, aur doosre tasks (maslan, user requests) process ho sakte hain.
2. **Efficiency:** Concurrent I/O operations perform karna mumkin ho jata hai, jis se overall throughput behtar hoti hai.
3. **Scalability:** Ziyada requests ya concurrent operations ko efficiently handle karne ki salahiyat behtar hoti hai.
4. **Integration:** Modern APIs aur services (databases, microservices, cloud APIs) aksar async interfaces faraham karte hain, isliye async instructions unke sath seamlessly integrate ho sakti hain.
5.

Nuqsanat (Considerations):

- **Complexity:** Async code likhna aur debug karna sync code se ziyada complex ho sakta hai.
- **Overhead:** Agar instruction generation bahut simple hai aur usmein koi I/O operation shamil nahin hai, to async ka overhead (event loop management) be wajah ho sakta hai.

Example Code ()

Aaiye ek misal dekhte hain jahan Agent ke `system_instruction` ko ek async function ke zariye dynamically generate kiya jata hai. Hum simulate karenge ke instructions kisi remote service se fetch ho rahi hain.

```
import asyncio
import time
import os
from openai import AsyncOpenAI # OpenAI SDK ka async client
from typing import Dict, Any, Optional

# Load environment variables (for API key)
from dotenv import load_dotenv
load_dotenv()

# --- Mock Async Instruction Fetcher ---
async def fetch_dynamic_system_instruction(user_role: str) -> str:
    """
    Simulates fetching dynamic system instructions asynchronously from a remote service.
    This could be a database call, an API call to a config service, etc.
    """
    print(f"\n[Async Instruction Fetcher]: Fetching instructions for role: '{user_role}'...")
    await asyncio.sleep(0.5) # Simulate network latency or database query

    if user_role == "admin":
        return "You are an expert system administrator. You have access to all tools. Be precise
and provide detailed technical solutions."
    elif user_role == "customer_support":
        return "You are a friendly customer support agent. Prioritize user satisfaction. Guide
users step-by-step and offer clear, concise help."
    else:
        return "You are a general-purpose assistant. Be helpful and provide accurate information."

# --- Async Agent Orchestrator ---
class AsyncAgentOrchestrator:
    def __init__(self, openai_api_key: str):
        self.client = AsyncOpenAI(api_key=openai_api_key)
        self.conversation_history = []
        self.assistant_id: Optional[str] = None
        self.thread_id: Optional[str] = None

    async def setup_assistant(self, user_role: str):
        """Sets up the OpenAI Assistant with dynamic instructions."""
        print("[Orchestrator]: Setting up Assistant with dynamic instructions...")

        # Call the async function to get instructions
        dynamic_instructions = await fetch_dynamic_system_instruction(user_role)

        # Check if an assistant already exists for simplicity or create a new one
        if not self.assistant_id:
            print(f"  Creating new Assistant with instructions: '{dynamic_instructions[:70]}...'")
```

```
            my_assistant = await self.client.beta.assistants.create(
                name=f"Dynamic AI Assistant ({user_role.capitalize()})",
                instructions=dynamic_instructions,
                model="gpt-4o", # Ya koi aur tool-enabled model
                tools=[{"type": "code_interpreter"}] # Example tool
            )
            self.assistant_id = my_assistant.id
            print(f"  Assistant created with ID: {self.assistant_id}")
        else:
            print(f"  Updating existing Assistant {self.assistant_id} with new instructions:
'{dynamic_instructions[:70]}...'")
            await self.client.beta.assistants.update(
                assistant_id=self.assistant_id,
                instructions=dynamic_instructions,
            )
            print("  Assistant instructions updated.")

        # Create a new thread for each run
        print("  Creating a new Thread...")
        thread = await self.client.beta.threads.create()
        self.thread_id = thread.id
        print(f"  Thread created with ID: {self.thread_id}")

    async def run_conversation(self, user_query: str):
        """Runs a single turn of conversation."""
        if not self.assistant_id or not self.thread_id:
            print("[Error]: Assistant or Thread not set up. Call setup_assistant first.")
            return

        print(f"\n[User]: {user_query}")
        await self.client.beta.threads.messages.create(
            thread_id=self.thread_id,
            role="user",
            content=user_query,
        )

        current_run = await self.client.beta.threads.runs.create(
            thread_id=self.thread_id,
            assistant_id=self.assistant_id,
        )
        print(f"  Run created with ID: {current_run.id}, status: {current_run.status}")

        while current_run.status in ["queued", "in_progress", "code_interpreter"]:
            await asyncio.sleep(0.5)
            current_run = await self.client.beta.threads.runs.retrieve(
                thread_id=self.thread_id,
                run_id=current_run.id,
            )
            print(f"  Run status: {current_run.status}")

        if current_run.status == "completed":
            messages = await self.client.beta.threads.messages.list(
                thread_id=self.thread_id,
                order="desc",
                limit=1,
            )
            if messages.data and messages.data[0].role == "assistant" and
messages.data[0].content[0].type == "text":
                print(f"[Assistant]: {messages.data[0].content[0].text.value}")
            else:
                print("[Assistant]: No text response.")
        elif current_run.status == "requires_action":
            print("[Run Status]: Requires Tool Action - Not handled in this example.")
        else:
```

```
            print(f"[Run Status]: Unexpected status: {current_run.status}")


# --- Main Async Execution ---
async def main():
    api_key = os.getenv("OPENAI_API_KEY")
    if not api_key:
        print("Error: OPENAI_API_KEY environment variable not set.")
        return

    orchestrator = AsyncAgentOrchestrator(openai_api_key=api_key)

    print("--- Scenario 1: Admin User ---")
    await orchestrator.setup_assistant("admin")
    await orchestrator.run_conversation("Please help me debug a server issue. What's the best
approach for log analysis?")

    print("\n" + "="*80 + "\n")

    # Resetting assistant_id and thread_id to simulate new user/session,
    # and setting up with new instructions
    orchestrator.assistant_id = None
    orchestrator.thread_id = None

    print("--- Scenario 2: Customer Support User ---")
    await orchestrator.setup_assistant("customer_support")
    await orchestrator.run_conversation("My order hasn't arrived yet. What should I do?")

    # Clean up (optional)
    # If you want to delete the assistant after the demo
    # print("\n[Cleanup]: Deleting Assistant...")
    # await client.beta.assistants.delete(orchestrator.assistant_id)

if __name__ == "__main__":
    asyncio.run(main())
```

Code Example Ki Wazahat ():

1. **`AsyncOpenAI` Client**: Hum `openai` library se `AsyncOpenAI` client ko import karte hain, jo asynchronous API calls karne ki ijazat deta hai.
2. **`fetch_dynamic_system_instruction(user_role)` (Async Dynamic Instruction):**
    - Yeh aik `async def` function hai.
    - Yeh simulate karta hai ke system instructions ko kisi external source (maslan, database, microservice, ya config store) se fetch kiya ja raha hai, jismein `await asyncio.sleep(0.5)` ke zariye network latency ka izhar kiya gaya hai.
    - Instruction ka content `user_role` (admin ya customer_support) par munhasir hai, jo isay **dynamic** banata hai.
3. **`AsyncAgentOrchestrator` Class:**
    - Yeh hamare Agent ke core logic ko handle karta hai.
    - **`setup_assistant(user_role)`:**
        - Yeh method `fetch_dynamic_system_instruction` ko `await` karta hai takay dynamic instructions hasil ki ja saken.
        - Phir yeh instructions `client.beta.assistants.create` ya `client.beta.assistants.update` ko `instructions` parameter ke taur par pas ki jati hain.
        - Har setup ke liye naya thread banaya jata hai.

- **`run_conversation(user_query):`** Yeh user query ko process karta hai aur Assistant se jawab hasil karta hai. Yeh bhi `await` calls istemal karta hai `client.beta.threads.messages.create` aur `client.beta.threads.runs.create` ke liye.

4. **`main()` Async Function:**
   - Yeh hamara entry point hai jahan hum `AsyncAgentOrchestrator` ko instantiate karte hain.
   - Hum do scenarios chalate hain: aik "admin" role ke liye, aur doosra "customer_support" role ke liye. Har scenario mein, `setup_assistant` call hoti hai jo **dynamic `async` instructions fetch karti hai**, aur woh instructions phir Assistant ke behavior ko define karti hain.
   - `asyncio.run(main())` ka istemal kiya jata hai takay `main` async function ko execute kiya ja sake.

Is misal mein, aap dekhenge ke `fetch_dynamic_system_instruction` function async hai aur yeh non-blocking tareeqay se apni instructions provide karta hai, jis se Agent setup ka process smoothly chalta hai. Output mein, Assistant ka jawab us role ke mutabiq hoga jiske liye instructions fetch ki gai thin.

Summary ()

Haan, **dynamic instructions async functions** **ho sakti hain**, aur yeh aik aam aur behtareen practice hai modern Agent frameworks aur AI systems mein. Jab Agent ke liye instructions generate karne mein I/O-bound operations (jaise network requests, database queries, ya external service calls) shamil hon, to async functions ka istemal karna **system responsiveness, efficiency, aur scalability** ko behtar banata hai.

Async instructions aapko run-time par Agent ke behavior ko flexibly adapt karne ki ijazat deti hain, jo user ke role, current system state, ya real-time data jaisi cheezon par munhasir ho sakti hain, baghair main application thread ko block kiye.

.

34. What happens when **tool_use_behavior** isset to 'stop_on_first_tool' ?

`tool_use_behavior` Ko `'stop_on_first_tool'` Set Karne Se Kya Hota Hai?

Definition ()

Jab OpenAI Assistants API mein `tool_use_behavior` parameter ko `'stop_on_first_tool'` par set kiya jata hai, to iska matlab hai ke Assistant (LLM) **sirf pehla tool call generate karne ke baad hi run ko rok dega**, aur mazeed koi text response ya doosre tool calls generate nahin karega. Yeh Run `requires_action` state mein chala jayega, aur aapko pehle tool call ko execute kar ke uski output submit karni hogi.

Explanation ()

OpenAI Assistants API mein, jab aap aik `Run` banate hain (`client.beta.threads.runs.create()`), to Assistant us `Run` ko mukammal karne ki koshish karta hai, jismein LLM ke zariye reasoning, tool calls, aur user-facing messages shamil ho sakte hain.

`tool_use_behavior` parameter is run ke dauran tool calls ko kis tarah handle kiya jaye, usay control karta hai. Iski teen (ya mazeed ho sakti hain) mukhtalif qismein hain:

1. `'auto'` **(Default Behavior):**
    - **Maqsad:** Yeh default behavior hai. Assistant ko ijazat hoti hai ke woh apni marzi se mukhtalif tool calls kare, text generate kare, aur kai iterations mein reasoning kare, jab tak woh task ko mukammal na kar le ya `max_iterations` tak na pahunch jaye.
    - **Kaise Kam Karta Hai:** Agar Assistant ko kisi tool ki zarurat padti hai, to woh tool call generate karta hai, aur `Run` `requires_action` state mein chala jata hai. Tool output submit karne ke baad, `Run` dobara shuru hota hai, aur Assistant mazeed tool calls kar sakta hai ya final jawab de sakta hai. Yeh aik "multi-turn tool use" scenario hai.
2. `'stop_on_first_tool'`**:**
    - **Maqsad:** Is setting ka maqsad ye hai ke Agent (LLM) ko **sirf aik tool call generate karne par majboor kiya jaye, aur phir fori taur par ruk jaye.** Chahe Agent mazeed tool calls karna chahta ho ya final jawab dena chahta ho, woh ruk jayega.
    - **Kaise Kam Karta Hai:**
        - Jab aap `Run` banate waqt `tool_use_behavior='stop_on_first_tool'` set karte hain, to Assistant messages ko process karta hai.
        - Agar Assistant ko kisi tool ka istemal karna zaroori lagta hai, to woh **sirf pehla tool call** generate karega.
        - Jaise hi yeh pehla tool call generate hota hai, `Run` ka status `requires_action` par set ho jata hai.
        - Assistant koi mazeed reasoning ya text generation nahin karega is `Run` ke dauran, jab tak aap is tool call ki output submit na karein aur naya `Run` shuru na karein.
    - **Fawaid:**
        - **Fine-grained Control:** Aap LLM ke tool-calling behavior par ziyada control hasil karte hain. Yeh useful hai jab aap tool execution ko manual review ya external logic se handle karna chahte hain.
        - **Debugging:** Tool call chain ko step-by-step debug karna asan ho jata hai.
        - **Complex Workflows:** Jab aap Agent ko kisi orchestration layer (jaise human-in-the-loop) mein integrate kar rahe hon jahan har tool call ke baad koi bahar se action lena ho.
        - **Cost Control:** Aap LLM ko ziyada tokens consume karne se rok sakte hain agar aap sirf pehla necessary tool call chahte hain.

Example Code ()

Aaiye dekhte hain ke `tool_use_behavior='stop_on_first_tool'` kaise kaam karta hai. Hum aik Assistant banayenge jiske paas do tools hain, lekin hum use pehle tool call par hi rok denge.

```
from openai import OpenAI
import os
import time
import json
from dotenv import load_dotenv

load_dotenv()

client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

# --- Mock Tools ---
def get_current_time(timezone: str) -> str:
    """Gets the current time for a given timezone."""
    print(f"\n[Tool Execution]: Executing get_current_time for {timezone}...")
    # Simulate different times
    if "karachi" in timezone.lower():
        return json.dumps({"time": "4:30 PM", "timezone": "PKT"})
    elif "london" in timezone.lower():
        return json.dumps({"time": "12:30 PM", "timezone": "GMT"})
    return json.dumps({"time": "N/A", "error": "Invalid timezone"})

def get_weather(city: str) -> str:
    """Gets the current weather for a given city."""
    print(f"\n[Tool Execution]: Executing get_weather for {city}...")
    time.sleep(0.2) # Simulate slight delay
    if "karachi" in city.lower():
        return json.dumps({"city": "Karachi", "weather": "Sunny", "temperature": "35C"})
    elif "london" in city.lower():
        return json.dumps({"city": "London", "weather": "Cloudy", "temperature": "15C"})
    return json.dumps({"error": "City not found", "city": city})

# Map tool names to functions
available_functions = {
    "get_current_time": get_current_time,
    "get_weather": get_weather
}

# --- Main Assistant Flow ---
async def demonstrate_stop_on_first_tool():
    # 1. Assistant create ya retrieve karein
    try:
        # Apne Assistant ID ko yahan dalen agar pehle se mojood hai
        # ya use comment kar dein agar naya banana hai
        my_assistant_id = "asst_your_multi_tool_assistant_id"
        my_assistant = client.beta.assistants.retrieve(my_assistant_id)
        print(f"Existing Assistant istemal ho raha hai: {my_assistant.id}")
    except Exception:
        print("Naya Assistant banaya ja raha hai...")
        my_assistant = client.beta.assistants.create(
            name="Time & Weather Assistant",
            instructions="You are a helpful assistant that can provide current time and weather.",
            model="gpt-4o", # Ya koi aur tool-enabled model
            tools=[
                {"type": "function", "function": {
                    "name": "get_current_time",
```

```
                    "description": "Get the current time for a given timezone",
                    "parameters": {
                        "type": "object",
                        "properties": {"timezone": {"type": "string", "description": "The city or
timezone, e.g. Karachi, London"}},
                        "required": ["timezone"]
                    }
                }},
                {"type": "function", "function": {
                    "name": "get_weather",
                    "description": "Get the current weather for a given city",
                    "parameters": {
                        "type": "object",
                        "properties": {"city": {"type": "string", "description": "The city name,
e.g. Karachi, London"}},
                        "required": ["city"]
                    }
                }}
            ]
        )
        my_assistant_id = my_assistant.id
        print(f"Naya Assistant ban gaya ID ke sath: {my_assistant_id}")

    # 2. Thread create karein
    print("\nNaya Thread banaya ja raha hai...")
    thread = client.beta.threads.create()
    print(f"Naya Thread ban gaya ID ke sath: {thread.id}")

    # 3. User message add karein jo do tool calls trigger kare
    user_message = "What's the time in Karachi and what's the weather like in London?"
    print(f"\nUser: {user_message}")
    client.beta.threads.messages.create(
        thread_id=thread.id,
        role="user",
        content=user_message,
    )

    # 4. Run create karein with tool_use_behavior='stop_on_first_tool'
    print(f"\nRun banaya ja raha hai with tool_use_behavior='stop_on_first_tool'...")
    current_run = client.beta.threads.runs.create(
        thread_id=thread.id,
        assistant_id=my_assistant_id,
        # Yahan hum 'stop_on_first_tool' set karte hain
        tool_choice='auto', # 'auto' ya 'required' ke sath 'stop_on_first_tool' use hota hai
        tool_use_behavior='stop_on_first_tool'
    )
    print(f"Run ban gaya ID ke sath: {current_run.id}, ibtedai status: {current_run.status}")

    # 5. Run status ko poll karein
    print("\nRun status check kiya ja raha hai (stopping on first tool call)...")
    while current_run.status == "queued" or current_run.status == "in_progress":
        time.sleep(0.5)
        current_run = client.beta.threads.runs.retrieve(thread_id=thread.id,
run_id=current_run.id)
        print(f"Run status: {current_run.status}")

    # 6. Agar requires_action hai, tool execute karein aur output submit karein
    if current_run.status == "requires_action":
        print("\nRun requires_action state mein hai: Assistant ne pehla tool call generate kiya.")
        tool_outputs_to_submit = []

        # NOTE: Hum yahan sirf pehle tool call ko process karenge
        # "tool_use_behavior": "stop_on_first_tool" ki wajah se required_action
        # mein sirf aik tool_call hoga.
```

```
        for tool_call in current_run.required_action.submit_tool_outputs.tool_calls:
            print(f"  Tool Call ID: {tool_call.id}")
            print(f"  Function Name: {tool_call.function.name}")
            print(f"  Arguments: {tool_call.function.arguments}")

            function_name = tool_call.function.name
            function_args = json.loads(tool_call.function.arguments)

            output = ""
            if function_name in available_functions:
                output = available_functions[function_name](**function_args)
            else:
                output = json.dumps({"error": f"Function {function_name} not found"})

            tool_outputs_to_submit.append({
                "tool_call_id": tool_call.id,
                "output": output,
            })
            print(f"  Tool Output: {output}")

            # Since we only stop on first tool, we expect only one tool call here.
            # Agar multiple tool calls hote, to bhi yeh sirf pehla wala dekhta.
            break # Stop after processing the first (and only expected) tool call

        print("\nTool outputs submit kiye ja rahe hain aur Run dobara shuru kiya ja raha hai...")
        current_run = client.beta.threads.runs.submit_tool_outputs(
            thread_id=thread.id,
            run_id=current_run.id,
            tool_outputs=tool_outputs_to_submit,
        )
        print(f"Run dobara shuru ho gaya status ke sath: {current_run.status}")

        # Submit karne ke baad dobara poll karein completion tak
        print("\nRun status check kiya ja raha hai (after first tool submission)...")
        while current_run.status == "queued" or current_run.status == "in_progress":
            time.sleep(0.5)
            current_run = client.beta.threads.runs.retrieve(thread_id=thread.id,
run_id=current_run.id)
            print(f"Run status: {current_run.status}")

        # Ab dobara check karein kya Assistant ko koi aur tool call chahiye
        # Ya final response dena chahta hai
        if current_run.status == "requires_action":
            print("\nAssistant ko ab doosra tool call karna chahiye! (Aap dobara Run ko submit kar
sakte hain)")
            for tool_call in current_run.required_action.submit_tool_outputs.tool_calls:
                print(f"  Second Tool Call identified: {tool_call.function.name}")
            print("Is example mein hum second tool call ko manually execute nahin kar rahe hain.")
            print("Agar aap ise bhi execute karna chahte hain, to is block ko handle karein.")

        elif current_run.status == "completed":
            print("\nRun mukammal ho gaya. Assistant ka final message hasil kiya ja raha hai...")
            messages = client.beta.threads.messages.list(thread_id=thread.id, order="desc",
limit=1)
            if messages.data and messages.data[0].role == "assistant" and
messages.data[0].content[0].type == "text":
                print(f"\nAssistant: {messages.data[0].content[0].text.value}")
            else:
                print("\nAssistant ne text response nahin diya ya response ghair-mutawaqqo format
mein hai.")
        elif current_run.status == "failed":
            print(f"\nRun failed. Last error: {current_run.last_error}")
        else:
```

```
            print(f"\nRun ghair-mutawaqqo status par khatam hua: {current_run.status}")

    else:
        print(f"\nRun ghair-mutawaqqo status par khatam hua: {current_run.status}")


if __name__ == "__main__":
    import asyncio
    asyncio.run(demonstrate_stop_on_first_tool())
```

Code Example Ki Wazahat ():

1. **Assistant Setup:** Hum `get_current_time` aur `get_weather` do tools ke sath aik Assistant banate hain.
2. **User Message:** User ka message ("What's the time in Karachi and what's the weather like in London?") do tools ke istemal ka ishara deta hai.
3. **`tool_use_behavior='stop_on_first_tool'` Ka Istemal:**
    - Jab hum `client.beta.threads.runs.create()` call karte hain, to hum `tool_use_behavior='stop_on_first_tool'` set karte hain.
    - **Pehla Run (`current_run`):** Jab Assistant is message ko process karta hai, to woh pehle `get_current_time` tool call ko identify karta hai (ya `get_weather` ko, LLM ke reasoning par munhasir hai).
    - **Run Ruk Jata Hai:** Jaise hi yeh pehla tool call (`get_current_time` in this case) generate hota hai, `Run` fori taur par `requires_action` state mein chala jata hai. Bhale hi Assistant ko maloom ho ke doosra tool (`get_weather`) bhi call karna hai, woh use generate nahin karta kyunkay `stop_on_first_tool` enabled hai.
    - `print(f" Tool Call ID: {tool_call.id}")` ke neeche, aap output mein sirf aik tool call dekhenge.
4. **Tool Execution Aur Output Submission:**
    - Hum `current_run.required_action.submit_tool_outputs.tool_calls` se tool details nikalte hain. Aap dekhenge ke is list mein **sirf aik tool call** mojood hoga.
    - Hum us tool ko execute karte hain (`get_current_time`).
    - Hum us tool ki output ko `client.beta.threads.runs.submit_tool_outputs()` ke zariye submit karte hain. Is se `Run` dobara shuru hota hai.
5. **Doosra Run (After Submission):**
    - Jab `Run` dobara shuru hota hai, to Assistant ke paas `get_current_time` tool ka result hota hai.
    - Ab Assistant dobara reasoning karta hai. Chunkay pehla hissa (time) mukammal ho gaya hai, ab woh doosre hisse (`get_weather`) ko handle karne ki koshish karega.
    - LLM `get_weather` tool ko call karne ka faisla karega.
    - `tool_use_behavior` ki setting **har naye Run par apply hoti hai**. Agar humne `submit_tool_outputs` ke baad koi naya `tool_use_behavior` specify nahin kiya, to woh `Run` apni default setting (`auto`) par wapas chala jayega. Agar aap chahte hain ke har tool call par woh ruke, to aapko `submit_tool_outputs` ke run update mein bhi `tool_use_behavior='stop_on_first_tool'` dena hoga.
    - Is misal mein, doosre Run mein Assistant `get_weather` tool ko bhi call karega. Lekin hum is example mein us doosre tool call ko execute nahin kar rahe, balki sirf dikha rahe hain ke Assistant ab doosre tool ko call karne ke liye tayyar hai.

Is tarah, **`stop_on_first_tool`** aapko LLM ke tool-calling ke flow ko step-by-step control karne ki ijazat deta hai.

Summary ()

`tool_use_behavior` **ko** `'stop_on_first_tool'` par set karne se OpenAI Assistant ka `Run` **fori taur par ruk jata hai jab woh pehla tool call generate karta hai.** Run `requires_action` state mein chala jata hai, aur Assistant mazeed koi reasoning, text generation, ya doosre tool calls generate nahin karega.

Yeh setting aapko LLM ke tool-calling pipeline par **nihayat zabardast control** deti hai. Yeh khaas taur par un scenarios mein mufeed hai jahan aapko:

- Har tool call ko manual review karna ho.
- Tool execution ko external business logic ya human intervention ke zariye handle karna ho.
- Debugging ke maqsad se tool call chain ko step-by-step trace karna ho.

Iske baad, aapko tool ki output submit karni hogi aur Run ko dobara shuru karna hoga. Is setting ke baghair, Assistant `auto` behavior mein multiple tool calls aur turns khud hi handle karne ki koshish karta hai.

35. What's the difference between mutable and immutable context patterns ?

Mutable Aur Immutable Context Patterns Mein Kya Farq Hai?

Context patterns software architecture mein data ko modules, functions, ya components ke darmiyan share karne ke tareeqay hain. **Mutable** aur **Immutable** ka farq is baat par munhasir hai ke yeh shared data run-time par tabdeel kiya ja sakta hai ya nahin.

Definition ()

Mutable Context Pattern (): Ek aisa pattern jahan data ya state jo context ke taur par share ki jati hai, usay **run-time par modify (tabdeel)** kiya ja sakta hai. Yani, data ko reference ke zariye pas kiya jata hai, aur receiving component us data ke contents ko badal sakta hai, aur yeh tabdeeli doosre components ko bhi nazar aati hai jo ussi context ko share kar rahe hain.

Immutable Context Pattern (): Ek aisa pattern jahan data ya state jo context ke taur par share ki jati hai, usay **run-time par modify nahin** kiya ja sakta. Jab koi component context data ko "badalna" chahta hai, to woh asal data ko tabdeel karne ke bajaye us data ki aik **nai copy** banata hai jismein required tabdeeliyan shamil hoti hain. Purana context joon ka toon (intact) rehta hai.

Explanation ()

Mutable aur immutable context patterns ka intekhab aapke system ki design, performance needs, concurrency concerns, aur maintainability ko bohat had tak mutasir karta hai.

Mutable Context Pattern

Kaise Kaam Karta Hai:

- Context data aam taur par aik shared object ya dictionary hota hai jise functions ya modules ko pass kiya jata hai.
- Jab koi function is context object mein koi value badalta hai, to woh asal object mein tabdeeli karta hai.
- Tamam doosre functions ya modules jo ussi object ka reference rakhte hain, unhein yeh tabdeeli fori taur par nazar aa jati hai.

Fawaid (Advantages):

- **Performance ():** Data ki copies banane ki zaroorat nahin padti, jis se memory consumption aur processing overhead kam ho sakta hai, khaas taur par bade data structures ke liye.
- **Simplicity for Small Changes ():** Choti moti tabdeeliyan karna seedha hota hai, bas value ko update kar do.

Nuqsanat (Disadvantages):

- **Side Effects ():** Code ko debug karna mushkil ho jata hai kyunkay koi bhi function context ko badal sakta hai, aur yeh jan'na mushkil ho jata hai ke kaunsa function kis waqt kya tabdeeli kar raha hai. Unexpected behavior ya "action at a distance" ho sakti hai.

- **Concurrency Issues ():** Multi-threaded ya concurrent environments mein race conditions aur data corruption ka khatra barh jata hai agar mutiple threads aik hi mutable context ko baghair proper locking ke modify karne ki koshish karein.
- **Predictability ():** System ki state (context) kab aur kahan tabdeel ho rahi hai, iski predictability kam ho jati hai.
- **Testability ():** Functions jo mutable context par depend karte hain, unhein test karna mushkil hota hai kyunkay unki output context ki ibtedai halat ke sath sath un tabdeeliyon par bhi depend karti hai jo doosre functions ne ki hain.

Istemal Ke Scenarios:

- Bahut chote, simple applications jahan concurrency aur state management ke issues nahin hain.
- Internal utility functions jahan context sirf aik temporary working space ho.

Immutable Context Pattern

Kaise Kaam Karta Hai:

- Context data ko is tarah design kiya jata hai ke usay banne ke baad badla na ja sake (maslan, Python mein tuples, frozensets, ya custom immutable objects).
- Jab kisi function ko context mein tabdeeli ki zarurat hoti hai, to woh asal context ki aik nai copy banata hai jismein tabdeel shuda values shamil hoti hain. Yeh nai copy caller ya agle component ko wapas ki jati hai.
- Purana context apni asal halat mein barqarar rehta hai.

Fawaid (Advantages):

- **Predictability ():** State transitions wazeh hoti hain kyunkay har tabdeeli aik naya context object banati hai. Aap system ki state ko waqt ke sath trace kar sakte hain.
- **No Side Effects ():** Functions context ko badal nahin sakte, sirf naya context bana sakte hain. Is se code ko samajhna aur debug karna asan ho jata hai.
- **Concurrency Safety ():** Multiple threads aik hi immutable context ko baghair race conditions ke share kar sakte hain kyunkay koi bhi thread use modify nahin kar sakta. Locking ki zaroorat nahin hoti.
- **Testability ():** Functions ko test karna asan hota hai kyunkay unki output sirf input context par depend karti hai, na ke kisi global ya shared mutable state par. Pure functions banana asan hota hai.
- **Undo/Redo Functionality ():** State ki history maintain karna asan hota hai kyunkay har state aik naya object hota hai.

Nuqsanat (Disadvantages):

- **Performance Overhead ():** Har tabdeeli ke liye naye objects banana memory aur CPU ka zayada istemal kar sakta hai, khaas taur par jab context bahut bada ho aur baar baar tabdeel ho raha ho.
- **Increased Memory Usage ():** Bar bar nai copies banane se memory usage barh sakta hai, jab tak ke copy-on-write ya structural sharing jaisi techniques istemal na ki jayen.
- **Complexity for Simple Updates ():** Aik choti si tabdeeli ke liye bhi poore context object ki nai copy banani pad sakti hai, jo code ko thoda lamba kar sakti hai.

Istemal Ke Scenarios:

- Concurrent programming, multi-threaded applications.
- Functional programming paradigms.
- Applications jahan state management crucial hai (maslan, Redux jaisay frontend frameworks).
- API request contexts jahan aap state ko across middleware share karte hain.
- Auditing aur logging systems jahan state ki history zaroori hai.

Example Code ()

Aaiye Python mein mutable aur immutable context patterns ka farq dekhte hain.

```
import copy

# --- 1. Mutable Context Pattern ---
print("--- Mutable Context Pattern ---")

def process_data_mutable(context: dict):
    """
    Mutable context ko modify karta hai.
    """
    print(f"  [process_data_mutable]: Initial context ID: {id(context)}")
    if 'counter' in context:
        context['counter'] += 1
    else:
        context['counter'] = 1
    context['last_processor'] = 'process_data_mutable'
    print(f"  [process_data_mutable]: Modified context: {context}")

def log_data_mutable(context: dict):
    """
    Mutable context ko log karta hai.
    """
    print(f"  [log_data_mutable]: Current context ID: {id(context)}")
    print(f"  [log_data_mutable]: Logging context: {context}")
    # Ye function bhi context ko badal sakta hai agar hum chahen.
    context['logged_at'] = 'now'

# Main workflow using mutable context
shared_context_mutable = {'user_id': 'abc', 'data': [1, 2, 3]}
print(f"Initial mutable context: {shared_context_mutable}, ID: {id(shared_context_mutable)}")

process_data_mutable(shared_context_mutable)
print(f"After process_data_mutable: {shared_context_mutable}, ID: {id(shared_context_mutable)}")

log_data_mutable(shared_context_mutable)
print(f"After log_data_mutable: {shared_context_mutable}, ID: {id(shared_context_mutable)}")

# Notice that the ID of shared_context_mutable remains the same,
# indicating it's the same object being modified in place.
# 'logged_at' was added by log_data_mutable, affecting process_data_mutable if it ran again.

---

# --- 2. Immutable Context Pattern ---
print("\n--- Immutable Context Pattern ---")

# Python mein dict immutable nahin hoti, isliye hum deepcopy ka istemal karenge
# taake immutable-like behavior simulate kar saken.
# Real-world scenarios mein dataclasses (frozen=True) ya libraries jaise `attrs` ya `immutable`
istemal hote hain.

def process_data_immutable(context: dict) -> dict:
    """
    Immutable context se naya context banata hai.
    """
    print(f"  [process_data_immutable]: Initial context ID: {id(context)}")
    # Naya context banao
```

```
    new_context = copy.deepcopy(context)

    if 'counter' in new_context:
        new_context['counter'] += 1
    else:
        new_context['counter'] = 1
    new_context['last_processor'] = 'process_data_immutable'
    print(f"  [process_data_immutable]: New context created: {new_context}")
    print(f"  [process_data_immutable]: New context ID: {id(new_context)}")
    return new_context

def log_data_immutable(context: dict) -> dict:
    """
    Immutable context ko log karta hai aur naya context return karta hai (optional, for
consistency).
    """
    print(f"  [log_data_immutable]: Current context ID: {id(context)}")
    print(f"  [log_data_immutable]: Logging context: {context}")

    new_context = copy.deepcopy(context)
    new_context['logged_at'] = 'now'
    print(f"  [log_data_immutable]: New context created after logging: {new_context}")
    print(f"  [log_data_immutable]: New context ID (after logging): {id(new_context)}")
    return new_context

# Main workflow using immutable context
original_context_immutable = {'user_id': 'xyz', 'data': [4, 5, 6]}
print(f"Initial immutable context: {original_context_immutable}, ID:
{id(original_context_immutable)}")

# Har step par naya context receive karein
context_after_process = process_data_immutable(original_context_immutable)
print(f"After process_data_immutable: {original_context_immutable} (Original), ID:
{id(original_context_immutable)}")
print(f"New context after process_data_immutable: {context_after_process}, ID:
{id(context_after_process)}")

context_after_log = log_data_immutable(context_after_process)
print(f"After log_data_immutable: {context_after_process} (Previous), ID:
{id(context_after_process)}")
print(f"New context after log_data_immutable: {context_after_log}, ID: {id(context_after_log)}")

# Notice that the ID of the context changes with each "modification",
# and the previous context objects remain unchanged.
```

Code Example Ki Wazahat ():

1. Mutable Context Pattern:

- `shared_context_mutable` aik Python dictionary hai, jo by default **mutable** hoti hai.
- `process_data_mutable` function `shared_context_mutable` ko sidha modify karta hai (`context['counter'] += 1`).
- `log_data_mutable` function bhi ussi `shared_context_mutable` par kaam karta hai aur usay modify kar deta hai (`context['logged_at'] = 'now'`).
- Har step ke baad `shared_context_mutable` variable ki value tabdeel ho jati hai, aur `id()` function se pata chalta hai ke object memory mein **same** hai, sirf uske contents badle hain.
- Agar `process_data_mutable` ko `log_data_mutable` ke baad chalaya jata, to usay `logged_at` field bhi milta, jo shayad uske liye ghair-mutawaqqo hota.

2. Immutable Context Pattern:

- Hum `copy.deepcopy()` ka istemal kar rahe hain takay **immutable behavior ko simulate** kar saken, kyunkay Python `dict` by default mutable hai. Asli immutable data structures (maslan `frozendict` libraries se, ya `tuple` jaisay built-ins) is copy ki zaroorat nahin rakhte.
- `process_data_immutable` function `context` ki aik **nai copy** (`new_context`) banata hai, us copy mein tabdeeli karta hai, aur phir us nai copy ko **return** karta hai.
- `log_data_immutable` function bhi yahi karta hai.
- Har step ke baad, aap dekhenge ke original context object (`original_context_immutable` ya `context_after_process`) **tabdeel nahin hota**.
- Har "modification" ke baad aik **naya object** (`new_context`) banta hai, jiska `id()` mukhtalif hota hai.
- Is se har function ke liye input context predictable rehta hai, aur race conditions ka khatra kam hota hai kyunkay koi bhi function asal context ko modify nahin kar raha.

Summary ()

Mutable context patterns mein shared data ko direct modify kiya jata hai, jis se performance behtar ho sakti hai lekin side effects, concurrency issues, aur debugging mein mushkilat paida ho sakti hain. Yeh chote ya single-threaded scenarios ke liye mufeed hain.

Iske bar-aks, **Immutable context patterns** mein data ki copies bana kar tabdeeli ki jati hai, jis se asal data hamesha mehfooz rehta hai. Yeh pattern predictibility, concurrency safety, aur testability ko behtar banata hai, lekin thoda sa performance overhead (copying ki wajah se) aur memory usage (mutiple copies ki wajah se) ho sakta hai. Yeh complex, multi-threaded, aur maintainable systems ke liye ziyada pasand kiya jata hai.

Aapki application ki zarooriyat aur complexity ke mutabiq aapko in do patterns mein se aik ka intekhab karna chahiye. Modern development mein, immutable patterns ko aksar zyada preference di jati hai unke fawaid ki wajah se, khaas taur par jab system ki state management critical ho.

36. What causes the error 'additional Properties should not be set for object types' ?

'additionalProperties should not be set for object types' Error Kyun Ata Hai?

Yeh error Pydantic ya dusre JSON Schema validators mein tab ata hai jab aap kisi **Pydantic model** ya **JSON Schema definition** mein **`additionalProperties`** field ko `True` ya `False` set karne ki koshish karte hain, jabkay woh property ya to allowed nahin hai, ya usay kisi aise tareeqay se istemal kiya ja raha hai jo standard ke mutabiq nahin hai. Aam taur par, Pydantic ke context mein iska matlab hai ke aap ne `model_config` mein ya `Field` definition mein **`additionalProperties`** ko ghalat jagah ya ghalat tareeqay se configure kiya hai.

Definition ()

`additionalProperties` JSON Schema ka aik keyword hai jo yeh control karta hai ke kya kisi object mein aisi properties (fields) ho sakti hain jo schema mein explicitly define na ki gai hon.

- **`additionalProperties:`** `True` (default): Allowed. Object mein additional fields ho sakte hain.
- **`additionalProperties:`** `False`: Not Allowed. Object mein sirf wohi fields ho sakte hain jo schema mein define kiye gaye hain. Agar koi extra field ata hai, to validation fail ho jayegi.
- `additionalProperties: {schema}`: Allowed, lekin un extra fields ko bhi diye gaye sub-schema ke mutabiq validate kiya jayega.

Error **`'additionalProperties should not be set for object types'`** is baat ki nishandahi karta hai ke aap ne `additionalProperties` ko directly us tareeqay se istemal kiya hai jahan Pydantic ya JSON Schema validator usay expect nahin karta. Pydantic mein, models (jo by default objects ko represent karte hain) `additionalProperties` ko `model_config` ke zariye handle karte hain, na ke directly field level par ya har type par.

Explanation ()

Pydantic v2 mein, **`additionalProperties`** ko control karne ka sahi tareeqa `model_config` ka istemal karna hai. Pydantic models khud-ba-khud object types banate hain. Jab aap kisi `BaseModel` ko define karte hain, to Pydantic internal taur par isay aik JSON Schema object ke taur par dekhta hai.

Yeh error aksar in suraton mein samne ata hai:

1. **Directly `Field` mein `additionalProperties` istemal karna:**
    - Aap ghalti se `Field(additionalProperties=False)` jaisa kuch likh dete hain. `additionalProperties` Field level par valid parameter nahin hai; yeh model level par apply hota hai.
    - **Ghalat:**

      Python

      ```
      from pydantic import BaseModel, Field
      class MyModel(BaseModel):
          my_field: dict = Field(additionalProperties=False) # <--- Yahan ghalti
      ```

    - **Wajah:** `additionalProperties` meta-schema keyword hai jo `object` types par apply hota hai, na ke kisi field ki value par jo ke type-hinted ho. Jab aap `my_field: dict` kehte hain, to aap bata rahe hain ke

`my_field` ki value aik dictionary hogi. Us dictionary ke andar kya properties ho sakti hain, usay control karne ke liye aapko ya to aik aur nested Pydantic model define karna hoga, ya `dict` ke andar `extra` key (Pydantic ki apni property) istemal karni hogi, ya phir `Json` type.

2. **`model_config` ko ghalat tareeqay se istemal karna:**
    - Kabhi kabhi, `model_config` mein settings ko ghalat key ke zariye specify kiya jata hai. Agar aapne `model_config` ko `extra='forbid'` (ya `extra='allow'`) set kiya hai, to Pydantic automatically `additionalProperties` ko handle karta hai JSON Schema generation ke dauran. Aapko use dobara explicitly `model_config` mein `additionalProperties` ke naam se set karne ki zarurat nahin.
    - **Ghalat:**

      Python

      ```
      from pydantic import BaseModel, ConfigDict
      class MyModel(BaseModel):
          model_config = ConfigDict(extra='forbid', additionalProperties=False) # <--- Yahan ghalti
          # Pydantic v2 mein `extra` hi `additionalProperties` ko control karta hai.
          # Dono ko aik sath set karna redundancy ya conflict create kar sakta hai.
      ```

    - **Wajah:** Pydantic v2 mein, `extra` parameter hi `additionalProperties` JSON Schema keyword ko map karta hai. `extra='forbid'` ka matlab hai `additionalProperties: False`. Agar aapne `extra='forbid'` set kiya hai, to Pydantic internal taur par isay handle karta hai. Agar aap phir `additionalProperties=False` bhi set karein, to validator ko confusion ho sakti hai.
3. **JSON Schema direct use karte waqt ghalti:**
    - Agar aap Pydantic ke baghair direct JSON Schema likh rahe hain aur `additionalProperties` ko kisi aisi property ke sath define kar rahe hain jo `object` type ki nahin hai (maslan, aik string ya number par), to bhi yeh error aa sakta hai.

Pydantic V2 mein Sahi Tareeqa: `extra` parameter

Pydantic v2 mein, `additionalProperties` ki functionality ko control karne ke liye **`model_config` mein `extra` parameter** ka istemal hota hai.

- **`extra='ignore'`**: (Default for Pydantic V2) Model mein define na kiye gaye additional fields ko **ignore** kar diya jayega. Valid inputs mein woh shamil ho sakte hain, lekin model ke `model_dump()` mein nazar nahin aayenge. JSON Schema mein iska equivalent `additionalProperties: True` hoga.
- **`extra='allow'`**: Model mein define na kiye gaye additional fields ko **allow** kiya jayega aur woh model ke `model_dump()` mein bhi shamil honge. JSON Schema mein iska equivalent `additionalProperties: True` hoga. (Ismein `ignore` aur `allow` ke darmiyan nuqta yeh hai ke `allow` model ke `dict()` output mein extra fields ko shamil karta hai, jabkay `ignore` nahin karta.)
- **`extra='forbid'`**: Model mein define na kiye gaye additional fields ko **allow nahin** kiya jayega. Agar koi extra field ata hai, to validation fail ho jayegi. JSON Schema mein iska equivalent `additionalProperties: False` hoga.

Example Code ()

Aaiye ghalat aur sahi tareeqon ko dekhte hain.

```
from pydantic import BaseModel, Field, ValidationError, ConfigDict

# --- SCENARIO 1: Incorrect use of additionalProperties directly in Field ---
print("--- SCENARIO 1: Incorrect use of additionalProperties in Field ---")

try:
    class InvalidModelField(BaseModel):
        name: str
        # Ghalat tareeqa: 'additionalProperties' Field ka valid parameter nahin hai
        data: dict = Field(additionalProperties=False)

    # Is model ko define karte hi error aa jayega (during class definition),
    # validation ke dauran nahin.
    print("InvalidModelField defined successfully (unexpected).") # This line won't be reached
except TypeError as e:
    print(f"Error defining InvalidModelField: {e}")
    print("Reason: 'additionalProperties' is not a valid argument for pydantic.fields.Field.")

# --- SCENARIO 2: Correct way to control additional properties at Model Level ---
print("\n--- SCENARIO 2: Correct way using model_config (extra) ---")

# Example 2.1: Default behavior (extra='ignore') - additional properties are ignored
class UserDefaultBehavior(BaseModel):
    id: int
    name: str

# Valid input with an extra field 'age'
data_with_extra = {"id": 1, "name": "Alice", "age": 30}
try:
    user = UserDefaultBehavior(**data_with_extra)
    print(f"UserDefaultBehavior (with 'age' field - ignored): {user.model_dump()}")
    # Output will NOT contain 'age'
except ValidationError as e:
    print(f"UserDefaultBehavior Error: {e}")


# Example 2.2: extra='allow' - additional properties are allowed and kept
class UserAllowExtra(BaseModel):
    model_config = ConfigDict(extra='allow') # extra parameters allow karein

    id: int
    name: str

try:
    user_allow = UserAllowExtra(**data_with_extra)
    print(f"UserAllowExtra (with 'age' field - allowed): {user_allow.model_dump()}")
    # Output WILL contain 'age'
except ValidationError as e:
    print(f"UserAllowExtra Error: {e}")


# Example 2.3: extra='forbid' - additional properties are NOT allowed
class UserForbidExtra(BaseModel):
    model_config = ConfigDict(extra='forbid') # extra parameters forbid karein

    id: int
    name: str

try:
```

```
    user_forbid_valid = UserForbidExtra(id=1, name="Bob")
    print(f"UserForbidExtra (valid - no extra): {user_forbid_valid.model_dump()}")

    user_forbid_invalid = UserForbidExtra(**data_with_extra) # 'age' will cause error
    print(f"UserForbidExtra (invalid - with 'age' field): {user_forbid_invalid.model_dump()}") #
This won't be reached
except ValidationError as e:
    print(f"UserForbidExtra Error (expected for 'age' field): {e}")
    # Expected Error: 'Extra inputs are not permitted'

# --- SCENARIO 3: How to validate properties within a 'dict' field (using a nested model) ---
print("\n--- SCENARIO 3: Validating properties within a 'dict' field ---")

class Address(BaseModel):
    street: str
    city: str
    zip_code: str
    # Agar Address mein bhi additional properties nahin chahiye, to yahan bhi extra='forbid'
lagana padega
    # model_config = ConfigDict(extra='forbid')

class UserWithAddress(BaseModel):
    id: int
    name: str
    address: Address # 'address' field ab aik Pydantic model hai, not a plain dict

try:
    user_with_address_valid = UserWithAddress(
        id=1,
        name="Charlie",
        address={"street": "123 Main St", "city": "Springfield", "zip_code": "12345"}
    )
    print(f"UserWithAddress (valid): {user_with_address_valid.model_dump()}")

    # Agar Address model mein extra='forbid' nahin hai to extra field ignore ho jayega
    # Agar hota to yahan error aata
    user_with_address_invalid = UserWithAddress(
        id=2,
        name="Diana",
        address={"street": "456 Elm St", "city": "Shelbyville", "zip_code": "67890",
"extra_field": "test"}
    )
    print(f"UserWithAddress (extra field in address - ignored by default):
{user_with_address_invalid.model_dump()}")

except ValidationError as e:
    print(f"UserWithAddress Error: {e}")
```

Code Example Ki Wazahat ():

1. **SCENARIO 1 (Ghalat Tareeqa):**
    - Jab aap `data: dict = Field(additionalProperties=False)` jaisa kuch likhte hain, to Python (Pydantic ke zariye) fori taur par `TypeError` generate karta hai. Iski wajah yeh hai ke `additionalProperties` `Field` ka valid argument nahin hai. Yeh keyword sirf model level par, `model_config` ke zariye istemal hota hai.
2. **SCENARIO 2 (Sahi Tareeqa):**
    - **`UserDefaultBehavior`**: Hum `extra` ko explicitly set nahin karte, isliye Pydantic V2 ka default `extra='ignore'` lagu hota hai. `data_with_extra` mein `age` field mojood hai, lekin jab `user.model_dump()` karte hain, to `age` output mein nazar nahin ata kyunkay usay `ignore` kar diya gaya hai.

- **`UserAllowExtra`**: Hum `model_config = ConfigDict(extra='allow')` set karte hain. Jab `data_with_extra` pass karte hain, to `age` field ko allow kiya jata hai aur woh `user_allow.model_dump()` output mein bhi shamil hota hai.
- **`UserForbidExtra`**: Hum `model_config = ConfigDict(extra='forbid')` set karte hain. Jab hum `user_forbid_valid` (no extra fields) banate hain to woh theek hai. Lekin jab `user_forbid_invalid` (jismein `age` extra field hai) banate hain, to **`ValidationError` aata hai jismein "Extra inputs are not permitted" message hota hai.** Yahan `additionalProperties: False` ki functionality implement ho rahi hai.

3. **SCENARIO 3 (Dictionary Ke Andar Properties Ko Validate Karna):**
   - Agar aap kisi `dict` field ke andar ki properties ko validate karna chahte hain, to sabse behtareen tareeqa yeh hai ke us `dict` ke structure ke liye aik **nested Pydantic model** bana lein (jaise `Address` model).
   - `Address` model mein aap apni pasand ke mutabiq `extra` settings laga sakte hain takay uske andar ki additional properties ko allow, forbid, ya ignore kiya ja sake.
   - `UserWithAddress` model mein `address` field ab `Address` type ka hai.

Summary ()

Error **`'additionalProperties should not be set for object types'`** is wajah se ata hai kyunkay aap `additionalProperties` JSON Schema keyword ko Pydantic mein ghalat jagah istemal kar rahe hain, maslan directly `Field` definition mein ya `model_config` mein redundant tareeqay se.

Pydantic v2 mein, object types ke liye **`additionalProperties`** ki functionality ko control karne ka sahi tareeqa **`model_config` ke andar `extra` parameter** ka istemal karna hai:

- **`extra='ignore'` (default):** Extra fields ko input se parse karte waqt ignore kar diya jayega.
- **`extra='allow'`:** Extra fields ko input se parse kiya jayega aur model ke output mein bhi shamil kiya jayega.
- **`extra='forbid'`:** Extra fields bilkul allow nahin kiye jayenge, aur agar koi ata hai to `ValidationError` raise hoga.

Agar aap kisi field ke andar ki dictionary ki properties ko validate karna chahte hain, to uske liye **nested Pydantic models** banana sabse behtar practice hai.

37. When should you use non-strictschemas over strictschemas ?

Non-Strict Schemas Aur Strict Schemas Kab Istemal Karne Chahiye?

Pydantic mein **strict** aur **non-strict** schemas ke darmiyan intekhab aapke data sources ki nature, aapke application ki robustness requirements, aur aapke control ki had par munhasir hota hai. Yeh Pydantic ke **type coercion (qisam ki tabdeeli)** ke behavior ko control karte hain.

Definition ()

Strict Schema (): Jab aik schema **strict mode** mein hota hai (yaani `model_config = ConfigDict(strict=True)`), to Pydantic data validation mein bahut sakhti ikhtiyar karta hai. Yeh sirf un values ko qabool karta hai jo field ke type se bilkul **exact match** karti hon. Koi bhi automatic type coercion (jaise string ko integer mein badalna) nahin hoti, aur agar type match nahin karta to `ValidationError` raise hota hai.

Non-Strict Schema (): Jab aik schema **non-strict mode** mein hota hai (yaani `model_config = ConfigDict(strict=False)`), Pydantic data validation mein **ziyada liberal** hota hai. Yeh automatic type coercion ki ijazat deta hai jahan mumkin ho, maslan numeric strings ko numbers mein, ya booleans ko strings se. Agar data ko field ke type mein safely convert kiya ja sakta hai, to woh qabool kar liya jata hai. **Yeh Pydantic ka default behavior hai.**

Explanation ()

Aaiye dekhte hain ke kab kaunsa schema mode behtar hai:

Jab Non-Strict Schemas Istemal Karne Chahiye (`strict=False`)

Non-strict mode Pydantic ka default hai, aur yeh aksar un scenarios ke liye behtar hota hai jahan aapka data source mukammal taur par type-consistent nahin hota ya aap thodi flexibility chahte hain.

Scenarios:

1. **External APIs Ya Third-Party Data Sources ():**
   - **Wajah:** Aksar external APIs JSON mein data bhejte waqt types ki itni sakhti nahin baratte. Misal ke taur par, aik `id` field kabhi `123` (number) ke taur par aa sakta hai aur kabhi `"123"` (string) ke taur par. Non-strict mode is tarah ki variations ko gracefully handle karta hai.
   - **Example:** Webhooks, public APIs.
2. **Legacy Systems Ke Sath Integration ():**
   - **Wajah:** Purane systems ka data format inconsistent ho sakta hai. Databases mein kabhi `INT` field mein string values store ho jati hain ya `BOOLEAN` field mein `0/1` ya `"true"/"false"` store ho jata hai. Non-strict mode in mismatches ko adjust kar sakta hai.
3. **Human Input ():**
   - **Wajah:** Jab users se direct input liya jata hai (maslan, web forms mein), to woh hamesha exact type ke mutabiq nahin hota. Aik user "100" type kar sakta hai jabke aap `int` expect kar rahe hon. Non-strict mode isay parse karne mein madad karta hai.
   - **Example:** User registration forms, search queries.

4. **Flexible Data Ingestion ():**
    - **Wajah:** Jab aap large datasets ko ingest kar rahe hon jahan kuch minor type mismatches ko bardasht karna zaroori ho, bajaye iske ke poora process fail ho jaye. Aap sirf un errors par tawajjuh dena chahte hain jo conversion ke baad bhi theek nahin ho sakte.
5. **Rapid Prototyping ():**
    - **Wajah:** Development ke ibtedai marhalon mein jahan aap fori taur par functionality test karna chahte hain aur type rigidity par ziyada tawajjuh nahin dena chahte.

Advantages of Non-Strict:

- **Robustness:** Lesser chance of validation errors due to minor type mismatches.
- **Flexibility:** Adapts to varied input data.
- **Easier Integration:** Simpler to integrate with less type-safe systems.

Jab Strict Schemas Istemal Karne Chahiye (`strict=True`)

Strict mode data integrity, type safety, aur debugging clarity ke liye behtar hota hai.

Scenarios:

1. **Internal APIs Aur Microservices ():**
    - **Wajah:** Apne hi banaye hue systems ke darmiyan, aap data types par complete control rakhte hain. Strict mode aapko yaqeeni banata hai ke data exact type ka hai, jo system ke components ke darmiyan data integrity aur consistency ko barqarar rakhta hai.
    - **Example:** Service-to-service communication, internal data pipelines.
2. **High-Integrity Data ():**
    - **Wajah:** Jab data ki type safety bahut ahem ho (maslan, financial data, scientific calculations, security-sensitive applications) aur aap nahin chahte ke koi implicit type conversion ho.
    - **Example:** Banking transactions, medical records.
3. **Debugging Aur Error Clarity ():**
    - **Wajah:** Strict mode har type mismatch par fori taur par `ValidationError` raise karta hai, jo aapko data source ke masail ko jaldi pehchanne aur theek karne mein madad deta hai. Implicit conversions ke bajaye explicit parsing logic likhne par majboor karta hai.
    - **Example:** Development environments, data validation layers.
4. **Functional Programming Paradigms ():**
    - **Wajah:** Agar aap "pure functions" ka maqsad hasil karna chahte hain jahan functions ki output sirf unke input types par depend karti hai, to strict types zaroori hain.
5. **Documentation And API Contract ():**
    - **Wajah:** Strict schemas aapke API ke contract ko aur wazeh karte hain. Jab aapka schema `strict=True` ho, to user ko maloom hota hai ke woh exact types mein data bhej raha hai ya expected types mein conversion kar raha hai.

Advantages of Strict:

- **Data Integrity:** Guarantees that data types are exactly as defined.
- **Predictability:** No unexpected type coercions.
- **Early Error Detection:** Catches type mismatches at the earliest possible stage.
- **Clarity:** Clearer API contracts and internal data flows.

Example Code ()

```
from pydantic import BaseModel, ValidationError, ConfigDict
from typing import Literal

# --- Non-Strict Schema (Default Behavior) ---
print("--- Non-Strict Schema (strict=False or omitted) ---")

class UserProfileNonStrict(BaseModel):
    # model_config = ConfigDict(strict=False) # Explicitly non-strict, but also default
    user_id: int
    name: str
    is_active: bool
    score: float

print("Non-Strict: Valid data (exact match)")
try:
    user1 = UserProfileNonStrict(user_id=1, name="Ali", is_active=True, score=99.5)
    print(f"  Parsed: {user1.model_dump()}")
except ValidationError as e:
    print(f"  Error: {e}")

print("\nNon-Strict: Type Coercion (string to int/bool/float)")
try:
    user2 = UserProfileNonStrict(user_id="2", name="Sara", is_active="True", score="85.2")
    print(f"  Parsed: {user2.model_dump()}")
    # Output: user_id=2 (int), is_active=True (bool), score=85.2 (float)
except ValidationError as e:
    print(f"  Error: {e}")

print("\nNon-Strict: Invalid Type (cannot coerce)")
try:
    user3 = UserProfileNonStrict(user_id="abc", name="Ahmed", is_active="Yes", score="invalid")
    print(f"  Parsed: {user3.model_dump()}")
except ValidationError as e:
    print(f"  Error (Expected): {e}")
    # Expected errors: user_id: value is not a valid integer, is_active: value could not be parsed
to a boolean, score: value is not a valid float


# --- Strict Schema ---
print("\n--- Strict Schema (strict=True) ---")

class UserProfileStrict(BaseModel):
    model_config = ConfigDict(strict=True) # Strict mode enable kiya

    user_id: int
    name: str
    is_active: bool
    score: float

print("Strict: Valid data (exact match)")
try:
    user4 = UserProfileStrict(user_id=1, name="Ali", is_active=True, score=99.5)
    print(f"  Parsed: {user4.model_dump()}")
except ValidationError as e:
    print(f"  Error: {e}")

print("\nStrict: Type Mismatch (string for int/bool/float) - EXPECTED TO FAIL")
try:
    user5 = UserProfileStrict(user_id="2", name="Sara", is_active="True", score="85.2")
```

```
    print(f"  Parsed: {user5.model_dump()}")
except ValidationError as e:
    print(f"  Error (Expected): {e}")
    # Expected errors: user_id: Input should be a valid integer, is_active: Input should be a
valid boolean, score: Input should be a valid float

print("\nStrict: Invalid Type (cannot coerce) - Fails more strictly")
try:
    user6 = UserProfileStrict(user_id="abc", name="Ahmed", is_active=0, score=None)
    print(f"  Parsed: {user6.model_dump()}")
except ValidationError as e:
    print(f"  Error (Expected): {e}")
    # Expected errors: user_id: Input should be a valid integer, is_active: Input should be a
valid boolean, score: Input should be a valid float
```

Code Example Ki Wazahat ():

1. **`UserProfileNonStrict` (Non-Strict Schema):**
    - Jab hum `user_id="2"` (string), `is_active="True"` (string), aur `score="85.2"` (string) provide karte hain, to Pydantic **baghair error ke inhein convert kar deta hai** `int`, `bool`, aur `float` mein. Yeh non-strict mode ki flexibility ko zahir karta hai.
    - Tahum, agar data bilkul invalid ho (`"abc"` for `int`), to Pydantic ab bhi error dega.
2. **`UserProfileStrict` (Strict Schema):**
    - Humne `model_config = ConfigDict(strict=True)` set kiya hai.
    - Jab hum `user_id="2"` (string), `is_active="True"` (string), aur `score="85.2"` (string) provide karte hain, to Pydantic **fori taur par `ValidationError` raise karta hai**. Yahan koi type coercion nahin hoti.
    - Errors ke messages batate hain ke "Input should be a valid integer", "Input should be a valid boolean", waghaira. Yeh wazeh taur par batata hai ke input data exactly type hint ke mutabiq nahin hai.

Summary ()

Non-strict schemas **(`strict=False`, which is default)** tab istemal karne chahiye jab aapke paas **mukhtalif ya inconsistent data sources** hon, jaise external APIs, legacy systems, ya direct user input. Yeh data ko **gracefully parse aur coerce** karne ki ijazat dete hain, jo robustness aur integration mein asani paida karta hai.

Strict schemas **(`strict=True`)** tab istemal karne chahiye jab aapko **data integrity aur type safety ki sakht zaroorat** ho. Yeh internal systems, high-integrity applications, aur wazeh API contracts ke liye behtar hain. Strict mode aapko **data type mismatches ko fori taur par pehchanne** aur debug karne mein madad karta hai, kyunkay yeh kisi bhi implicit type conversion ko allow nahin karta.

Aapki application ki needs ke mutabiq, aap in do modes mein se aik ka intekhab kar sakte hain, ya unhein alag-alag modules mein mix-and-match bhi kar sakte hain.

38. In a **custom tool behavior function**, what parameters does it receive ?

Custom Tool Behavior Function Mein Kya Parameters Milte Hain?

Jab aap OpenAI Assistants API ke sath **custom tool behavior functions** istemal karte hain, khaas taur par jab aap `tool_choice` parameter ko fine-tune karte hain, to aapko yeh maloom hona chahiye ke aapki tool behavior function ko kaun kaun se arguments milte hain. Yeh functions Assistant ke `Run` ko banate waqt `tool_choice` ke liye call hote hain.

Definition ()

Aik **custom tool behavior function** aik Python callable (function) hota hai jise aap OpenAI Assistants API ke `client.beta.threads.runs.create()` ya `client.beta.threads.runs.update()` call mein `tool_choice` parameter ki value ke taur par provide karte hain. Is function ka maqsad dynamic tareeqay se yeh faisla karna hai ke Assistant ko kis tool ko call karna chahiye, ya kya use koi tool call nahin karna chahiye. Yeh function Assistant ke run-time behavior ko control karne mein madad deta hai.

Explanation ()

Jab aap `tool_choice` ko aik function ke taur par provide karte hain, to OpenAI API us function ko kuch khaas parameters ke sath invoke karta hai. Yeh parameters aapko current context aur available tools ke baray mein information faraham karte hain, jiski bunyad par aap apna custom logic likh sakte hain.

Aik custom tool behavior function ko aam taur par darj zail teen parameters milte hain:

1. **`run_id` (string):**
    - **Wazahat:** Yeh current `Run` object ki ID hai jis ke liye tool choice kiya ja raha hai.
    - **Istemal:** Yeh Run ko identify karne ke liye mufeed ho sakta hai, maslan agar aap logs mein ya external systems mein is run ke baray mein information record karna chahte hain.
2. **`messages` (list of `Message` objects):**
    - **Wazahat:** Yeh `Thread` mein mojood `Message` objects ki list hoti hai jo ab tak conversation mein shamil hain. Is mein user ke messages, Assistant ke previous messages, aur tools ki outputs shamil hoti hain.
    - **Istemal:**
        - **Contextual Information:** Assistant ka current conversation context samajhne ke liye. Aap user ki latest query, ya previous turns ki history dekh sakte hain.
        - **Decision Making:** Messages ki bunyad par tool choice ka faisla karna, maslan agar user ne khaas tool se mutaliq koi sawal kiya hai.
3. **`tools` (list of `Tool` objects):**
    - **Wazahat:** Yeh un `Tool` objects ki list hoti hai jo Assistant ke sath associated hain aur use istemal ke liye available hain. Har `Tool` object mein `id`, `type` (maslan `function`), aur `function` definition (name, description, parameters) jaisi tafseelat hoti hain.
    - **Istemal:**
        - **Available Tools Ki Shinakht:** Yeh jan'ne ke liye ke kaun kaun se tools Assistant ke paas mojood hain.
        - **Tool Selection Logic:** Aap tools ki list mein iterate kar sakte hain aur unke names ya descriptions ki bunyad par faisla kar sakte hain ke kaunsa tool chunna hai.

- **Dynamic Tool Properties:** Kuch advanced scenarios mein, aap tools ki properties (maslan, descriptions) par apni logic apply kar sakte hain.

Expected Return Value:

Aapki custom function ko in teen values mein se koi aik return karni chahiye:

- `'none'`: Assistant ko koi tool call nahin karna chahiye.
- `'auto'`: Assistant ko khud-ba-khud faisla karna chahiye ke kaunsa tool call karna hai (ya nahin karna).
- `{'type': 'function', 'function': {'name': 'your_tool_name'}}`: Assistant ko khaas tor par aik specific tool (jiska naam `your_tool_name` hai) call karna chahiye.

Example Code ()

Aaiye ek misal dekhte hain jahan hum aik **custom tool behavior function** banate hain jo user ke latest message ki bunyad par tool ko dynamically select karta hai.

```
import os
import time
import json
from openai import OpenAI
from openai.types.beta.threads import Message
from openai.types.beta.threads.runs import Tool
from typing import List, Dict, Union, Literal, Any, Optional
from dotenv import load_dotenv

load_dotenv()

client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

# --- Mock Tools ---
def get_current_time(timezone: str) -> str:
    """Gets the current time for a given timezone."""
    print(f"\n[Tool Execution]: Executing get_current_time for {timezone}...")
    # Simulate different times
    if "karachi" in timezone.lower():
        return json.dumps({"time": "4:30 PM", "timezone": "PKT"})
    elif "london" in timezone.lower():
        return json.dumps({"time": "12:30 PM", "timezone": "GMT"})
    return json.dumps({"time": "N/A", "error": "Invalid timezone"})

def get_weather(city: str) -> str:
    """Gets the current weather for a given city."""
    print(f"\n[Tool Execution]: Executing get_weather for {city}...")
    time.sleep(0.2) # Simulate slight delay
    if "karachi" in city.lower():
        return json.dumps({"city": "Karachi", "weather": "Sunny", "temperature": "35C"})
    elif "london" in city.lower():
        return json.dumps({"city": "London", "weather": "Cloudy", "temperature": "15C"})
    return json.dumps({"error": "City not found", "city": city})

# Map tool names to functions
available_functions = {
    "get_current_time": get_current_time,
    "get_weather": get_weather
}
```

```
# --- Custom Tool Behavior Function ---
def my_custom_tool_chooser(
    run_id: str,
    messages: List[Message],
    tools: List[Tool]
) -> Union[Literal['none', 'auto'], Dict[str, Any]]:
    """
    This function dynamically decides which tool to use based on the latest user message.
    It receives run_id, messages (list of Message objects), and tools (list of Tool objects).
    """
    print(f"\n[Custom Tool Chooser]: Invoked for Run ID: {run_id}")

    # Get the latest user message
    latest_user_message = None
    for message in reversed(messages): # Reverse to get latest first
        if message.role == "user" and message.content:
            # Assume first content block is text
            if message.content[0].type == 'text':
                latest_user_message = message.content[0].text.value
                break

    print(f"  [Custom Tool Chooser]: Latest user message: '{latest_user_message}'")

    # Simple keyword-based logic for demonstration
    if latest_user_message:
        if "time" in latest_user_message.lower():
            print("  [Custom Tool Chooser]: 'Time' keyword found. Suggesting 'get_current_time'
tool.")
            # Check if 'get_current_time' is actually in the available tools list
            if any(tool.function.name == "get_current_time" for tool in tools if tool.type ==
'function'):
                return {"type": "function", "function": {"name": "get_current_time"}}
            else:
                print("  [Custom Tool Chooser]: 'get_current_time' tool not found. Falling back to
'auto'.")
                return "auto" # Fallback if tool isn't available

        elif "weather" in latest_user_message.lower():
            print("  [Custom Tool Chooser]: 'Weather' keyword found. Suggesting 'get_weather'
tool.")
            if any(tool.function.name == "get_weather" for tool in tools if tool.type ==
'function'):
                return {"type": "function", "function": {"name": "get_weather"}}
            else:
                print("  [Custom Tool Chooser]: 'get_weather' tool not found. Falling back to
'auto'.")
                return "auto"

    print("  [Custom Tool Chooser]: No specific keyword found. Falling back to 'auto' for LLM to
decide.")
    return "auto" # Let the LLM decide if no specific keyword is found


# --- Main Assistant Orchestration ---
async def run_assistant_with_custom_tool_chooser(user_query: str):
    # 1. Assistant create ya retrieve karein
    # (Same as previous example, ensuring assistant with tools is created)
    try:
        my_assistant_id = "asst_your_tool_chooser_assistant_id" # Change this or remove to create
new
        my_assistant = client.beta.assistants.retrieve(my_assistant_id)
        print(f"Existing Assistant istemal ho raha hai: {my_assistant.id}")
    except Exception:
        print("Naya Assistant banaya ja raha hai...")
```

```
        my_assistant = client.beta.assistants.create(
            name="Custom Chooser Assistant",
            instructions="You are a helpful assistant.",
            model="gpt-4o",
            tools=[
                {"type": "function", "function": {
                    "name": "get_current_time",
                    "description": "Get the current time for a given timezone",
                    "parameters": {
                        "type": "object", "properties": {"timezone": {"type": "string"}},
"required": ["timezone"]
                    }
                }},
                {"type": "function", "function": {
                    "name": "get_weather",
                    "description": "Get the current weather for a given city",
                    "parameters": {
                        "type": "object", "properties": {"city": {"type": "string"}}, "required":
["city"]
                    }
                }}
            ]
        )
        my_assistant_id = my_assistant.id
        print(f"Naya Assistant ban gaya ID ke sath: {my_assistant_id}")

    # 2. Thread create karein
    thread = client.beta.threads.create()
    print(f"Naya Thread ban gaya ID ke sath: {thread.id}")

    # 3. User message add karein
    print(f"\nUser: {user_query}")
    client.beta.threads.messages.create(
        thread_id=thread.id,
        role="user",
        content=user_query,
    )

    # 4. Run create karein with custom tool_choice function
    print(f"\nRun banaya ja raha hai with custom tool_choice function...")
    current_run = client.beta.threads.runs.create(
        thread_id=thread.id,
        assistant_id=my_assistant_id,
        # Yahan hum apna custom function pas karte hain
        tool_choice=my_custom_tool_chooser
    )
    print(f"Run ban gaya ID ke sath: {current_run.id}, ibtedai status: {current_run.status}")

    # 5. Run status ko poll karein aur tool actions handle karein
    while current_run.status in ["queued", "in_progress"]:
        time.sleep(0.5)
        current_run = client.beta.threads.runs.retrieve(thread_id=thread.id,
run_id=current_run.id)
        print(f"Run status: {current_run.status}")

    if current_run.status == "requires_action":
        print("\nRun requires_action state mein hai: Tool action required.")
        tool_outputs_to_submit = []
        for tool_call in current_run.required_action.submit_tool_outputs.tool_calls:
            print(f"  Tool Call ID: {tool_call.id}")
            print(f"  Function Name: {tool_call.function.name}")
            print(f"  Arguments: {tool_call.function.arguments}")

            function_name = tool_call.function.name
```

```
            function_args = json.loads(tool_call.function.arguments)

            output = ""
            if function_name in available_functions:
                output = available_functions[function_name](**function_args)
            else:
                output = json.dumps({"error": f"Function {function_name} not found"})

            tool_outputs_to_submit.append({
                "tool_call_id": tool_call.id,
                "output": output,
            })
            print(f"  Tool Output: {output}")

        print("\nTool outputs submit kiye ja rahe hain aur Run dobara shuru kiya ja raha hai...")
        current_run = client.beta.threads.runs.submit_tool_outputs(
            thread_id=thread.id,
            run_id=current_run.id,
            tool_outputs=tool_outputs_to_submit,
        )
        print(f"Run dobara shuru ho gaya status ke sath: {current_run.status}")

        # Dobara poll karein completion tak
        while current_run.status in ["queued", "in_progress"]:
            time.sleep(0.5)
            current_run = client.beta.threads.runs.retrieve(thread_id=thread.id,
run_id=current_run.id)
            print(f"Run status (after submission): {current_run.status}")

    if current_run.status == "completed":
        print("\nRun mukammal ho gaya. Assistant ka final message hasil kiya ja raha hai...")
        messages = client.beta.threads.messages.list(thread_id=thread.id, order="desc", limit=1)
        if messages.data and messages.data[0].role == "assistant" and
messages.data[0].content[0].type == "text":
            print(f"\nAssistant: {messages.data[0].content[0].text.value}")
        else:
            print("\nAssistant ne text response nahin diya ya response ghair-mutawaqqo format mein
hai.")
    else:
        print(f"\nRun ghair-mutawaqqo status par khatam hua: {current_run.status}")

# --- Run Demonstrations ---
if __name__ == "__main__":
    import asyncio

    # Scenario 1: User asks about time - custom function should choose 'get_current_time'
    asyncio.run(run_assistant_with_custom_tool_chooser("What is the current time in Karachi?"))

    print("\n" + "="*80 + "\n")

    # Scenario 2: User asks about weather - custom function should choose 'get_weather'
    asyncio.run(run_assistant_with_custom_tool_chooser("What's the weather like in London?"))

    print("\n" + "="*80 + "\n")

    # Scenario 3: User asks a general question - custom function should return 'auto'
    asyncio.run(run_assistant_with_custom_tool_chooser("Tell me a fun fact."))
```

Code Example Ki Wazahat ():

1. **`my_custom_tool_chooser` Function:**
    - Yeh woh main function hai jo custom tool behavior ko define karta hai.
    - Isay theek teen parameters milte hain:
        - `run_id: str`: Current run ki ID.
        - `messages: List[Message]`: Conversation ki history.
        - `tools: List[Tool]`: Available tools ki tafseelat.
    - Function ke andar, hum `messages` list ko reverse karte hain takay sabse naya user message hasil kar saken.
    - Aik simple `if/elif` condition ki bunyad par, hum check karte hain ke kya user ke message mein "time" ya "weather" jaisay keywords hain.
    - Agar koi keyword milta hai, aur woh tool `tools` list mein mojood hai, to hum us tool ka naam return karte hain (maslan `{"type": "function", "function": {"name": "get_current_time"}}`).
    - Agar koi khaas keyword nahin milta, ya suggested tool available nahin hai, to hum `'auto'` return karte hain, jis se LLM ko khud faisla karne ki ijazat mil jati hai.
2. **Assistant Setup:** Hum Assistant ko `get_current_time` aur `get_weather` tools ke sath configure karte hain.
3. **Run Creation With Custom Function:**
    - **Aham Nuqta:** `client.beta.threads.runs.create()` call mein, hum `tool_choice=my_custom_tool_chooser` set karte hain.
    - Jab yeh Run shuru hota hai, to OpenAI API hamare `my_custom_tool_chooser` function ko invoke karega, use `run_id`, `messages`, aur `tools` ke parameters provide karega.
    - Hamara function phir apna logic chalayega aur LLM ko batayega ke kaunsa tool chunna hai.
    - Baqi ka logic (Run status polling, tool execution, output submission) Assistant API ke standard flow ke mutabiq hai.

Is misal mein, aap dekhenge ke jab user "time" ke baray mein poochta hai, to `my_custom_tool_chooser` `get_current_time` ko force karta hai, aur jab "weather" ke baray mein poochta hai, to `get_weather` ko force karta hai. Agar koi keyword nahin milta, to Assistant `auto` mode mein jaisa woh chahega waisa karega (shayad koi tool call kare ya direct jawab de).

Summary ()

Aik **custom tool behavior function** jo `tool_choice` parameter ke liye istemal hota hai, use darj zail teen parameters milte hain:

1. **`run_id`**: Current `Run` ki unique identifier.
2. **`messages`**: `Thread` mein ab tak ke tamam `Message` objects ki list, jo conversation ka context provide karti hai.
3. **`tools`**: `Assistant` ke sath associated tamam `Tool` objects ki list, jismein har tool ki tafseelat (naam, description, parameters) shamil hoti hain.

In parameters ka istemal karte hue, aap complex aur dynamic logic likh sakte hain takay yeh faisla kiya ja sake ke LLM ko kaunsa tool call karna chahiye, ya kya use koi tool call nahin karna chahiye. Aapki function ko `'none'`, `'auto'`, ya specific tool call object return karna chahiye. Yeh approach LLM ke tool-calling behavior par behtareen control faraham karti hai.

39. What does Runner.run_streamed() return ?

`Runner.run_streamed()` Kya Return Karta Hai?

OpenAI Assistants API mein, jab aap **streaming** ka istemal karte hain, khaas taur par jab aap messages ya tool outputs ko real-time mein hasil karna chahte hain, to `Runner.run_streamed()` method ek khaas object return karta hai jo aapko events ko process karne ki ijazat deta hai.

Definition ()

`Runner.run_streamed()` method OpenAI Assistants API ke **beta** features ka hissa hai. Yeh ek **async iterator** return karta hai, yani ek aisa object jise aap `async for` loop mein istemal kar sakte hain. Jab aap is iterator ko loop karte hain, to yeh Assistant ke `Run` ke dauran hone wale mukhtalif **stream events** ko yield karta hai.

Har stream event aik khaas type ka object hota hai jo `Run` ki current state, Assistant ke messages, tool calls, aur doosre updates ki maloomat deta hai.

Explanation ()

`Runner.run_streamed()` method ko design kiya gaya hai takay aap Assistant ke run-time progress ko **real-time aur non-blocking tareeqay se monitor aur process** kar saken. Yeh traditional polling (`retrieve()` method ko baar baar call karna) se behtar hai kyunkay yeh server se fori updates hasil karta hai, jis se latency kam hoti hai aur responsiveness behtar hoti hai.

`Runner.run_streamed()` Kya Return Karta Hai (Detail Mein):

`Runner.run_streamed()` method mukhtalif types ke `StreamingEvent` objects ko yield karta hai. Yeh events Assistant ke lifecycle ke mukhtalif marhalon ko represent karte hain. Kuch ahem event types yeh hain:

1. **`thread.run.created`**: Jab naya `Run` shuru hota hai. Is mein `run` object shamil hota hai.
2. **`thread.run.queued`**: Jab `Run` queue mein hota hai.
3. **`thread.run.in_progress`**: Jab `Run` processing mein hota hai.
4. **`thread.run.step.created`**: Jab `Run` ke andar aik naya step (maslan, message creation step, tool use step) banaya jata hai. Is mein `run_step` object hota hai.
5. **`thread.run.step.in_progress`**: Jab aik `RunStep` progress mein hota hai.
6. **`thread.run.step.completed`**: Jab aik `RunStep` mukammal ho jata hai.
7. **`thread.message.created`**: Jab Assistant ya user ki taraf se koi naya message banaya jata hai. Is mein `message` object shamil hota hai.
8. **`thread.message.in_progress`**: Jab message generate ho raha hota hai.
9. **`thread.message.delta`**: Jab message ka content stream ho raha hota hai (character-by-character ya word-by-word). Yeh `delta` object text ke incremental chunks provide karta hai.
10. **`thread.message.completed`**: Jab aik message mukammal ho jata hai.
11. **`thread.tool.steps.tool_calls.created`**: Jab Assistant koi tool call generate karta hai. Is mein `tool_code_interpreter` ya `tool_function` object shamil ho sakta hai.

12. **`thread.tool.steps.tool_calls.in_progress`**: Jab tool call progress mein hota hai.
13. **`thread.tool.steps.tool_calls.output`**: Jab tool execution ki output available hoti hai.
14. **`thread.tool.steps.tool_calls.completed`**: Jab tool call mukammal ho jata hai.
15. **`thread.run.requires_action`**: Jab `Run` ko tool output submit karne ki zarurat hoti hai. Is event mein `run` object shamil hota hai jismein `required_action` details hoti hain.
16. **`thread.run.completed`**: Jab poora `Run` mukammal ho jata hai. Is mein final `run` object shamil hota hai.
17. **`thread.run.failed`**: Agar `Run` fail ho jata hai.
18. **`error`**: Jab streaming connection mein koi error hota hai.

Istemal Kaise Karein:

Aap `Runner.run_streamed()` ko `async for` loop ke sath istemal karte hain. Har iteration mein aapko aik `StreamingEvent` object milta hai. Aap us event ke `event` type ko check karte hain aur uske mutabiq action lete hain (maslan, message ka text print karna, tool call ko execute karna).

Example Code ()

Aaiye ek misal dekhte hain jahan hum **`Runner.run_streamed()`** ka istemal karte hain takay Assistant ke output ko real-time mein process kar saken.

```
import os
import time
import json
import asyncio
from openai import OpenAI
from openai.types.beta.threads import Message, Run, RunStep
from openai.types.beta.threads.runs import ToolOutput
from openai.lib.streaming import AssistantStreamManager # Stream events ko handle karne ke liye
from dotenv import load_dotenv

load_dotenv()

client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

# --- Mock Tool ---
def get_current_datetime_pakistan() -> str:
    """Gets the current date and time in Pakistan Standard Time (PKT)."""
    current_pkt_time = time.strftime("%Y-%m-%d %H:%M:%S PKT", time.localtime())
    print(f"\n[Tool Execution]: Executing get_current_datetime_pakistan()...")
    return json.dumps({"datetime": current_pkt_time})

available_functions = {
    "get_current_datetime_pakistan": get_current_datetime_pakistan,
}

# --- Async Function to Run Assistant with Streaming ---
async def run_assistant_streamed(user_query: str):
    # 1. Assistant create ya retrieve karein
    try:
        # Apne Assistant ID ko yahan dalen agar pehle se mojood hai
        # ya use comment kar dein agar naya banana hai
        my_assistant_id = "asst_your_streamed_assistant_id"
        my_assistant = client.beta.assistants.retrieve(my_assistant_id)
        print(f"Existing Assistant istemal ho raha hai: {my_assistant.id}")
    except Exception:
        print("Naya Assistant banaya ja raha hai...")
        my_assistant = client.beta.assistants.create(
            name="Streaming Demo Assistant",
            instructions="You are a helpful assistant. You can tell the current date and time in Pakistan.",
            model="gpt-4o", # Ya koi aur tool-enabled model
```

```
            tools=[
                {"type": "function", "function": {
                    "name": "get_current_datetime_pakistan",
                    "description": "Get the current date and time in Pakistan Standard Time (PKT)",
                    "parameters": {
                        "type": "object", "properties": {}, "required": []
                    }
                }},
                {"type": "code_interpreter"} # Code interpreter bhi shamil kar lete hain
            ]
        )
        my_assistant_id = my_assistant.id
        print(f"Naya Assistant ban gaya ID ke sath: {my_assistant_id}")

    # 2. Thread create karein
    print("\nNaya Thread banaya ja raha hai...")
    thread = client.beta.threads.create()
    print(f"Naya Thread ban gaya ID ke sath: {thread.id}")

    # 3. User message add karein
    print(f"\nUser: {user_query}")
    client.beta.threads.messages.create(
        thread_id=thread.id,
        role="user",
        content=user_query,
    )

    # 4. Run create karein and stream events
    print("\nRun shuru kiya ja raha hai aur events stream kiye ja rahe hain...")

    # client.beta.threads.runs.create_and_stream() is the recommended way for streaming
    # It returns an AssistantStreamManager which is an async iterator

    async with client.beta.threads.runs.create_and_stream(
        thread_id=thread.id,
        assistant_id=my_assistant_id,
    ) as stream:
        # The stream object yields various events
        async for event in stream:
            # print(f"Event Type: {event.event}") # Uncomment to see all event types

            # Handle different event types
            if event.event == "thread.run.created":
                print(f"  [RUN CREATED]: {event.data.id} - Status: {event.data.status}")

            elif event.event == "thread.run.in_progress":
                print(f"  [RUN IN PROGRESS]: {event.data.id} - Status: {event.data.status}")

            elif event.event == "thread.run.step.created":
                step: RunStep = event.data
                print(f"    [STEP CREATED]: {step.id} - Type: {step.type} - Status: {step.status}")
                if step.type == 'tool_calls':
                    print(f"      [TOOL CALLS PENDING]: {len(step.step_details.tool_calls)} tool calls
initiated.")

            elif event.event == "thread.message.created":
                message: Message = event.data
                print(f"\n  [ASSISTANT MESSAGE]: {message.id} (Role: {message.role}) - Content Status:
{message.content[0].type}")
                # Empty message at creation, content will come in delta events

            elif event.event == "thread.message.delta":
                # Yeh real-time text generate karta hai
                delta_content = event.data.delta.content[0]
                if delta_content.type == 'text':
                    if delta_content.text and delta_content.text.value:
                        print(delta_content.text.value, end="", flush=True) # Print character by character
                elif delta_content.type == 'tool_calls':
                    # Tool call details bhi yahan stream ho sakti hain
                    # print(f"    [TOOL CALL DELTA]: {delta_content.tool_calls}")
                    pass # Handled in requires_action
```

```
            elif event.event == "thread.run.requires_action":
                run: Run = event.data
                print(f"\n  [RUN REQUIRES ACTION]: {run.id} - Status: {run.status}")
                tool_outputs = []
                for tool_call in run.required_action.submit_tool_outputs.tool_calls:
                    print(f"    [TOOL CALL REQUESTED]: Name: {tool_call.function.name}, Args:
{tool_call.function.arguments}")

                    # Execute the tool
                    function_name = tool_call.function.name
                    function_args = json.loads(tool_call.function.arguments)

                    output = ""
                    if function_name in available_functions:
                        output = available_functions[function_name](**function_args)
                    else:
                        output = json.dumps({"error": f"Function {function_name} not found"})

                    tool_outputs.append(ToolOutput(tool_call_id=tool_call.id, output=output))
                    print(f"    [TOOL OUTPUT]: {output}")

                # Submit tool outputs and continue the stream
                await stream.submit_tool_outputs(tool_outputs=tool_outputs)
                print(f"  [TOOL OUTPUTS SUBMITTED]. Continuing stream...")

            elif event.event == "thread.run.completed":
                print(f"\n  [RUN COMPLETED]: {event.data.id} - Status: {event.data.status}")

            elif event.event == "thread.run.failed":
                print(f"\n  [RUN FAILED]: {event.data.id} - Error: {event.data.last_error}")

            # Aap mazeed events ko handle kar sakte hain (e.g., thread.run.cancelled,
thread.run.step.cancelled)

# --- Main Async Execution ---
async def main():
    api_key = os.getenv("OPENAI_API_KEY")
    if not api_key:
        print("Error: OPENAI_API_KEY environment variable not set.")
        return

    # Scenario 1: User asks a question that requires a tool
    await run_assistant_streamed("What is the current date and time in Pakistan?")

    print("\n" + "="*80 + "\n")

    # Scenario 2: User asks a general question (no tool needed for direct answer)
    await run_assistant_streamed("Tell me about the capital of France.")


if __name__ == "__main__":
    asyncio.run(main())
```

Code Example Ki Wazahat ():

1. **`client.beta.threads.runs.create_and_stream()`:**
    - Yeh method `Runner.run_streamed()` ka hi aik wrapper hai jo aik naya `Run` banata hai aur fori taur par streaming shuru kar deta hai.
    - Yeh aik `AssistantStreamManager` object return karta hai, jise hum `async with` statement ke andar istemal karte hain. `async with` is baat ko yaqeeni banata hai ke stream theek se close ho jaye, chahe koi error aaye.
2. **`async for event in stream:`:**

- Yeh woh ahem hissa hai jahan aap stream se events ko receive karte hain. Har iteration mein `event` variable mein aik `StreamingEvent` object hota hai.
- `event.event` attribute current event ka type batata hai (maslan, `"thread.run.created"`, `"thread.message.delta"`, `"thread.run.requires_action"`).
- `event.data` attribute mein event se mutaliq actual object hota hai (maslan, `Run` object, `Message` object, `RunStep` object).

3. **Event Handling Logic:**
   - **`thread.run.created/in_progress/completed/failed`**: `Run` ke overall lifecycle ko monitor karne ke liye.
   - **`thread.run.step.created`**: Jab Assistant koi naya step shuru karta hai (maslan, tool call ya code interpreter).
   - **`thread.message.created/delta`**:
     - `thread.message.created` jab message object ban jata hai (shuru mein khali).
     - `thread.message.delta` sabse ahem hai Assistant ke text responses ko real-time mein print karne ke liye. `delta.content[0].text.value` incremental text chunks provide karta hai. `end=""` aur `flush=True` istemal karne se text aik hi line par type hota hua nazar ata hai.
   - **`thread.run.requires_action`**: Jab Assistant ko tool outputs ki zarurat hoti hai.
     - Is event mein `run.required_action.submit_tool_outputs.tool_calls` mein tool calls ki list hoti hai.
     - Hum har tool call ko execute karte hain (`available_functions` map ka istemal karte hue).
     - **Aham Nuqta:** Tool outputs submit karne ke liye `await stream.submit_tool_outputs(tool_outputs=tool_outputs)` istemal karte hain. Yeh current `Run` ko resume karta hai aur streaming ko continue karta hai.

Is misal mein, aap dekhenge ke kaise Assistant ka response ya tool execution ki details real-time mein stream hoti hain, jabkay background mein Assistant apna kaam kar raha hota hai.

Summary ()

`Runner.run_streamed()` method (ya uski wrapper `client.beta.threads.runs.create_and_stream()`) OpenAI Assistants API mein streaming functionality ka buniyadi path hai. Yeh ek **async iterator** return karta hai.

Jab aap is iterator ko `async for` loop mein istemal karte hain, to yeh Assistant ke `Run` ke mukhtalif marhalon (creation, progress, completion, steps, message generation, tool calls) se mutaliq **`StreamingEvent` objects** ko yield karta hai. Har `StreamingEvent` mein event ka type aur us event se mutaliq data (maslan, `Run`, `Message`, `RunStep` objects) shamil hota hai.

Yeh method traditional polling ke muqable mein **real-time responsiveness, behtar user experience, aur efficient resource utilization** faraham karta hai, khaas taur par jab aapko interactive, low-latency AI applications banane hon.

40. How do you create dynamic instructions that change based on context ?

Dynamic Instructions Jo Context Ki Bunyad Par Tabdeel Hoti Hain

Dynamic instructions se murad woh hidayat ya guidelines hain jo run-time par **context** (hathon mein mojood halat) ki bunyad par generate ya modify ki jati hain. Yeh Agent (ya LLM) ke behavior, capabilities, aur tone ko us waqt ki zaroorat ke mutabiq adapt karti hain.

Definition ()

Dynamic Instructions: Aik AI model (khaas taur par aik Large Language Model ya Agent) ko di jane wali hidayat jo hard-coded nahin hotin, balkay application ke run-time environment, user input, system state, user roles, ya bahar ke data jaisi **contextual information** ki bunyad par banai ya tabdeel ki jati hain. Iska maqsad model ko ziyada relevant, accurate, aur context-aware responses ya actions karne mein madad dena hai.

Context (): Woh tamam mutaliqa maloomat jo aik khaas lamhe mein system ki halat, user ki niyyat, ya mahol ko bayan karti hain. Misalon mein shamil hain:

- **User Role/Permissions:** User admin hai ya guest?
- **Conversation History:** Pichli baatcheet kya thi?
- **External Data:** Real-time stock prices, weather, database records.
- **System State:** Kya service down hai? Maintenance mode mein hai?
- **Time/Date:** Din ka waqt, tareekh.
- **Location:** User ki geographic location.
- **User Preferences:** User ki pasandeeda zubaan, tone, ya style.

Explanation ()

Dynamic instructions banane ka maqsad LLM ko sirf aik "general-purpose" entity banaye rakhne ke bajaye, usay aik **context-aware aur adaptive system** banana hai. Jab LLM ko uske current "mahool" ki sahih maloomat di jati hain, to woh behtar aur ziyada relevant jawab de sakta hai.

Dynamic Instructions Banane Ke Tareeqe (Techniques for Creating Dynamic Instructions):

1. **Conditional Logic ():**
   - **Kaise:** `if-else` statements, `switch-case` (agar programming language support karti ho) ka istemal karte hue.
   - **Wazahat:** Sabse seedha tareeqa. Mukhtalif conditions ki bunyad par alag-alag instruction strings select ki jati hain.
   - **Misal:** Agar `user_role == "admin"`, to Assistant ko "diagnose system issues" ki hidayat do. Agar `user_role == "guest"`, to "provide basic info" ki hidayat do.
2. **String Formatting / Template Engines ():**
   - **Kaise:** F-strings (Python), `.format()` method, ya Jinja2 jaisay template engines ka istemal karte hue.
   - **Wazahat:** Aik base instruction template banaya jata hai jismein placeholders hote hain. Run-time par in placeholders ko dynamic values se fill kiya jata hai.
   - **Misal:** `You are a {user_role} assistant for a {company_name} company. Your goal is to {task_description}.`

3. **Database / Configuration Service Se Retrieval ():**
    - **Kaise:** Instructions ko database (SQL, NoSQL), remote configuration service (Firebase, AWS AppConfig), ya key-value store mein store karna.
    - **Wazahat:** Instructions ko code se alag rakha jata hai, jisse unhein run-time par update karna asan ho jata hai baghair code deploy kiye. Context (maslan, `env_name`, `feature_flag`) ki bunyad par relevant instruction fetch ki jati hai.
    - **Misal:** Production environment ke liye debugging instructions, testing environment ke liye detailed logging instructions.
4. **Chaining / Meta-Prompting ():**
    - **Kaise:** Aik LLM ko hi instructions generate karne ke liye istemal karna, jo doosre (ya wahi) LLM ke liye instructions banata hai.
    - **Wazahat:** Ziyada complex scenarios mein jahan instructions khud bhi dynamic reasoning ki paidaish hon. Aik "meta-LLM" context ko samajhta hai aur uski bunyad par "system instruction" generate karta hai.
    - **Misal:** User ki mushkil sawaal ki bunyad par, aik "instruction-generator" LLM us topic par expert tone aur guidelines generate karta hai.
5. **Tool-Based Instruction Generation ():**
    - **Kaise:** Jab Agent framework (maslan, LangChain, LlamaIndex) mein Agent tools ka istemal karta hai. Tool ki output ki bunyad par instructions ko adapt kiya jata hai.
    - **Wazahat:** Agar Agent ne koi tool call kiya aur uski output se koi specific state ya data hasil hua, to us data ko mazeed instructions mein shamil kiya jata hai.

Fawaid (Advantages):

- **Relevance:** LLM ke jawab ziyada relevant aur context-specific hote hain.
- **Adaptability:** System mukhtalif scenarios aur user needs ke mutabiq adapt kar sakta hai.
- **Personalization:** User ke role ya preferences ki bunyad par personalized experience provide karna.
- **Reduced Hallucinations:** Sahi context dene se LLM ke ghalat jawab dene ke imkanat kam hote hain.
- **Maintainability:** Instructions ko context se alag rakhne se code clean rehta hai aur instructions ko asani se update kiya ja sakta hai.

Challenges (Challenges):

- **Complexity:** Dynamic instructions ki logic likhna aur manage karna mushkil ho sakta hai.
- **Prompt Injection Risk:** Agar external input ko directly instructions mein dala jaye, to prompt injection attacks ka khatra ho sakta hai. Sanitization zaroori hai.
- **Performance Overhead:** Agar instructions complex database queries ya LLM calls se generate ho rahi hain, to latency barh sakti hai.

Example Code ()

Hum Python mein aik simple misal dekhenge jahan dynamic instructions user ke **role** aur **current time** ki bunyad par tabdeel hoti hain.

```
import os
import datetime
from openai import OpenAI
from dotenv import load_dotenv

load_dotenv()

client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
```

```
# Function to get current time and greeting based on context
def get_time_based_greeting():
    current_hour = datetime.datetime.now().hour
    if 5 <= current_hour < 12:
        return "Good morning!"
    elif 12 <= current_hour < 18:
        return "Good afternoon!"
    else:
        return "Good evening!"

# Function to generate dynamic instructions based on user role and time
def generate_dynamic_instructions(user_role: str) -> str:
    greeting = get_time_based_greeting()
    instructions = f"{greeting} You are an AI assistant. "

    if user_role.lower() == "admin":
        instructions += (
            "Your primary role is to provide detailed technical support and diagnostics for system
issues. "
            "You have access to advanced system tools (though not implemented here). Be precise,
analytical, and prioritize problem-solving."
        )
    elif user_role.lower() == "customer":
        instructions += (
            "Your primary role is to provide friendly and helpful customer support. "
            "Address user queries clearly and concisely. Focus on user satisfaction."
        )
    elif user_role.lower() == "developer":
        instructions += (
            "Your primary role is to assist developers with coding questions, API documentation,
and debugging tips. "
            "Provide code examples when relevant. Be technically accurate and concise."
        )
    else:
        instructions += (
            "You are a general-purpose helpful assistant. "
            "Provide clear and concise answers to common questions."
        )

    # Add a current time context if relevant, but often LLMs can deduce it if necessary
    # instructions += f" The current time is {datetime.datetime.now().strftime('%Y-%m-%d
%H:%M:%S')} PKT."

    return instructions

# Function to interact with the LLM
def chat_with_assistant(user_query: str, user_role: str):
    print(f"\n--- Chat as '{user_role.capitalize()}' User ---")

    dynamic_instructions = generate_dynamic_instructions(user_role)
    print(f"Generated Dynamic Instructions: {dynamic_instructions[:150]}...") # Print first 150
chars

    try:
        response = client.chat.completions.create(
            model="gpt-4o", # Ya koi aur suitable model
            messages=[
                {"role": "system", "content": dynamic_instructions},
                {"role": "user", "content": user_query}
            ],
            max_tokens=150,
            temperature=0.7
        )
        print(f"User Query: {user_query}")
```

```
        print(f"Assistant Response: {response.choices[0].message.content.strip()}")
    except Exception as e:
        print(f"Error during API call: {e}")

# --- Demonstrations ---
if __name__ == "__main__":
    if not os.getenv("OPENAI_API_KEY"):
        print("Error: OPENAI_API_KEY environment variable not set.")
    else:
        # Scenario 1: Admin User
        chat_with_assistant("My server is showing high CPU usage. What should I check first?",
"admin")

        # Scenario 2: Customer User
        chat_with_assistant("I forgot my password. How can I reset it?", "customer")

        # Scenario 3: Developer User
        chat_with_assistant("How do I implement a singleton pattern in Python?", "developer")

        # Scenario 4: Unknown Role
        chat_with_assistant("What is the capital of France?", "unknown")
```

Code Example Ki Wazahat ():

1. **`get_time_based_greeting()`:**
    - Yeh aik simple utility function hai jo current system time ki bunyad par `Good morning!`, `Good afternoon!`, ya `Good evening!` jaisa greeting return karta hai. Yeh **time-based context** ki misal hai.
2. **`generate_dynamic_instructions(user_role: str)`:**
    - Yeh hamara core function hai jo dynamic instructions banata hai.
    - Yeh `user_role` parameter leta hai.
    - Ibteda mein `get_time_based_greeting()` se greeting hasil kiya jata hai.
    - Phir, `if/elif/else` statements ka istemal karte hue, `user_role` ki bunyad par `instructions` string mein mukhtalif text add kiya jata hai.
        - "admin" user ke liye technical support ki hidayat.
        - "customer" user ke liye friendly support ki hidayat.
        - "developer" user ke liye coding assistance ki hidayat.
        - Baaki sab ke liye general-purpose hidayat.
    - Yeh `f-string` (string formatting) ka istemal karta hai takay dynamic variables ko instructions mein shamil kiya ja sake.
3. **`chat_with_assistant(user_query: str, user_role: str)`:**
    - Yeh function user query aur `user_role` leta hai.
    - Yeh `generate_dynamic_instructions()` ko call karta hai takay current context (user role aur time) ke mutabiq instructions banayi ja saken.
    - Yeh generated `dynamic_instructions` ko OpenAI API call mein `{"role": "system", "content": dynamic_instructions}` ke taur par `messages` list mein shamil karta hai.
    - LLM phir in instructions ki bunyad par jawab deta hai.

Demonstrations: Jab aap code chalayenge, to aap dekhenge ke har `user_role` ke liye LLM ka tone aur focus tabdeel ho jayega, kyunkay usay mukhtalif system instructions mil rahi hain.

- "admin" ke jawab mein system diagnostics ka zikar hoga.
- "customer" ke jawab mein password reset ke steps friendly tareeqay se honge.
- "developer" ke jawab mein Python singleton pattern ke baray mein technical tafseelat honge.

Summary ()

Dynamic instructions banane ka matlab hai ke aap AI model (khaas taur par LLMs) ko aisi hidayat dete hain jo hard-coded nahin hotin, balkay **contextual information** ki bunyad par run-time par generate ya modify ki jati hain. Isse LLM ke jawab aur actions ziyada relevant, accurate, aur adaptive ho jate hain.

Aap dynamic instructions banane ke liye mukhtalif tareeqe istemal kar sakte hain:

- **Conditional Logic (`if-else`):** Sabse seedha tareeqa.
- **String Formatting/Template Engines:** Placeholders ko dynamic values se fill karna.
- **Database/Config Retrieval:** Instructions ko external source mein store kar ke fetch karna.
- **Chaining/Meta-Prompting:** Aik LLM se doosre LLM ke liye instructions generate karwana.

In tareeqon se aap apne AI systems ko ziyada smart, flexible, aur user-centric bana sakte hain, jo user ke liye behtar tajurba faraham karte hain.