OPENAI SDK

SYEDA FARZANA SHAH GIAIC STUDENT
And
ABDUL REHMAN Prepared The SDK Notes.

Introduction

OpenAI Agents SDK

OpenAI Agents SDK aapko halkay, istimaal mein asaan package mein agentic AI apps bananay ki ijazat deta hai jismein bohot kam abstractions hain. Yeh hamaray pichlay agents ke experiments, Swarm, ka production-ready upgrade hai. Agents SDK mein bohot kam buniyadi cheezein hain:

* **Agents**: Yeh LLMs (Large Language Models) hotay hain jinhein instructions aur tools diye hotay hain.

* **Handoffs**: Yeh agents ko dosray agents ko khaas kaamon ke liye zimmedari denay ki ijazat detay hain.

* **Guardrails**: Yeh agents ke inputs ko validate karnay mein madad kartay hain.

Python ke saath mil kar, yeh buniyadi cheezein tools aur agents ke darmiyan complex rishton ko zaahir karnay ke liye kaafi powerful hain, aur aapko baghair kisi mushkil seekhnay ke asal duniya ki applications bananay ki ijazat detay hain. Iske ilawa, SDK mein built-in **tracing** hai jo aapko apnay agentic flows ko visualize aur debug karnay deta hai, saath hi unhein evaluate aur apni application ke liye models ko fine-tune karnay ki bhi ijazat deta hai.

Agents SDK kyun istemal karein?

SDK ke do buniyadi design principles hain:

* Itni features hon ke istemal karna faida-mand ho, lekin itni kam buniyadi cheezein hon ke jaldi seekh saken.
* Out of the box acha kaam kare, lekin aap theek wohi customize kar saken jo aap chahtay hain.

Yahan SDK ki khaas features hain:

* **Agent loop**: Built-in agent loop jo tools ko call karnay, results LLM ko bhejnay, aur LLM ke khatam honay tak loop karnay ko handle karta hai.

* **Python-first**: Agents ko orchestrate aur chain karnay ke liye built-in language features istemal karein, bajaye iske ke nayi abstractions seekhni padein.

* **Handoffs**: Multiple agents ke darmiyan coordination aur delegation ke liye aik powerful feature.

* **Guardrails**: Apnay agents ke mutabiq input validations aur checks ko parallel mein chalayein, agar checks fail hon toh foran rokein.
* **Function tools**: Kisi bhi Python function ko tool mein badal dein, automatic schema generation aur Pydantic-powered validation ke saath.
* **Tracing**: Built-in tracing jo aapko apnay workflows ko visualize, debug aur monitor karnay deta hai, saath hi OpenAI ke evaluation, fine-tuning aur distillation tools ka istemal bhi.

Installation

`pip install openai-agents`

Hello World Misaal

```python
from agents import Agent, Runner

agent = Agent(name="Assistant", instructions="You are a helpful assistant")

result = Runner.run_sync(agent, "Write a haiku about recursion in programming.")
print(result.final_output)
```

Aur iska output hoga:

```
# Code within the code,
# Functions calling themselves,
# Infinite loop's dance.
```

(Agar aap isay chala rahay hain, toh `OPENAI_API_KEY` environment variable set karna yaqini banayein)

```
export OPENAI_API_KEY=sk-...
```

QUICK START

Quickstart (Tez Shuruaat)

### Project aur Virtual Environment Banayen

Yeh aapko sirf aik baar karna hoga.

```bash
mkdir my_project
cd my_project
python -m venv .venv
```

Virtual Environment Activate Karein

Jab bhi aap naya terminal session shuru karein, yeh karein.

Bash

```
source .venv/bin/activate
```

Agents SDK Install Karein

Bash

```
pip install openai-agents # ya `uv add openai-agents`, waghera
```

OpenAI API Key Set Karein

Agar aapke paas nahi hai, toh OpenAI API key bananay ke liye yeh hidayat follow karein.

Bash

```
export OPENAI_API_KEY=sk-...
```

Apna Pehla Agent Banayen

Agents ko instructions, naam, aur ikhtiyari config (jaise `model_config`) ke saath define kiya jata hai.

Python

```
from agents import Agent

agent = Agent(
    name="Math Tutor",
    instructions="Aap math ke masail mein madad karte hain. Har qadam par apni wajahat bayan
karein aur misalein shamil karein.",
)
```

Kuch Aur Agents Shamil Karein

Mazeed agents ko isi tarah define kiya ja sakta hai. `handoff_descriptions` handoff routing ka faisla karnay ke liye mazeed context faraham kartay hain.

Python

```
from agents import Agent

history_tutor_agent = Agent(
    name="History Tutor",
    handoff_description="Tareekhi sawalaat ke liye mahir agent",
    instructions="Aap tareekhi sawalaat mein madad faraham karte hain. Aham waqiat aur background ko wazeh
taur par bayan karein.",
)

math_tutor_agent = Agent(
    name="Math Tutor",
    handoff_description="Math ke sawalaat ke liye mahir agent",
    instructions="Aap math ke masail mein madad karte hain. Har qadam par apni wajahat bayan karein aur
misalein shamil karein.",
)
```

Apne Handoffs Define Karein

Har agent par, aap outgoing handoff options ki inventory define kar saktay hain jinhein agent apne kaam mein tarakki karnay ke liye istemal kar sakta hai.

Python

```
triage_agent = Agent(
    name="Triage Agent",
    instructions="Aap user ke homework ke sawal ki bunyad par yeh tay karte hain ke kaun sa
agent istemal karna hai.",
    handoffs=[history_tutor_agent, math_tutor_agent]
)
```

Agent Orchestration Chalayen

Aayiye check karein ke workflow chalta hai aur triage agent dono specialist agents ke darmiyan durust tareeqay se route karta hai.

Python

```
from agents import Runner

async def main():
    result = await Runner.run(triage_agent, "France ka darul hukumat kya hai?")
    print(result.final_output)
```

Aik Guardrail Shamil Karein

Aap input ya output par chalne ke liye custom guardrails define kar sakte hain.

Python

```
from agents import GuardrailFunctionOutput, Agent, Runner
from pydantic import BaseModel

class HomeworkOutput(BaseModel):
    is_homework: bool
    reasoning: str

guardrail_agent = Agent(
    name="Guardrail check",
    instructions="Check karein ke user homework ke baray mein pooch raha hai ya nahi.",
    output_type=HomeworkOutput,
)

async def homework_guardrail(ctx, agent, input_data):
    result = await Runner.run(guardrail_agent, input_data, context=ctx.context)
    final_output = result.final_output_as(HomeworkOutput)
    return GuardrailFunctionOutput(
        output_info=final_output,
        tripwire_triggered=not final_output.is_homework,
    )
```

Sab Kuch Ikatha Karein

Aayiye sab kuch ikatha karein aur poora workflow chalayen, handoffs aur input guardrail ka istemal karte hue.

Python

```
from agents import Agent, InputGuardrail, GuardrailFunctionOutput, Runner
from pydantic import BaseModel
import asyncio

class HomeworkOutput(BaseModel):
    is_homework: bool
    reasoning: str

guardrail_agent = Agent(
    name="Guardrail check",
    instructions="Check karein ke user homework ke baray mein pooch raha hai ya nahi.",
    output_type=HomeworkOutput,
)

math_tutor_agent = Agent(
    name="Math Tutor",
    handoff_description="Math ke sawalaat ke liye mahir agent",
    instructions="Aap math ke masail mein madad karte hain. Har qadam par apni wajahat bayan
karein aur misalein shamil karein.",
)

history_tutor_agent = Agent(
    name="History Tutor",
    handoff_description="Tareekhi sawalaat ke liye mahir agent",
    instructions="Aap tareekhi sawalaat mein madad faraham karte hain. Aham waqiat aur
background ko wazeh taur par bayan karein.",
)


async def homework_guardrail(ctx, agent, input_data):
    result = await Runner.run(guardrail_agent, input_data, context=ctx.context)
    final_output = result.final_output_as(HomeworkOutput)
    return GuardrailFunctionOutput(
        output_info=final_output,
        tripwire_triggered=not final_output.is_homework,
    )

triage_agent = Agent(
    name="Triage Agent",
    instructions="Aap user ke homework ke sawal ki bunyad par yeh tay karte hain ke kaun sa
agent istemal karna hai.",
    handoffs=[history_tutor_agent, math_tutor_agent],
    input_guardrails=[
        InputGuardrail(guardrail_function=homework_guardrail),
    ],
)

async def main():
    result = await Runner.run(triage_agent, "America ke pehle saddar kaun the?")
    print(result.final_output)

    result = await Runner.run(triage_agent, "Zindagi kya hai?") # Is par guardrail trigger hoga
    print(result.final_output)

if __name__ == "__main__":
    asyncio.run(main())
```

Apni Traces Dekhein

Apne agent run ke dauran kya hua, yeh dekhne ke liye OpenAI Dashboard mein Trace viewer par jayen.

Agle Iqdamat

Mazeed complex agentic flows banana seekhein:

- Agents ko configure karnay ke baray mein seekhein.
- Agents chalane ke baray mein seekhein.
- Tools, guardrails aur models ke baray mein seekhein.

<!-- end list -->

EXAMPLE

Examples

OpenAI Agents SDK ko practice mein dekhne ke liye, **repo ke examples section** par jaa kar check karein! Aapko SDK ki mukhtalif sample implementations milengi jo alag-alag patterns aur salahiyaton ko dikhati hain.

Categories

Examples ko kai categories mein organize kiya gaya hai:

- **Agent Patterns**: Is hissay mein aap ko aam agent design patterns ki misalein milengi, jaisay **deterministic workflows** (woh workflows jin ka nateeja pehlay se pata hota hai), **agents as tools** (jab aik agent doosre agent ke liye tool ka kaam kare), aur **parallel agent execution** (jab agents aik hi waqt mein kai kaamon par kaam karein).
- **Basic**: Yeh examples SDK ki buniyadi salahiyaton ko highlight karte hain, jaisay **dynamic system prompts** (woh prompts jo halaat ke mutabiq badalte hain), **streaming outputs** (jab nateejay ahista ahista samne aayen), aur **lifecycle events** (agent ke kaam ke mukhtalif marhalay).
- **Tool Examples**: Seekhein ke **OpenAI hosted tools** jaisay **web search** aur **file search** ko kaise implement karna hai aur unhein apne agents mein kaise shamil karna hai.
- **Model Providers**: Yeh explore karein ke **non-OpenAI models** ko SDK ke saath kaise istemal karna hai.

- **Handoffs**: **Agent handoffs** ki amali misalein dekhein, jahan agents asaani se aik dosre ko kaam sonp sakte hain.
- **MCP**: Is category mein aap seekhein ge ke **MCP** (Multi-Agent Collaboration Protocol) ke saath agents kaise banayein.
- **Customer Service** aur **Research Bot**: Yeh do mazeed mukammal examples hain jo real-world applications ko wazeh karte hain:
  - **Customer Service**: Aik airline ke liye **customer service system** ki misaal.
  - **Research Bot**: Aik simple **deep research clone**.
- **Voice**: Hamare **TTS** (Text-to-Speech) aur **STT** (Speech-to-Text) models ko istemal karte huay **voice agents** ki misalein dekhein.

Yeh examples SDK ki versatility ko samajhne aur apne agent projects shuru karne ke liye behtareen tareeqa hain.

DOCUMENTATION

AGENTS

Agents Agents aapki apps ke bunyadi building block hain. Aik agent ek large language model (LLM) hota hai, jo instructions aur tools ke saath configure kiya jata hai.

Basic Configuration Aik agent ki sab se aam properties jo aap configure karein ge woh yeh hain:

- **instructions**: Isay developer message ya system prompt bhi kehte hain.
- **model**: Kaun sa LLM istemal karna hai, aur ikhtiyari `model_settings` jaisay temperature, top_p, waghera jaise model tuning parameters configure karne ke liye.
- **tools**: Woh tools jo agent apne kaamon ko anjaam denay ke liye istemal kar sakta hai.

Python

```
from agents import Agent, ModelSettings, function_tool

@function_tool
def get_weather(city: str) -> str:
    return f"The weather in {city} is sunny"

agent = Agent(
    name="Haiku agent",
    instructions="Hamesha haiku form mein jawab do", # Hamesha haiku shakal mein jawab dein
    model="o3-mini",
    tools=[get_weather],
)
```

Context Agents apni `context` type par generic hotay hain. Context ek dependency-injection tool hai: yeh ek object hai jo aap banate hain aur `Runner.run()` ko pass karte hain, jo har agent, tool, handoff waghera ko pass kiya jata hai, aur yeh agent run ke liye dependencies aur state ka ek grab bag hai. Aap kisi bhi Python object ko context ke tor par faraham kar sakte hain.

Python

```
@dataclass
class UserContext:
    uid: str
    is_pro_user: bool

    async def fetch_purchases() -> list[Purchase]:
        return ...

agent = Agent[UserContext](
    ...,
)
```

Output Types Default roop se, agents saadi text (yaani `str`) outputs paida karte hain. Agar aap chahtay hain ke agent kisi khaas qism ka output paida kare, toh aap `output_type` parameter istemal kar sakte hain. Ek aam intikhab Pydantic objects ko istemal karna hai, lekin hum kisi bhi type ko support karte hain jisay Pydantic `TypeAdapter` mein wrap kiya ja sake - dataclasses, lists, TypedDict, waghera.

Python

```
from pydantic import BaseModel
from agents import Agent


class CalendarEvent(BaseModel):
    name: str
    date: str
    participants: list[str]

agent = Agent(
    name="Calendar extractor",
    instructions="Text se calendar events nikal do", # Text se calendar ke waqiyat nikalain
    output_type=CalendarEvent,
)
```

Note Jab aap `output_type` pass karte hain, toh yeh model ko batata hai ke woh regular plain text responses ke bajaye `structured outputs` istemal kare.

Handoffs Handoffs sub-agents hotay hain jinhein agent delegate kar sakta hai. Aap handoffs ki list faraham karte hain, aur agent mutaliqa hone par unhein delegate karne ka intikhab kar sakta hai. Yeh ek taaqatwar pattern hai jo modular, specialized agents ko orchestrate karne ki ijazat deta hai jo aik hi kaam mein mahir hotay hain. Mazeed malumat `handoffs` documentation mein parhein.

Python

```
from agents import Agent

booking_agent = Agent(...)
refund_agent = Agent(...)

triage_agent = Agent(
    name="Triage agent",
    instructions=(
        "User ko unke sawalon mein madad karo." # User ko unke sawalaat mein madad karein.
        "Agar woh booking ke baray mein poochte hain, toh booking agent ko handoff karo." #
Agar woh booking ke mutaliq poochtay hain, toh booking agent ko handoff karein.
        "Agar woh refunds ke baray mein poochte hain, toh refund agent ko handoff karo." # Agar
woh refund ke mutaliq poochtay hain, toh refund agent ko handoff karein.
    ),
    handoffs=[booking_agent, refund_agent],
)
```

Dynamic Instructions Zyadatar mamlaat mein, aap agent banate waqt instructions faraham kar sakte hain. Taham, aap ek function ke zariye dynamic instructions bhi faraham kar sakte hain. Function agent aur context receive kare ga, aur prompt wapas kare ga. Regular aur `async` dono functions qabil-e-qabool hain.

Python
```
def dynamic_instructions(
    context: RunContextWrapper[UserContext], agent: Agent[UserContext]
) -> str:
    return f"User ka naam {context.context.name} hai. Unhein unke sawalon mein madad karo." # 
User ka naam {context.context.name} hai. Unhein unke sawalaat mein madad karein.


agent = Agent[UserContext](
    name="Triage agent",
    instructions=dynamic_instructions,
)
```

Lifecycle Events (Hooks) Kabhi kabhi, aap agent ke lifecycle ko observe karna chahte hain. Maslan, aap events ko log karna chahte hain, ya kuch events hone par data pre-fetch karna chahte hain. Aap `hooks` property ke zariye agent lifecycle mein hook kar sakte hain. `AgentHooks` class ko subclass karein, aur un methods ko override karein jin mein aap dilchaspi rakhte hain.

Guardrails Guardrails aapko user input par checks/validations chalane ki ijazat dete hain, agent ke chalne ke parallel mein. Maslan, aap user ke input ko relevance ke liye screen kar sakte hain. Mazeed malumat `guardrails` documentation mein parhein.

Cloning/Copying Agents Aik agent par `clone()` method istemal karke, aap aik Agent ki duplicate bana sakte hain, aur ikhtiyari tor par apni pasand ki koi bhi properties tabdeel kar sakte hain.

Python
```
pirate_agent = Agent(
    name="Pirate",
    instructions="Pirate ki tarah likho", # Pirate ki tarah likhein
    model="o3-mini",
)

robot_agent = pirate_agent.clone(
    name="Robot",
    instructions="Robot ki tarah likho", # Robot ki tarah likhein
)
```

Forcing Tool Use Tools ki list faraham karne ka matlab hamesha yeh nahi hota ke LLM tool istemal kare ga. Aap `ModelSettings.tool_choice` set kar ke tool use ko force kar sakte hain. Valid values hain:

- **auto**, jo LLM ko tool istemal karne ya na karne ka faisla karne ki ijazat deta hai.
- **required**, jo LLM ko tool istemal karne ki zaroorat deta hai (lekin yeh aqalmandi se faisla kar sakta hai ke kaun sa tool).
- **none**, jo LLM ko tool istemal na karne ki zaroorat deta hai.
- Kisi khaas string ko set karna maslan `my_tool`, jo LLM ko woh khaas tool istemal karne ki zaroorat deta hai.

Note Infinite loops ko rokne ke liye, framework tool call ke baad `tool_choice` ko automatically "auto" par reset kar deta hai. Yeh ravaiyya `agent.reset_tool_choice` ke zariye configurable hai. Infinite loop is wajah se hoti hai ke tool results LLM ko bheje jate hain, jo phir `tool_choice` ki wajah se aik aur tool call generate karta hai, ad infinitum. Agar aap chahte hain ke Agent tool call ke baad mukammal tor par ruk jaye (auto mode mein jari rakhne ke bajaye), toh aap `[Agent.tool_use_behavior="stop_on_first_tool"]` set kar sakte hain jo tool output ko mazeed LLM processing ke baghair seedha final response ke tor par istemal kare ga.

RUNNING AGENTS

Running Agents

Aap agents ko **Runner** class ke zariye chala sakte hain. Aapke paas 3 options hain:

- **Runner.run()**: Yeh async chalta hai aur **RunResult** wapas karta hai.
- **Runner.run_sync()**: Yeh ek sync method hai aur asal mein `Runner.run()` ko hi background mein chalata hai.
- **Runner.run_streamed()**: Yeh async chalta hai aur **RunResultStreaming** wapas karta hai. Yeh LLM ko streaming mode mein call karta hai, aur un events ko aap tak streams karta hai jaisay hi woh receive hotay hain.

Python
```
from agents import Agent, Runner

async def main():
    agent = Agent(name="Assistant", instructions="Aap aik madadgar assistant hain")

    result = await Runner.run(agent, "Programming mein recursion ke baray mein aik haiku likho.")
    print(result.final_output)

    # Code within the code,
    # Functions calling themselves,
    # Infinite loop's dance.
```

Mazeed jaanne ke liye, **results guide** parhein.

The Agent Loop

Jab aap **Runner** mein `run` method istemal karte hain, toh aap aik shuruaati agent aur input dete hain. Input ya toh **string** ho sakta hai (jisay user message mana jata hai), ya **input items ki list** ho sakti hai, jo OpenAI Responses API mein items hotay hain.

Runner phir aik loop chalata hai:

1. Hum maujooda agent ke liye LLM ko call karte hain, maujooda input ke saath.
2. LLM apna output paida karta hai.
    - Agar LLM **final_output** wapas karta hai, toh loop khatam ho jata hai aur hum nateeja wapas kar dete hain.
    - Agar LLM **handoff** karta hai, toh hum maujooda agent aur input ko update karte hain, aur loop ko dobara chalate hain.
    - Agar LLM **tool calls** paida karta hai, toh hum un tool calls ko chalate hain, results ko shamil karte hain, aur loop ko dobara chalate hain.
3. Agar hum **max_turns** se tajawuz kar jate hain, toh hum **MaxTurnsExceeded** exception raise karte hain.

Note: LLM output ko "final output" tab mana jata hai jab woh matlooba qisam ka text output paida kare, aur koi tool calls na hon.

Streaming

Streaming aapko LLM ke chalne ke dauran streaming events bhi receive karne ki ijazat deta hai. Ek baar stream mukammal ho jaye toh, **RunResultStreaming** mein run ke baray mein mukammal malumat shamil hongi, jin mein paida shuda tamaam naye outputs bhi shamil hain. Aap streaming events ke liye `.stream_events()` ko call kar sakte hain. Mazeed malumat ke liye, **streaming guide** parhein.

Run Config

run_config parameter aapko agent run ke liye kuch global settings configure karne deta hai:

- **model**: Har Agent ke `model` se ghafil, istemal karne ke liye aik global LLM model set karne ki ijazat deta hai.
- **model_provider**: Model names ko look up karne ke liye aik model provider, jo default OpenAI hota hai.
- **model_settings**: Agent-specific settings ko override karta hai. Masalan, aap aik global `temperature` ya `top_p` set kar sakte hain.
- **input_guardrails, output_guardrails**: Sabhi runs par shamil karne ke liye input ya output guardrails ki list.
- **handoff_input_filter**: Sabhi handoffs par lagu hone wala aik global input filter, agar handoff ke paas pehle se koi nahi hai. Input filter aapko naye agent ko bheje jane walay inputs ko edit karne ki ijazat deta hai. Mazeed tafseelat ke liye **Handoff.input_filter** documentation dekhein.
- **tracing_disabled**: Poore run ke liye **tracing** ko disable karne ki ijazat deta hai.
- **trace_include_sensitive_data**: Configure karta hai ke traces mein mumkina tor par sensitive data shamil hoga ya nahi, jaisay LLM aur tool call inputs/outputs.
- **workflow_name, trace_id, group_id**: Run ke liye tracing workflow name, trace ID aur trace group ID set karta hai. Hum kam az kam **workflow_name** set karne ki sifarish karte hain. Group ID aik ikhtiyari field hai jo aapko mutadid runs mein traces ko link karne deta hai.
- **trace_metadata**: Tamaam traces par shamil karne ke liye metadata.

Conversations/Chat Threads

Kisi bhi run methods ko call karne ke nateejay mein aik ya aik se zyada agents chal sakte hain (aur is tarah aik ya aik se zyada LLM calls), lekin yeh chat conversation mein aik single logical turn ko zahir karta hai. Masalan:

- **User turn**: User text enter karta hai.

- **Runner run**: Pehla agent LLM ko call karta hai, tools chalata hai, doosre agent ko handoff karta hai, doosra agent mazeed tools chalata hai, aur phir output paida karta hai.

Agent run ke ikhtitam par, aap chun sakte hain ke user ko kya dikhana hai. Masalan, aap user ko agents se paida hone wala har naya item dikha sakte hain, ya sirf final output. Kisi bhi tarah, user phir aik followup question pooch sakta hai, jis soorat mein aap run method ko dobara call kar sakte hain.

Aap next turn ke inputs hasil karne ke liye base **RunResultBase.to_input_list()** method istemal kar sakte hain.

Python

```
import asyncio
from agents import Agent, Runner
from agents.trace import trace # Yeh line import statements mein shamil karna zaroori hai

async def main():
    agent = Agent(name="Assistant", instructions="Mukhtasar jawab do.") # Bohat mukhtasar jawab dein.

    # Suppose thread_id is defined elsewhere, e.g., thread_id = "some_unique_id"
    # For demonstration, let's define it here
    thread_id = "conversation_123"

    with trace(workflow_name="Conversation", group_id=thread_id):
        # First turn
        result = await Runner.run(agent, "Golden Gate Bridge kis shehar mein hai?") # Golden Gate Bridge kis sheher
mein hai?
        print(result.final_output)
        # San Francisco

        # Second turn
        new_input = result.to_input_list() + [{"role": "user", "content": "Yeh kis riyasat mein hai?"}] # Yeh kis
riyasat mein hai?
        result = await Runner.run(agent, new_input)
        print(result.final_output)
        # California

if __name__ == "__main__":
    asyncio.run(main())
```

Exceptions

SDK kuch khaas mamlaat mein exceptions raise karta hai. Mukammal list **agents.exceptions** mein hai. Aik overview ke tor par:

- **AgentsException**: SDK mein raise hone walay tamaam exceptions ki base class hai.
- **MaxTurnsExceeded**: Jab run `max_turns` se tajawuz kar jaye jo run methods ko pass ki gayi thi.
- **ModelBehaviorError**: Jab model ghair-mozoon outputs paida karta hai, maslan kharab JSON ya na-maujood tools ka istemal.
- **UserError**: Jab aap (SDK istemal karne wala shakhs) SDK istemal karte waqt ghalti karte hain.
- **InputGuardrailTripwireTriggered, OutputGuardrailTripwireTriggered**: Jab aik **guardrail** trigger hota hai.

RESULTS

Results of Running Agents

Jab aap `Runner.run` methods ko call karte hain, toh aapko ya toh **RunResult** (agar aap `run` ya `run_sync` call karte hain) ya **RunResultStreaming** (agar aap `run_streamed` call karte hain) milta hai. Yeh dono `RunResultBase` se inherit karte hain, jahan zyada tar useful maloomat maujood hoti hain.

Final Output

`final_output` property mein **aakhri chalne wale agent ka final output** hota hai. Yeh ya toh:

- Aik **string** (`str`) hoga, agar aakhri agent mein `output_type` define nahi tha.
- `last_agent.output_type` qisam ka object hoga, agar agent mein `output_type` define tha.

Note: `final_output` ki type `Any` hoti hai. Hum isay statically type nahi kar sakte handoffs ki wajah se. Agar handoffs hotay hain, toh koi bhi Agent aakhri agent ho sakta hai, isliye humein statically mukhtalif possible output types ka pata nahi hota.

Next Turn Ke Liye Inputs

Aap `result.to_input_list()` ka istemal kar sakte hain taa ke nateeje ko aik input list mein tabdeel kar sakein. Yeh list aapke asal input ko agent run ke dauran generate kiye gaye items ke saath mila deti hai. Is se aik agent run ke outputs ko lena aur unhein doosre run mein pass karna, ya usay loop mein chalana aur har baar naye user inputs shamil karna aasan ho jata hai.

Last Agent

`last_agent` property mein **aakhri chalne wala agent** shamil hota hai. Aapki application ke hisaab se, yeh aksar agli baar user ke kuch input karne par mufeed hota hai. Masalan, agar aap ke paas aik frontline triage agent hai jo kisi khaas zabaan ke agent ko handoff karta hai, toh aap aakhri agent ko store kar sakte hain aur agli baar jab user agent ko message kare toh usay dobara istemal kar sakte hain.

New Items

`new_items` property mein **run ke dauran generate hone wale naye items** shamil hotay hain. Yeh items `RunItems` hotay hain. Aik run item LLM se generate hone wale raw item ko wrap karta hai:

- **MessageOutputItem**: Batata hai ke LLM se aik message aya hai. Raw item woh generate shuda message hota hai.
- **HandoffCallItem**: Batata hai ke LLM ne handoff tool ko call kiya. Raw item LLM se tool call item hota hai.
- **HandoffOutputItem**: Batata hai ke handoff hua hai. Raw item handoff tool call ka tool response hota hai. Aap item se source/target agents tak bhi rasaai hasil kar sakte hain.
- **ToolCallItem**: Batata hai ke LLM ne aik tool invoke kiya.

- **ToolCallOutputItem**: Batata hai ke aik tool call kiya gaya tha. Raw item tool response hota hai. Aap item se tool output tak bhi rasaai hasil kar sakte hain.
- **ReasoningItem**: LLM se reasoning item batata hai. Raw item generate shuda reasoning hota hai.

Doosri Maloomat

- **Guardrail Results**: `input_guardrail_results` aur `output_guardrail_results` properties mein guardrails ke nateeje shamil hotay hain, agar koi hon. Guardrail results mein kabhi kabhi mufeed maloomat ho sakti hain jinhein aap log ya store karna chahte hain, isliye hum unhein aapke liye available rakhte hain.
- **Raw Responses**: `raw_responses` property mein LLM se generate hone wale `ModelResponses` shamil hotay hain.
- **Original Input**: `input` property mein woh asal input shamil hota hai jo aapne `run` method ko faraham kiya tha. Zyada tar mamlaat mein aapko iski zaroorat nahi padegi, lekin agar aapko zaroorat ho toh yeh dastiyab hai.

STREAMING

Streaming aapko agent run ke dauran hone wale updates ko subscribe karne ki ijazat deta hai. Yeh end-user ko progress updates aur partial responses dikhane ke liye mufeed ho sakta hai.

Stream karne ke liye, aap **Runner.run_streamed()** ko call kar sakte hain, jo aapko **RunResultStreaming** dega. **result.stream_events()** ko call karne se aapko **StreamEvent** objects ka async stream milta hai, jin ki tafseel neechay di gayi hai.

Raw Response Events

RawResponsesStreamEvent woh raw events hotay hain jo seedhay LLM se pass kiye jate hain. Yeh OpenAI Responses API format mein hotay hain, jis ka matlab hai ke har event ki aik type hoti hai (jaisay `response.created`, `response.output_text.delta`, etc.) aur data hota hai. Yeh events is soorat mein mufeed hotay hain jab aap response messages ko user ko fori tor par stream karna chahte hon jaisay hi woh generate hon.

Masalan, yeh LLM se generate hone wala text token-by-token output karega:

Python

```
import asyncio
from openai.types.responses import ResponseTextDeltaEvent
from agents import Agent, Runner

async def main():
    agent = Agent(
        name="Joker",
        instructions="Aap aik madadgar assistant hain.",
    )
    result = Runner.run_streamed(agent, input="Meherbani kar ke 5 jokes sunao.")
    async for event in result.stream_events():
        if event.type == "raw_response_event" and isinstance(event.data, ResponseTextDeltaEvent):
            print(event.data.delta, end="", flush=True)


if __name__ == "__main__":
    asyncio.run(main())
```

Run Item Events aur Agent Events

RunItemStreamEvents high-level events hotay hain. Yeh aapko batate hain ke aik item mukammal tor par generate ho gaya hai. Yeh aapko "message generated", "tool ran", waghera ki satah par progress updates push karne ki ijazat deta hai, bajaye har token ke. Isi tarah, **AgentUpdatedStreamEvent** aapko updates deta hai jab current agent badalta hai (masalan, handoff ke nateejay mein).

Masalan, yeh raw events ko nazar انداز karega aur user ko updates stream karega:

Python

```
import asyncio
import random
from agents import Agent, ItemHelpers, Runner, function_tool

@function_tool
def how_many_jokes() -> int:
    return random.randint(1, 10)


async def main():
    agent = Agent(
        name="Joker",
        instructions="Pehle `how_many_jokes` tool ko call karo, phir utne jokes sunao.",
        tools=[how_many_jokes],
    )

    result = Runner.run_streamed(
        agent,
        input="Hello",
    )
    print("=== Run shuru ho raha hai ===")

    async for event in result.stream_events():
        # Hum raw responses event deltas ko nazar انداز karein ge
        if event.type == "raw_response_event":
            continue
        # Jab agent update ho, toh print karein
        elif event.type == "agent_updated_stream_event":
            print(f"Agent update hua: {event.new_agent.name}")
            continue
        # Jab items generate hon, toh unhein print karein
        elif event.type == "run_item_stream_event":
            if event.item.type == "tool_call_item":
                print("-- Tool call kiya gaya")
            elif event.item.type == "tool_call_output_item":
                print(f"-- Tool output: {event.item.output}")
            elif event.item.type == "message_output_item":
                print(f"-- Message output:\n {ItemHelpers.text_message_output(event.item)}")
            else:
                pass  # Doosri event types ko nazar انداز karein

    print("=== Run mukammal ho gaya ===")


if __name__ == "__main__":
    asyncio.run(main())
```

REPL UTILITY

REPL Utility (REPL Upyogita)

SDK (Software Development Kit) aapko tez interactive testing ke liye **run_demo_loop** provide karta hai.

Python

```
import asyncio
from agents import Agent, run_demo_loop

async def main() -> None:
    agent = Agent(name="Assistant", instructions="Aap aik madadgar assistant hain.")
    await run_demo_loop(agent)

if __name__ == "__main__":
    asyncio.run(main())
```

run_demo_loop aik loop mein user input ke liye prompt karta hai, aur conversation history ko turns ke darmiyan qaim rakhta hai. Default roop se, yeh model output ko jaisay hi paida hota hai, stream karta hai. Loop se bahar nikalne ke liye **quit** ya **exit** type karein (ya **Ctrl-D** dabayein).

TOOLS

Tools agents ko actions lene ki ijazat dete hain: jaisay data fetch karna, code run karna, external APIs ko call karna, aur yahan tak ke computer istemal karna. Agent SDK mein tools ki teen classes hain:

- **Hosted Tools**: Yeh LLM servers par AI models ke saath chalte hain. OpenAI retrieval, web search, aur computer use ko hosted tools ke tor par pesh karta hai.
- **Function Calling**: Yeh aapko kisi bhi Python function ko tool ke tor par istemal karne ki ijazat deta hai.

- **Agents as Tools**: Yeh aapko aik agent ko tool ke tor par istemal karne ki ijazat deta hai, jis se Agents dosre agents ko unhein handoff kiye baghair call kar sakte hain.

Hosted Tools

OpenAI `OpenAIResponsesModel` istemal karte waqt kuch built-in tools faraham karta hai:

- **WebSearchTool**: Yeh aik agent ko web search karne deta hai.
- **FileSearchTool**: Aapke OpenAI Vector Stores se maloomat nikalne ki ijazat deta hai.
- **ComputerTool**: Computer use ke tasks ko automate karne ki ijazat deta hai.
- **CodeInterpreterTool**: LLM ko sandboxed environment mein code execute karne deta hai.
- **HostedMCPTool**: Remote MCP server ke tools ko model ke liye expose karta hai.
- **ImageGenerationTool**: Prompt se images generate karta hai.
- **LocalShellTool**: Aapki machine par shell commands chalata hai.

Python

```python
from agents import Agent, FileSearchTool, Runner, WebSearchTool

agent = Agent(
    name="Assistant",
    tools=[
        WebSearchTool(),
        FileSearchTool(
            max_num_results=3,
            vector_store_ids=["VECTOR_STORE_ID"], # Apni Vector Store ID yahan daalein
        ),
    ],
)

async def main():
    # Example query for the agent
    result = await Runner.run(agent, "San Francisco mein aaj ka mausam aur meri pasand ko madd-e-nazar rakhte hue mujhe kis coffee shop mein jana chahiye?")
    print(result.final_output)
```

Function Tools

Aap kisi bhi Python function ko tool ke tor par istemal kar sakte hain. Agents SDK tool ko automatically setup karega:

- Tool ka naam Python function ka naam hoga (ya aap naam faraham kar sakte hain).
- Tool ki description function ke docstring se li jayegi (ya aap description faraham kar sakte hain).
- Function ke arguments se input ke liye schema automatically banaya jata hai.
- Har input ke liye descriptions function ke docstring se liye jate hain, jab tak ke isay disable na kiya jaye.

Hum Python ke `inspect` module ka istemal function signature ko extract karne ke liye karte hain, is ke saath `griffe` docstrings ko parse karne ke liye aur `pydantic` schema banane ke liye istemal hota hai.

Python

```
import json
from typing_extensions import TypedDict, Any
from agents import Agent, FunctionTool, RunContextWrapper, function_tool

class Location(TypedDict):
    lat: float
    long: float

@function_tool
async def fetch_weather(location: Location) -> str:
    """Kisi di gayi location ka mausam maloom karein.

    Args:
        location: Woh location jahan ka mausam maloom karna hai.
    """
    # Asal zindagi mein, hum weather API se mausam maloom karein ge
    return "sunny"

@function_tool(name_override="fetch_data")
def read_file(ctx: RunContextWrapper[Any], path: str, directory: str | None = None) -> str:
    """Aik file ke contents ko parhein.

    Args:
        path: Woh file ka path jise parhna hai.
        directory: Woh directory jahan se file parhni hai.
    """
    # Asal zindagi mein, hum file system se file parhein ge
    return "<file contents>"

agent = Agent(
    name="Assistant",
    tools=[fetch_weather, read_file],
)

for tool in agent.tools:
    if isinstance(tool, FunctionTool):
        print(tool.name)
        print(tool.description)
        print(json.dumps(tool.params_json_schema, indent=2))
        print()
```

Custom Function Tools

Kabhi kabhi, aap Python function ko tool ke tor par istemal nahi karna chahte. Aap seedha `FunctionTool` bana sakte hain agar aap pasand karein. Aapko faraham karna hoga:

- `name`

- `description`
- `params_json_schema`, jo arguments ke liye JSON schema hai.
- `on_invoke_tool`, jo aik async function hai jo context aur arguments ko JSON string ke tor par receive karta hai, aur tool output ko string ke tor par wapas karna zaroori hai.

Python

```python
from typing import Any
from pydantic import BaseModel
from agents import RunContextWrapper, FunctionTool

def do_some_work(data: str) -> str:
    return "done"

class FunctionArgs(BaseModel):
    username: str
    age: int

async def run_function(ctx: RunContextWrapper[Any], args: str) -> str:
    parsed = FunctionArgs.model_validate_json(args)
    return do_some_work(data=f"{parsed.username} is {parsed.age} years old")

tool = FunctionTool(
    name="process_user",
    description="User ke extract kiye gaye data ko process karta hai",
    params_json_schema=FunctionArgs.model_json_schema(),
    on_invoke_tool=run_function,
)
```

Automatic Argument aur Docstring Parsing

Jaisay ke pehle zikr kiya gaya hai, hum automatically function signature ko parse karte hain taa ke tool ke liye schema ko extract kar sakein, aur hum docstring ko parse karte hain taa ke tool aur individual arguments ke liye descriptions ko extract kar sakein. Is baray mein kuch notes:

- Signature parsing `inspect` module ke zariye kiya jata hai. Hum type annotations ko arguments ke types ko samajhne ke liye istemal karte hain, aur dynamically aik Pydantic model banate hain jo overall schema ko represent karta hai. Yeh zyada tar types ko support karta hai, jin mein Python primitives, Pydantic models, TypedDicts, aur mazeed shamil hain.
- Hum `griffe` docstrings ko parse karne ke liye istemal karte hain. Supported docstring formats `google`, `sphinx` aur `numpy` hain. Hum docstring format ko automatically detect karne ki koshish karte hain, lekin yeh best-effort hai aur aap `function_tool` ko call karte waqt isay explicitly set kar sakte hain. Aap `use_docstring_info` ko `False` set kar ke docstring parsing ko disable bhi kar sakte hain.
- Schema extraction ke liye code `agents.function_schema` mein maujood hai.

Agents as Tools

Kuch workflows mein, aap aik markazi agent chahenge jo specialized agents ke network ko orchestrate kare, bajaye iske ke control handoff kare. Aap agents ko tools ke tor par model kar ke yeh kar sakte hain.

Python

```
from agents import Agent, Runner
import asyncio

spanish_agent = Agent(
    name="Spanish agent",
    instructions="Aap user ke message ko Spanish mein tarjuma karte hain",
)

french_agent = Agent(
    name="French agent",
    instructions="Aap user ke message ko French mein tarjuma karte hain",
)

orchestrator_agent = Agent(
    name="orchestrator_agent",
    instructions=(
        "Aap aik translation agent hain. Aap di gayi tools ko tarjuma karne ke liye istemal
karte hain."
        "Agar multiple translations ke liye kaha jaye, toh aap mutaliqa tools ko call karte
hain."
    ),
    tools=[
        spanish_agent.as_tool(
            tool_name="translate_to_spanish",
            tool_description="User ke message ko Spanish mein tarjuma karein",
        ),
        french_agent.as_tool(
            tool_name="translate_to_french",
            tool_description="User ke message ko French mein tarjuma karein",
        ),
    ],
)

async def main():
    result = await Runner.run(orchestrator_agent, input="Spanish mein 'Hello, how are you?'
kaho.")
    print(result.final_output)
```

Customizing Tool-Agents

`agent.as_tool` function aik convenience method hai jo agent ko tool mein tabdeel karna aasan banata hai. Taham, yeh tamaam configuration ko support nahi karta; masalan, aap `max_turns` set nahi kar sakte. Advanced use cases ke liye, apni tool implementation mein seedha `Runner.run` istemal karein:

Python

```
@function_tool
```

```
async def run_my_agent() -> str:
    """Aik tool jo agent ko custom configs ke saath chalata hai"""

    agent = Agent(name="My agent", instructions="...")

    result = await Runner.run(
        agent,
        input="...",
        max_turns=5,
        run_config=...
    )

    return str(result.final_output)
```

Custom Output Extraction

Kuch suraton mein, aap markazi agent ko wapas karne se pehle tool-agents ke output ko tabdeel karna chahenge. Yeh mufeed ho sakta hai agar aap chahte hain:

- Sub-agent ki chat history se aik khaas maloomat (masalan, aik JSON payload) extract karna.
- Agent ke final answer ko convert ya reformat karna (masalan, Markdown ko plain text ya CSV mein tabdeel karna).
- Output ko validate karna ya fallback value faraham karna jab agent ka response ghayab ya kharab ho.

Aap `as_tool` method ko `custom_output_extractor` argument faraham kar ke yeh kar sakte hain:

Python

```
from agents import RunResult, ToolCallOutputItem

async def extract_json_payload(run_result: RunResult) -> str:
    # Agent ke outputs ko ulte tarteeb mein scan karein jab tak hamein tool call se JSON-jaisa
message na mil jaye.
    for item in reversed(run_result.new_items):
        if isinstance(item, ToolCallOutputItem) and item.output.strip().startswith("{"):
            return item.output.strip()
    # Agar kuch nahi mila toh empty JSON object par fallback karein
    return "{}"


# Suppose 'data_agent' is an existing agent definition
# json_tool = data_agent.as_tool(
#     tool_name="get_data_json",
#     tool_description="Data agent ko chalayein aur sirf iska JSON payload wapas karein",
#     custom_output_extractor=extract_json_payload,
# )
```

Handling Errors in Function Tools

Jab aap `@function_tool` ke zariye aik function tool banate hain, toh aap `failure_error_function` pass kar sakte hain. Yeh aik function hai jo tool call crash hone ki soorat mein LLM ko error response faraham karta hai.

- By default (yaani agar aap kuch bhi pass nahi karte), yeh `default_tool_error_function` chalata hai jo LLM ko batata hai ke error ho gaya hai.
- Agar aap apna error function pass karte hain, toh yeh usay chalata hai, aur response LLM ko bhejta hai.
- Agar aap explicitly `None` pass karte hain, toh koi bhi tool call errors aapke handle karne ke liye dobara raise kiye jayenge. Yeh `ModelBehaviorError` ho sakta hai agar model ne invalid JSON paida kiya, ya `UserError` agar aapka code crash ho gaya, waghera.

- Agar aap manually `FunctionTool` object bana rahe hain, toh aapko `on_invoke_tool` function ke andar errors handle karna zaroori hai.

Model context protocol (MCP)

Model Context Protocol (MCP)

Model context protocol (jisay MCP bhi kehte hain) LLM ko tools aur context faraham karne ka aik tareeqa hai. MCP docs se:

"MCP aik open protocol hai jo applications ke LLMs ko context faraham karne ka standard banata hai. MCP ko AI applications ke liye USB-C port ki tarah samjhiye. Bilkul isi tarah jaise USB-C aapke devices ko mukhtalif peripherals aur accessories se jorne ka aik standard tareeqa faraham karta hai, MCP AI models ko mukhtalif data sources aur tools se jorne ka aik standard tareeqa faraham karta hai."

Agents SDK mein MCP ki support maujood hai. Yeh aapko mukhtalif MCP servers ko apne Agents ko tools faraham karne ke liye istemal karne ki ijazat deta hai.

MCP Servers

Filhal, MCP spec mein teen qism ke servers ki tareef ki gayi hai, unke istemal hone wale transport mechanism ki buniyad par:

- **stdio servers** aapki application ke sub-process ke tor par chalte hain. Aap unhein "locally" chalte hue samajh sakte hain.
- **HTTP over SSE servers** remotely chalte hain. Aap unhein URL ke zariye connect karte hain.
- **Streamable HTTP servers** remotely chalte hain, MCP spec mein tareef kiye gaye Streamable HTTP transport ka istemal karte hue.

Aap in servers se connect karne ke liye **MCPServerStdio**, **MCPServerSse**, aur **MCPServerStreamableHttp** classes ka istemal kar sakte hain. Masalan, aap **official MCP filesystem server** ko is tarah istemal karein ge:

Python

```
async with MCPServerStdio(
    params={
        "command": "npx",
        "args": ["-y", "@modelcontextprotocol/server-filesystem", samples_dir],
    }
) as server:
    tools = await server.list_tools()
```

MCP Servers ka Istemal

MCP servers ko Agents mein shamil kiya ja sakta hai. Agents SDK har baar Agent ke chalne par MCP servers par **list_tools()** call karega. Yeh LLM ko MCP server ke tools se waqif karta hai. Jab LLM kisi MCP server se koi tool call karta hai, toh SDK us server par **call_tool()** call karta hai.

Python

```
agent=Agent(
    name="Assistant",
    instructions="Task haasil karne ke liye tools istemal karo", # Task haasil karne ke liye
tools istemal karein
    mcp_servers=[mcp_server_1, mcp_server_2]
)
```

Caching

Har baar jab aik Agent chalta hai, toh woh MCP server par **list_tools()** call karta hai. Yeh latency ka sabab ban sakta hai, khaas tor par agar server remote server ho. Tools ki list ko automatically cache karne ke liye, aap **MCPServerStdio**, **MCPServerSse**, aur **MCPServerStreamableHttp** ko `cache_tools_list=True` pass kar sakte hain. Aapko aisa tabhi karna chahiye jab aapko yaqeen ho ke tool list tabdeel nahi hogi. Agar aap cache ko invalidate karna chahte hain, toh aap servers par **invalidate_tools_cache()** call kar sakte hain.

End-to-End Examples

Mukammal chalne wali misalein **examples/mcp** mein dekhein.

Tracing

Tracing MCP operations ko automatically capture karta hai, jin mein shamil hain:

- MCP server ko tools list karne ke liye calls
- Function calls par MCP se mutaliq maloomat

Handoffs

Handoffs agents ko doosre agents ko kaam delegate karne ki ijazat dete hain. Yeh un scenarios mein khaas tor par mufeed hai jahan mukhtalif agents mukhtalif areas mein mahir hotay hain. Masalan, aik customer support app mein aise agents ho sakte hain jo khaas tor par order status, refunds, FAQs, waghera jaise kaamon ko handle karte hon.

Handoffs LLM ke liye tools ke tor par represent kiye jate hain. Lehaza, agar kisi agent ko `Refund Agent` ko handoff kiya jata hai, toh tool ka naam `transfer_to_refund_agent` hoga.

Creating a Handoff (Handoff Banana)

Sabhi agents mein aik `handoffs` parameter hota hai, jo ya toh direct aik **Agent** le sakta hai, ya aik **Handoff** object jo Handoff ko customize karta hai.

Aap Agents SDK ki faraham karda `handoff()` function ka istemal kar ke handoff bana sakte hain. Yeh function aapko woh agent specify karne ki ijazat deta hai jisay handoff karna hai, saath hi ikhtiyari overrides aur input filters bhi.

Basic Usage (Buniyadi Istemal)

Yahan aik simple handoff banane ka tareeqa hai:

Python

```python
from agents import Agent, handoff

billing_agent = Agent(name="Billing agent")
refund_agent = Agent(name="Refund agent")

triage_agent = Agent(name="Triage agent", handoffs=[billing_agent, handoff(refund_agent)])
```

Customizing Handoffs via the `handoff()` Function (Handoff() Function Ke Zariye Handoffs Ko Customize Karna)

`handoff()` function aapko cheezon ko customize karne deta hai:

- **agent**: Yeh woh agent hai jisay cheezein handoff ki jayengi.
- **tool_name_override**: Default tor par, `Handoff.default_tool_name()` function istemal hota hai, jo `transfer_to_<agent_name>` mein badal jata hai. Aap isay override kar sakte hain.
- **tool_description_override**: `Handoff.default_tool_description()` se default tool description ko override karein.
- **on_handoff**: Aik callback function jo handoff invoke hone par execute hoti hai. Yeh un cheezon ke liye mufeed hai jaisay data fetching shuru karna jaisay hi aapko pata chale ke handoff invoke ho raha hai. Yeh function agent context receive karta hai, aur ikhtiyari tor par LLM generated input bhi receive kar sakta hai. Input data `input_type` param se control hota hai.

- **input_type**: Handoff se mutawaqqay input ki type (ikhtiyari).
- **input_filter**: Yeh aapko agle agent ko milne wale input ko filter karne deta hai. Mazeed tafseelat ke liye neechay dekhein.

Python

```
from agents import Agent, handoff, RunContextWrapper

def on_handoff(ctx: RunContextWrapper[None]):
    print("Handoff call kiya gaya")

agent = Agent(name="Mera agent") # 'My agent'

handoff_obj = handoff(
    agent=agent,
    on_handoff=on_handoff,
    tool_name_override="custom_handoff_tool",
    tool_description_override="Custom description",
)
```

Handoff Inputs (Handoff Ke Inputs)

Kuch suraton mein, aap chahte hain ke LLM handoff ko call karte waqt kuch data faraham kare. Masalan, aik "Escalation agent" ko handoff ka tasavvur karein. Aap shayad chahenge ke aik wajah faraham ki jaye, taa ke aap usay log kar sakein.

Python

```
from pydantic import BaseModel
from agents import Agent, handoff, RunContextWrapper

class EscalationData(BaseModel):
    reason: str

async def on_handoff(ctx: RunContextWrapper[None], input_data: EscalationData):
    print(f"Escalation agent ko is wajah se call kiya gaya: {input_data.reason}")

agent = Agent(name="Escalation agent")

handoff_obj = handoff(
    agent=agent,
    on_handoff=on_handoff,
    input_type=EscalationData,
)
```

Input Filters (Input Filters)

Jab aik handoff hota hai, toh yeh aisa hai jaisay naya agent conversation ka charge le leta hai, aur usay poori pichli conversation history dekhne ko milti hai. Agar aap isay tabdeel karna chahte hain, toh aap `input_filter` set kar sakte hain. Aik input filter aik function hai jo maujooda input ko `HandoffInputData` ke zariye receive karta hai, aur usay aik naya `HandoffInputData` wapas karna zaroori hai.

Kuch aam patterns (masalan history se tamaam tool calls ko hatana), `agents.extensions.handoff_filters` mein aapke liye implement kiye gaye hain.

Python

```
from agents import Agent, handoff
from agents.extensions import handoff_filters

agent = Agent(name="FAQ agent")

handoff_obj = handoff(
    agent=agent,
    input_filter=handoff_filters.remove_all_tools,
)
```

Recommended Prompts (Sifarish Karda Prompts)

Yeh yakeeni banane ke liye ke LLMs handoffs ko sahih tareeqe se samjhein, hum apne agents mein handoffs ke baray mein maloomat shamil karne ki sifarish karte hain. Hamare paas `agents.extensions.handoff_prompt.RECOMMENDED_PROMPT_PREFIX` mein aik suggested prefix hai, ya aap apne prompts mein recommended data ko automatically shamil karne ke liye `agents.extensions.handoff_prompt.prompt_with_handoff_instructions` ko call kar sakte hain.

Python

```
from agents import Agent
from agents.extensions.handoff_prompt import RECOMMENDED_PROMPT_PREFIX

billing_agent = Agent(
    name="Billing agent",
    instructions=f"""{RECOMMENDED_PROMPT_PREFIX}
    <Apna baaki prompt yahan daalein>.""", # Apne prompt ka baqi hissa yahan bharein
)
```

Tracing

Agents SDK mein built-in tracing shamil hai, jo agent run ke dauran events ka aik mukammal record jama karta hai: LLM generations, tool calls, handoffs, guardrails, aur yahan tak ke custom events jo hotay hain. **Traces dashboard** ka istemal karte huay, aap development aur production mein apne workflows ko debug, visualize, aur monitor kar sakte hain.

Note: Tracing by default enabled hota hai. Tracing ko disable karne ke do tareeqay hain:

- Aap environment variable `OPENAI_AGENTS_DISABLE_TRACING=1` set kar ke globaly tracing ko disable kar sakte hain.
- Aap aik single run ke liye `agents.run.RunConfig.tracing_disabled` ko `True` set kar ke tracing ko disable kar sakte hain. OpenAI ki APIs istemal karte huay Zero Data Retention (ZDR) policy ke tehat kaam karne wali organizations ke liye, tracing dastiyab nahi hai.

Traces aur Spans

Traces aik "workflow" ke aik single end-to-end operation ko represent karte hain. Woh Spans se mil kar bante hain. Traces mein darj zail properties hoti hain:

- **workflow_name**: Yeh logical workflow ya app hai. Masalan "Code generation" ya "Customer service".
- **trace_id**: Trace ke liye aik unique ID. Agar aap pass nahi karte toh automatically generate hota hai. `trace_<32_alphanumeric>` format mein hona zaroori hai.
- **group_id**: Ikhtiyari group ID, aik hi conversation se mutadid traces ko link karne ke liye. Masalan, aap chat thread ID istemal kar sakte hain.
- **disabled**: Agar `True` ho, toh trace record nahi kiya jayega.
- **metadata**: Trace ke liye ikhtiyari metadata.

Spans woh operations represent karte hain jin ka aik shuru aur ikhtitam ka waqt hota hai. Spans mein yeh hota hai:

- `started_at` aur `ended_at` timestamps.
- `trace_id`, jo us trace ko represent karta hai jis se woh talluq rakhte hain.
- `parent_id`, jo is Span ke parent Span ki taraf ishara karta hai (agar koi ho).
- `span_data`, jo Span ke baray mein maloomat hai. Masalan, `AgentSpanData` mein Agent ke baray mein maloomat hoti hai, `GenerationSpanData` mein LLM generation ke baray mein maloomat hoti hai, waghera.

Default Tracing

Default roop se, SDK darj zail ko trace karta hai:

- Poora `Runner.{run, run_sync, run_streamed}()` aik `trace()` mein wrap hota hai.
- Har baar jab aik agent chalta hai, toh woh `agent_span()` mein wrap hota hai.
- LLM generations `generation_span()` mein wrap hoti hain.

- Function tool calls har aik `function_span()` mein wrap hota hai.
- Guardrails `guardrail_span()` mein wrap hotay hain.
- Handoffs `handoff_span()` mein wrap hotay hain.
- Audio inputs (speech-to-text) aik `transcription_span()` mein wrap hotay hain.
- Audio outputs (text-to-speech) aik `speech_span()` mein wrap hotay hain.
- Mutaliqa audio spans aik `speech_group_span()` ke tehat parent kiye ja sakte hain.

Default roop se, trace ka naam "Agent trace" hota hai. Aap yeh naam set kar sakte hain agar aap `trace` istemal karte hain, ya aap `RunConfig` ke saath naam aur doosri properties configure kar sakte hain. Iske ilawa, aap **custom trace processors** set up kar sakte hain taa ke traces ko doosri manzil (replacement ke tor par, ya secondary manzil) tak push kar sakein.

Higher Level Traces

Kabhi kabhi, aap `run()` ki mutadid calls ko aik hi trace ka hissa banana chahte hain. Aap poore code ko `trace()` mein wrap kar ke yeh kar sakte hain.

Python

```
from agents import Agent, Runner, trace
import asyncio # Add asyncio import here

async def main():
    agent = Agent(name="Joke generator", instructions="Mazahiya jokes sunao.")

    with trace("Joke workflow"):
        first_result = await Runner.run(agent, "Mujhe aik joke sunao")
        second_result = await Runner.run(agent, f"Is joke ko rate karo:
{first_result.final_output}")
        print(f"Joke: {first_result.final_output}")
        print(f"Rating: {second_result.final_output}")

if __name__ == "__main__":
    asyncio.run(main())
```

Creating Traces (Traces Banana)

Aap `trace()` function ka istemal trace banane ke liye kar sakte hain. Traces ko shuru aur khatam karna zaroori hota hai. Aapke paas aisa karne ke do options hain:

- **Sifarish karda**: `trace` ko context manager ke tor par istemal karein, yaani `with trace(...) as my_trace`. Yeh trace ko sahih waqt par automatically shuru aur khatam karega.
- Aap manually `trace.start()` aur `trace.finish()` ko bhi call kar sakte hain.

Current trace ko Python `contextvar` ke zariye track kiya jata hai. Iska matlab hai ke yeh concurrency ke saath automatically kaam karta hai. Agar aap manually aik trace ko shuru/khatam karte hain, toh aapko `mark_as_current` aur `reset_current` ko `start()/finish()` mein pass karna hoga taa ke current trace ko update kar sakein.

Creating Spans (Spans Banana)

Aap mukhtalif `*_span()` methods ka istemal span banane ke liye kar sakte hain. Aam tor par, aapko manually spans banane ki zaroorat nahi hoti. Custom span maloomat ko track karne ke liye aik `custom_span()` function dastiyab hai. Spans automatically current trace ka hissa hotay hain, aur qareeb tareen current span ke neeche nested hotay hain, jisay Python `contextvar` ke zariye track kiya jata hai.

Sensitive Data (Hassas Data)

Kuch spans mumkina tor par sensitive data capture kar sakte hain. `generation_span()` LLM generation ke inputs/outputs ko store karta hai, aur `function_span()` function calls ke inputs/outputs ko store karta hai. In mein sensitive data shamil ho sakta hai, isliye aap `RunConfig.trace_include_sensitive_data` ke zariye us data ko capture karna disable kar sakte hain. Isi tarah, Audio spans by default input aur output audio ke liye base64-encoded PCM data shamil karte hain. Aap `VoicePipelineConfig.trace_include_sensitive_audio_data` ko configure kar ke is audio data ko capture karna disable kar sakte hain.

Custom Tracing Processors (Custom Tracing Processors)

Tracing ke liye high level architecture yeh hai:

- Initialization par, hum aik global `TraceProvider` banate hain, jo traces banane ke liye zimmedar hota hai.
- Hum `TraceProvider` ko aik `BatchTraceProcessor` ke saath configure karte hain jo traces/spans ko batches mein `BackendSpanExporter` ko bhejta hai, jo spans aur traces ko OpenAI backend ko batches mein export karta hai.

Is default setup ko customize karne ke liye, traces ko mutabadil ya mazeed backends par bhejne ya exporter behavior ko modify karne ke liye, aapke paas do options hain:

- `add_trace_processor()` aapko aik **additional** trace processor shamil karne deta hai jo traces aur spans ko tayyar hone par receive karega. Yeh aapko OpenAI ke backend ko traces bhejne ke ilawa apni processing karne deta hai.
- `set_trace_processors()` aapko default processors ko apne trace processors se **replace** karne deta hai. Iska matlab hai ke traces OpenAI backend ko nahi bheje jayenge jab tak aap aik `TracingProcessor` shamil na karein jo aisa karta hai.

External Tracing Processors List (External Tracing Processors Ki Fehrist)

- Weights & Biases
- Arize-Phoenix
- Future AGI
- MLflow (self-hosted/OSS)
- MLflow (Databricks hosted)
- Braintrust
- Pydantic Logfire
- AgentOps
- Scorecard
- Keywords AI

- LangSmith
- Maxim AI
- Comet Opik
- Langfuse
- Langtrace
- Okahu-Monocle
- Galileo
- Portkey AI

Context management

Context Management (Context Ka Intezaam)

Context aik aisa lafz hai jis ke mukhtalif maani hain. Do ahem qism ke context hain jin ki aapko fikar ho sakti hai:

1. **Context available locally to your code**: Yeh woh data aur dependencies hain jin ki aapko tool functions chalne ke dauran, callbacks jaisay `on_handoff` mein, lifecycle hooks mein, waghera mein zaroorat ho sakti hai.
2. **Context available to LLMs**: Yeh woh data hai jo LLM response generate karte waqt dekhta hai.

Local Context (Local Context)

Yeh `RunContextWrapper` class aur uske andar maujood `context` property ke zariye represent kiya jata hai. Yeh is tarah kaam karta hai:

1. Aap koi bhi Python object banate hain jo aap chahte hain. Aik aam tareeqa dataclass ya Pydantic object istemal karna hai.
2. Aap us object ko mukhtalif run methods ko pass karte hain (masalan, `Runner.run(..., context=whatever)`).
3. Aapke tamaam tool calls, lifecycle hooks waghera ko aik wrapper object, `RunContextWrapper[T]`, pass kiya jayega, jahan `T` aapke context object type ko represent karta hai jise aap `wrapper.context` ke zariye access kar sakte hain.
4. Sab se ahem baat jis ka khayal rakhna hai: har agent, tool function, lifecycle waghera, aik khaas agent run ke liye aik hi **type** ka context istemal karna chahiye.

Aap context ko in cheezon ke liye istemal kar sakte hain:

- **Contextual data for your run** (masalan, username/uid ya user ke baray mein doosri maloomat).
- **Dependencies** (masalan, logger objects, data fetchers, waghera).
- **Helper functions**.

Note: Context object LLM ko nahi bheja jata. Yeh sirf aik local object hai jisay aap parh sakte hain, likh sakte hain aur us par methods call kar sakte hain.

Python

```
import asyncio
from dataclasses import dataclass

from agents import Agent, RunContextWrapper, Runner, function_tool

@dataclass
class UserInfo:
    name: str
    uid: int

@function_tool
async def fetch_user_age(wrapper: RunContextWrapper[UserInfo]) -> str:
    return f"User {wrapper.context.name} ki umar 47 saal hai" # User {wrapper.context.name} is
47 years old.

async def main():
    user_info = UserInfo(name="John", uid=123)

    agent = Agent[UserInfo](
        name="Assistant",
        tools=[fetch_user_age],
    )

    result = await Runner.run(
        starting_agent=agent,
        input="User ki umar kya hai?", # What is the age of the user?
        context=user_info,
    )

    print(result.final_output)
    # The user John is 47 years old. (User John ki umar 47 saal hai.)

if __name__ == "__main__":
    asyncio.run(main())
```

Agent/LLM Context (Agent/LLM Context)

Jab LLM ko call kiya jata hai, toh woh **sirf** conversation history se data dekh sakta hai. Iska matlab hai ke agar aap LLM ko kuch naya data faraham karna chahte hain, toh aapko usay is tarah faraham karna hoga ke woh us history mein maujood ho. Aisa karne ke kuch tareeqay hain:

- Aap isay Agent `instructions` mein shamil kar sakte hain. Isay "system prompt" ya "developer message" bhi kehte hain. System prompts static strings ho sakte hain, ya woh dynamic functions ho sakte hain jo context receive karte hain aur string output karte hain. Yeh aisi maloomat ke liye aik aam tareeqa hai jo hamesha mufeed hoti hai (masalan, user ka naam ya mojooda tareekh).
- `Runner.run` functions ko call karte waqt `input` mein shamil karein. Yeh `instructions` tareeqay se milta julta hai, lekin aapko messages ko `chain of command` mein neechay rakhne ki ijazat deta hai.
- Function tools ke zariye expose karein. Yeh `on-demand` context ke liye mufeed hai - LLM faisla karta hai ke usay kab kuch data ki zaroorat hai, aur us data ko fetch karne ke liye tool ko call kar sakta hai.
- Retrieval ya web search istemal karein. Yeh khaas tools hain jo files ya databases (retrieval) se, ya web (web search) se mutaliqa data fetch kar sakte hain. Yeh response ko mutaliqa contextual data mein "ground" karne ke liye mufeed hai.

Guardrails

Guardrails

Guardrails aapke agents ke **parallel** chalte hain, jo aapko user input ki checks aur validations karne ki ijazat dete hain. Masalan, tasawwur karein aapke paas aik agent hai jo customer requests mein madad ke liye aik bohat smart (aur is wajah se slow/expensive) model istemal karta hai. Aap nahi chahenge ke bad-niyat users model ko unke math homework mein madad karne ke liye kahein. Toh, aap aik fast/cheap model ke saath guardrail chala sakte hain. Agar guardrail bad-niyat istemal ka pata laga leta hai, toh woh fori tor par error raise kar sakta hai, jo expensive model ko chalne se rokta hai aur aapka waqt/paisa bachata hai.

Guardrails ki do qismein hain:

1. **Input guardrails**: Yeh shuruaati user input par chalte hain.
2. **Output guardrails**: Yeh final agent output par chalte hain.

Input Guardrails

Input guardrails 3 steps mein chalte hain:

1. Sab se pehle, guardrail agent ko pass kiya gaya wohi input receive karta hai.
2. Agla, guardrail function `GuardrailFunctionOutput` paida karne ke liye chalta hai, jisay phir `InputGuardrailResult` mein wrap kiya jata hai.
3. Aakhir mein, hum check karte hain ke `.tripwire_triggered` true hai ya nahi. Agar true hai, toh `InputGuardrailTripwireTriggered` exception raise hota hai, taa ke aap user ko munasib jawab de sakein ya exception ko handle kar sakein.

Note: Input guardrails user input par chalne ke liye hain, isliye aik agent ke guardrails tabhi chalte hain jab agent **pehla** agent ho. Aap soch sakte hain, `guardrails` property agent par kyun hai, `Runner.run` ko pass karne ke bajaye? Iski wajah yeh hai ke guardrails asal Agent se mutaliq hotay hain - aap mukhtalif agents ke liye mukhtalif guardrails chalayenge, isliye code ko aik jagah rakhna readability ke liye mufeed hai.

Output Guardrails

Output guardrails 3 steps mein chalte hain:

1. Sab se pehle, guardrail agent ko pass kiya gaya wohi input receive karta hai.
2. Agla, guardrail function `GuardrailFunctionOutput` paida karne ke liye chalta hai, jisay phir `OutputGuardrailResult` mein wrap kiya jata hai.

3. Aakhir mein, hum check karte hain ke `.tripwire_triggered` true hai ya nahi. Agar true hai, toh `OutputGuardrailTripwireTriggered` exception raise hota hai, taa ke aap user ko munasib jawab de sakein ya exception ko handle kar sakein.

Note: Output guardrails final agent output par chalne ke liye hain, isliye aik agent ke guardrails tabhi chalte hain jab agent **aakhri** agent ho. Input guardrails ki tarah, hum aisa isliye karte hain kyunke guardrails asal Agent se mutaliq hotay hain - aap mukhtalif agents ke liye mukhtalif guardrails chalayenge, isliye code ko aik jagah rakhna readability ke liye mufeed hai.

Tripwires

Agar input ya output guardrail mein fail ho jata hai, toh Guardrail isay tripwire ke zariye signal kar sakta hai. Jaisay hi hum koi aisa guardrail dekhte hain jis ne tripwires ko trigger kiya hai, hum fori tor par `{Input,Output}GuardrailTripwireTriggered` exception raise karte hain aur Agent execution ko rokte hain.

Implementing a Guardrail (Guardrail Ko Implement Karna)

Aapko aik function faraham karna hoga jo input receive karta hai, aur `GuardrailFunctionOutput` wapas karta hai. Is misaal mein, hum yeh background mein aik Agent chala kar karein ge.

Python

```
from pydantic import BaseModel
from agents import (
    Agent,
    GuardrailFunctionOutput,
    InputGuardrailTripwireTriggered,
    RunContextWrapper,
    Runner,
    TResponseInputItem,
    input_guardrail,
)

class MathHomeworkOutput(BaseModel):
    is_math_homework: bool
    reasoning: str

guardrail_agent = Agent(
    name="Guardrail check",
    instructions="Check karo ke user aap se apna math homework solve karne ko keh raha hai ya
nahi.", # Check karein ke user aap se apna math homework hal karne ke liye keh raha hai ya
nahi.
    output_type=MathHomeworkOutput,
)

@input_guardrail
async def math_guardrail(
    ctx: RunContextWrapper[None], agent: Agent, input: str | list[TResponseInputItem]
) -> GuardrailFunctionOutput:
    result = await Runner.run(guardrail_agent, input, context=ctx.context)

    return GuardrailFunctionOutput(
        output_info=result.final_output,
        tripwire_triggered=result.final_output.is_math_homework,
    )

agent = Agent(
    name="Customer support agent",
    instructions="Aap aik customer support agent hain. Aap customers ko unke sawalon mein madad
karte hain.", # Aap aik customer support agent hain. Aap customers ko unke sawalaat mein madad
karte hain.
    input_guardrails=[math_guardrail],
)

async def main():
    # Is se guardrail trip hona chahiye
    try:
        await Runner.run(agent, "Hello, kya aap meri madad kar sakte hain x ko solve karne
mein: 2x + 3 = 11?") # Hello, kya aap meri madad kar sakte hain x ko hal karne mein: 2x + 3 =
11?
        print("Guardrail trip nahi hua - yeh ghair-mutawaqqe hai")

    except InputGuardrailTripwireTriggered:
        print("Math homework guardrail trip ho gaya")

if __name__ == "__main__":
    import asyncio
    asyncio.run(main())
```

Output guardrails bhi isi tarah hotay hain.

Python

```
from pydantic import BaseModel
from agents import (
    Agent,
    GuardrailFunctionOutput,
    OutputGuardrailTripwireTriggered,
    RunContextWrapper,
    Runner,
    output_guardrail,
)

class MessageOutput(BaseModel):
    response: str

class MathOutput(BaseModel):
    reasoning: str
    is_math: bool

guardrail_agent = Agent(
    name="Guardrail check",
    instructions="Check karo ke output mein koi math shamil hai ya nahi.", # Check karein ke
output mein koi math shamil hai ya nahi.
    output_type=MathOutput,
)

@output_guardrail
async def math_guardrail(
    ctx: RunContextWrapper, agent: Agent, output: MessageOutput
) -> GuardrailFunctionOutput:
    result = await Runner.run(guardrail_agent, output.response, context=ctx.context)

    return GuardrailFunctionOutput(
        output_info=result.final_output,
        tripwire_triggered=result.final_output.is_math,
    )

agent = Agent(
    name="Customer support agent",
    instructions="Aap aik customer support agent hain. Aap customers ko unke sawalon mein madad
karte hain.", # Aap aik customer support agent hain. Aap customers ko unke sawalaat mein madad
karte hain.
    output_guardrails=[math_guardrail],
    output_type=MessageOutput,
)

async def main():
    # Is se guardrail trip hona chahiye
    try:
        await Runner.run(agent, "Hello, kya aap meri madad kar sakte hain x ko solve karne
mein: 2x + 3 = 11?") # Hello, kya aap meri madad kar sakte hain x ko hal karne mein: 2x + 3 =
11?
        print("Guardrail trip nahi hua - yeh ghair-mutawaqqe hai")

    except OutputGuardrailTripwireTriggered:
        print("Math output guardrail trip ho gaya")

if __name__ == "__main__":
    import asyncio
    asyncio.run(main())
```

Orchestrating multiple agents

Orchestrating Multiple Agents (Kayee Agents Ko Organize Karna)

Orchestration aapki app mein agents ke flow ko kehte hain. Kon se agents chalte hain, kis tarteeb se, aur woh kaise faisla karte hain ke agla kya hoga? Agents ko organize karne ke do buniyadi tareeqay hain:

1. **LLM ko faisle karne dena**: Yeh LLM ki intelligence ko istemal karta hai taa ke woh plan kare, reasoning kare, aur uski buniyad par agle steps ka faisla kare.
2. **Code ke zariye organize karna**: Apne code ke zariye agents ke flow ka taayun karna.

Aap in patterns ko mila jul kar istemal kar sakte hain. Har aik ke apne tradeoffs hain, jin ki tafseel neechay di gayi hai.

Orchestrating via LLM (LLM Ke Zariye Organize Karna)

Aik agent aik LLM hota hai jo instructions, tools aur handoffs se lais hota hai. Iska matlab hai ke agar koi open-ended task diya jaye, toh LLM khudmukhtarana tor par yeh plan kar sakta hai ke woh task ko kaise hal karega, actions lene aur data hasil karne ke liye tools ka istemal karte huay, aur sub-agents ko tasks delegate karne ke liye handoffs ka istemal karte huay. Masalan, aik research agent ko in tools se lais kiya ja sakta hai:

- **Web search**: Online maloomat talash karne ke liye.
- **File search aur retrieval**: Proprietary data aur connections mein talash karne ke liye.
- **Computer use**: Computer par actions lene ke liye.
- **Code execution**: Data analysis karne ke liye.
- **Handoffs**: Specialized agents ko jo planning, report writing aur mazeed mein mahir hain.

Yeh pattern tab behtareen hai jab task open-ended ho aur aap LLM ki intelligence par bharosa karna chahte hon. Yahan sab se ahem tareeqay hain:

- **Achi prompts mein invest karein.** Wazeh karein ke kon se tools dastiyab hain, unhein kaise istemal karna hai, aur kin parameters ke andar woh operate karein ge.
- **Apni app ko monitor karein aur us par iterate karein.** Dekhein kahan ghaltiyan hoti hain, aur apni prompts par iterate karein.
- **Agent ko introspect aur improve karne ki ijazat dein.** Masalan, usay loop mein chalayein, aur usay khud ki tanqeed karne dein; ya, error messages faraham karein aur usay improve karne dein.

- **Aise specialized agents rakhein jo aik kaam mein mahir hon**, bajaye iske ke aik general purpose agent ho jis se kisi bhi cheez mein acha hone ki tawaqqo ki jaye.
- **Evals mein invest karein.** Yeh aapko apne agents ko behtar banane aur tasks mein behtar hone ke liye train karne deta hai.

Orchestrating via Code (Code Ke Zariye Organize Karna)

Jabke LLM ke zariye organize karna taaqatwar hai, code ke zariye organize karna tasks ko zyada deterministic aur qabil-e-predictable banata hai, raftaar, cost aur performance ke lihaz se. Yahan aam patterns hain:

- **Structured outputs ka istemal karna** taa ke achi tarah se bana hua data generate kiya ja sake jisay aap apne code se inspect kar sakein. Masalan, aap aik agent se task ko kuch categories mein classify karne ke liye keh sakte hain, aur phir category ki buniyad par agla agent chun sakte hain.
- **Mutadid agents ko chain karna** aik ke output ko agle ke input mein tabdeel kar ke. Aap aik blog post likhne jaise task ko steps ki series mein taqseem kar sakte hain - research karein, aik outline likhein, blog post likhein, us par tanqeed karein, aur phir usay improve karein.
- Task ko perform karne wale agent ko aik `while` loop mein chalana aik aise agent ke saath jo evaluate karta hai aur feedback faraham karta hai, jab tak ke evaluator na keh de ke output mukhtalif criteria par poora utarta hai.
- **Mutadid agents ko parallel mein chalana**, masalan Python primitives jaisay `asyncio.gather` ke zariye. Yeh raftaar ke liye mufeed hai jab aapke paas mutadid tasks hon jo aik doosre par depend nahi karte.

Hamare paas `examples/agent_patterns` mein kai misalein maujood hain.

https://github.com/openai/openai-agents-python/tree/main/examples/agent_patterns

MODELS

Models

Agents SDK OpenAI models ke liye out-of-the-box support ke saath aata hai, do tareeqon se:

- **Sifarish Karda**: **OpenAIResponsesModel**, jo naye **Responses API** ka istemal karte hue OpenAI APIs ko call karta hai.
- **OpenAIChatCompletionsModel**, jo **Chat Completions API** ka istemal karte hue OpenAI APIs ko call karta hai.

Non-OpenAI Models (Ghair-OpenAI Models)

Aap zyada tar doosre ghair-OpenAI models ko **LiteLLM integration** ke zariye istemal kar sakte hain. Sab se pehle, litellm dependency group install karein:

Bash

```
pip install "openai-agents[litellm]"
```

Phir, `litellm/` prefix ke saath kisi bhi **supported models** ka istemal karein:

Python

```
claude_agent = Agent(model="litellm/anthropic/claude-3-5-sonnet-20240620", ...)
gemini_agent = Agent(model="litellm/gemini/gemini-2.5-flash-preview-04-17", ...)
```

Other Ways to Use Non-OpenAI Models (Ghair-OpenAI Models Ko Istemal Karne Ke Doosre Tareeqay)

Aap doosre LLM providers ko 3 mazeed tareeqon se integrate kar sakte hain (examples **yahan** hain):

- **set_default_openai_client**: Yeh un mamlaat mein mufeed hai jahan aap globally `AsyncOpenAI` ki aik instance ko LLM client ke tor par istemal karna chahte hain. Yeh un mamlaat ke liye hai jahan LLM provider ke paas OpenAI compatible API endpoint hai, aur aap `base_url` aur `api_key` set kar sakte hain. Aik configurable example **examples/model_providers/custom_example_global.py** mein dekhein.
- **ModelProvider**: Yeh `Runner.run` level par hai. Yeh aapko yeh kehne deta hai ke "is run mein tamaam agents ke liye custom model provider istemal karo". Aik configurable example **examples/model_providers/custom_example_provider.py** mein dekhein.
- **Agent.model**: Yeh aapko aik khaas Agent instance par model specify karne deta hai. Yeh aapko mukhtalif agents ke liye mukhtalif providers ko mix aur match karne ke qabil banata hai. Aik configurable example **examples/model_providers/custom_example_agent.py** mein dekhein. Zyada tar dastiyab models ko istemal karne ka aik aasan tareeqa **LiteLLM integration** ke zariye hai.

Agar aapke paas `platform.openai.com` se API key nahi hai, toh hum `set_tracing_disabled()` ke zariye tracing ko disable karne, ya aik **different tracing processor** setup karne ki sifarish karte hain.

Note: In examples mein, hum Chat Completions API/model istemal karte hain, kyunke zyada tar LLM providers abhi tak Responses API ko support nahi karte. Agar aapka LLM provider isay support karta hai, toh hum Responses istemal karne ki sifarish karte hain.

Mixing and Matching Models (Models Ko Mix Aur Match Karna)

Aik single workflow ke andar, aap har agent ke liye mukhtalif models istemal karna chahenge. Masalan, aap triage ke liye aik chhota, tez model istemal kar sakte hain, jabke complex tasks ke liye aik bara, zyada qabil model istemal kar sakte hain. Aik **Agent** ko configure karte waqt, aap aik khaas model ko ya toh chun sakte hain:

- Aik model ka naam pass kar ke.
- Kisi bhi model ka naam + aik **ModelProvider** pass kar ke jo us naam ko Model instance se map kar sake.
- Seedha aik **Model** implementation faraham kar ke.

Note: Jabke hamara SDK **OpenAIResponsesModel** aur **OpenAIChatCompletionsModel** dono shapes ko support karta hai, hum har workflow ke liye aik single model shape istemal karne ki sifarish karte hain kyunke dono shapes features aur tools ka mukhtalif set support karte hain. Agar aapke workflow ko model shapes ko mix aur match karne ki zaroorat hai, toh yeh yakeeni banayein ke aap jo tamaam features istemal kar rahe hain woh dono par dastiyab hain.

Python

```python
from agents import Agent, Runner, AsyncOpenAI, OpenAIChatCompletionsModel
import asyncio

spanish_agent = Agent(
    name="Spanish agent",
    instructions="Aap sirf Spanish bolte hain.",
    model="o3-mini",
)

english_agent = Agent(
    name="English agent",
    instructions="Aap sirf English bolte hain.",
    model=OpenAIChatCompletionsModel(
        model="gpt-4o",
        openai_client=AsyncOpenAI()
    ),
)

triage_agent = Agent(
    name="Triage agent",
    instructions="Request ki zabaan ki buniyad par munasib agent ko handoff karo.", # Request
ki language ki buniyad par munasib agent ko handoff karein.
    handoffs=[spanish_agent, english_agent],
    model="gpt-3.5-turbo",
)

async def main():
    result = await Runner.run(triage_agent, input="Hola, ¿cómo estás?")
    print(result.final_output)

if __name__ == "__main__":
    asyncio.run(main())
```

Jab aap kisi agent ke liye istemal hone wale model ko mazeed configure karna chahte hain, toh aap **ModelSettings** pass kar sakte hain, jo ikhtiyari model configuration parameters jaisay temperature faraham karta hai.

Python

```
from agents import Agent, ModelSettings

english_agent = Agent(
    name="English agent",
    instructions="Aap sirf English bolte hain.",
    model="gpt-4o",
    model_settings=ModelSettings(temperature=0.1),
)
```

Common Issues with Using Other LLM Providers (Doosre LLM Providers Ko Istemal Karne Mein Aam Masail)

Tracing client error 401 (Tracing client error 401)

Agar aapko tracing se mutaliq errors milte hain, toh iski wajah yeh hai ke traces OpenAI servers par upload hote hain, aur aapke paas OpenAI API key nahi hai. Isay hal karne ke teen options hain:

1. **Tracing ko mukammal tor par disable karein**: `set_tracing_disabled(True)`.
2. **Tracing ke liye OpenAI key set karein**: `set_tracing_export_api_key(...)`. Yeh API key sirf traces upload karne ke liye istemal hogi, aur `platform.openai.com` se honi chahiye.
3. **Ghair-OpenAI trace processor istemal karein**. **tracing docs** dekhein.

Responses API support (Responses API ki support)

SDK default roop se Responses API istemal karta hai, lekin zyada tar doosre LLM providers abhi tak isay support nahi karte. Iske nateejay mein aapko 404s ya isi tarah ke masail dekhne ko mil sakte hain. Hal karne ke liye, do options hain:

1. **set_default_openai_api("chat_completions")** ko call karein. Yeh kaam karta hai agar aap `OPENAI_API_KEY` aur `OPENAI_BASE_URL` ko environment variables ke zariye set kar rahe hain.
2. **OpenAIChatCompletionsModel** istemal karein. Examples **yahan** hain.

Structured outputs support (Structured outputs ki support)

Kuch model providers mein **structured outputs** ki support nahi hoti. Iske nateejay mein kabhi kabhi aik error aata hai jo is tarah ka dikhta hai:

```
BadRequestError: Error code: 400 - {'error': {'message': "'response_format.type' : value is not
one of the allowed values ['text','json_object']", 'type': 'invalid_request_error'}}
```

Yeh kuch model providers ki kami hai - woh JSON outputs ko support karte hain, lekin aapko output ke liye `json_schema` specify karne ki ijazat nahi dete. Hum iska hal talash kar rahe hain, lekin hum un providers par bharosa karne ki sifarish karte hain jo JSON schema output ko support karte hain, kyunke warna aapki app aksar kharab JSON ki wajah se toot jayegi.

Mixing models across providers (Providers ke darmiyan models ko mix karna)

Aapko model providers ke darmiyan feature differences se waqif hona zaroori hai, warna aapko errors ka saamna karna pad sakta hai. Masalan, OpenAI structured outputs, multimodal input, aur hosted file search aur web search ko support karta hai, lekin boht se doosre providers in features ko support nahi karte. In hudood se waqif rahen:

- Aise providers ko **unsupported tools** na bhejein jo unhein nahi samajhte.
- Text-only models ko call karne se pehle multimodal inputs ko filter out karein.
- Yaad rakhein ke providers jo structured JSON outputs ko support nahi karte woh kabhi kabhi invalid JSON paida karein ge.

Using any model via LiteLLM

LiteLLM ke Zariye Koi Bhi Model Istemal Karna

Note: LiteLLM integration abhi beta mein hai. Aapko kuch model providers ke saath masail ka saamna karna pad sakta hai, khaas tor par chhote providers ke saath. Barae meharbani, **Github issues** ke zariye koi bhi masla report karein, aur hum fori theek kar denge.

LiteLLM aik library hai jo aapko single interface ke zariye 100+ models istemal karne ki ijazat deta hai. Humne Agents SDK mein LiteLLM integration shamil kiya hai taa ke aap koi bhi AI model istemal kar sakein.

Setup

Aapko yeh yakeeni banana hoga ke **litellm** dastiyab hai. Aap isay optional **litellm** dependency group install kar ke kar sakte hain:

Bash

```
pip install "openai-agents[litellm]"
```

Aik baar ho jane ke baad, aap kisi bhi agent mein **LitellmModel** istemal kar sakte hain.

Example

Yeh aik mukammal working example hai. Jab aap isay chalayenge, toh aapko model name aur API key ke liye prompt kiya jayega. Masalan, aap enter kar sakte hain:

- **openai/gpt-4.1** model ke liye, aur apni OpenAI API key.
- **anthropic/claude-3-5-sonnet-20240620** model ke liye, aur apni Anthropic API key.
- Waghera.

LiteLLM mein supported models ki mukammal list ke liye, **litellm providers docs** dekhein.

Python

```python
from __future__ import annotations

import asyncio

from agents import Agent, Runner, function_tool, set_tracing_disabled
from agents.extensions.models.litellm_model import LitellmModel

@function_tool
def get_weather(city: str):
    print(f"[debug] {city} ke liye mausam maloom kiya ja raha hai") # [debug] getting weather for {city}
    return f"{city} mein mausam dhoop wala hai." # The weather in {city} is sunny.

async def main(model: str, api_key: str):
    agent = Agent(
        name="Assistant",
        instructions="Aap sirf haikus mein jawab dete hain.", # You only respond in haikus.
        model=LitellmModel(model=model, api_key=api_key),
        tools=[get_weather],
    )

    result = await Runner.run(agent, "Tokyo mein mausam kaisa hai?") # What's the weather in Tokyo?
    print(result.final_output)

if __name__ == "__main__":
    # Pehle args se model/api key lene ki koshish karein
    import argparse

    parser = argparse.ArgumentParser()
    parser.add_argument("--model", type=str, required=False)
    parser.add_argument("--api-key", type=str, required=False)
    args = parser.parse_args()

    model = args.model
    if not model:
        model = input("Litellm ke liye model ka naam enter karein: ") # Enter a model name for Litellm:

    api_key = args.api_key
    if not api_key:
        api_key = input("Litellm ke liye API key enter karein: ") # Enter an API key for Litellm:

    asyncio.run(main(model, api_key))
```

Configuring the SDK

SDK Ko Configure Karna

SDK (Software Development Kit) ko configure karna aapko apne Agent applications ki performance aur behavior ko control karne ki ijazat deta hai. Yeh hissa batata hai ke aap API keys, tracing, aur logging ko kaise set up kar sakte hain.

API Keys aur Clients

By default, SDK LLM requests aur tracing ke liye **OPENAI_API_KEY** environment variable ko dekhta hai jaisay hi isay import kiya jata hai. Agar aap apni app shuru hone se pehle is environment variable ko set nahi kar sakte, toh aap key set karne ke liye **set_default_openai_key()** function istemal kar sakte hain.

Python

```
from agents import set_default_openai_key

set_default_openai_key("sk-...") # Apni API key yahan dalein
```

Mutabadil ke tor par, aap istemal hone wale OpenAI client ko bhi configure kar sakte hain. Default roop se, SDK aik **AsyncOpenAI** instance banata hai, environment variable ya upar di gayi default key se API key ka istemal karte huay. Aap **set_default_openai_client()** function ka istemal kar ke isay tabdeel kar sakte hain.

Python

```
from openai import AsyncOpenAI
from agents import set_default_openai_client

custom_client = AsyncOpenAI(base_url="...", api_key="...") # Apni base_url aur API key
configure karein
set_default_openai_client(custom_client)
```

Aakhir mein, aap istemal hone wale OpenAI API ko bhi customize kar sakte hain. Default roop se, hum OpenAI Responses API istemal karte hain. Aap isay **set_default_openai_api()** function ka istemal kar ke Chat Completions API istemal karne ke liye override kar sakte hain.

Python

```
from agents import set_default_openai_api

set_default_openai_api("chat_completions")
```

Tracing

Tracing default roop se enable hota hai. Yeh default roop se upar wale section se OpenAI API keys (yaani environment variable ya aapki set karda default key) istemal karta hai. Aap khaas tor par tracing ke liye istemal hone wali API key ko **set_tracing_export_api_key** function ka istemal kar ke set kar sakte hain.

Python

```
from agents import set_tracing_export_api_key

set_tracing_export_api_key("sk-...") # Tracing export ke liye apni API key dalein
```

Aap **set_tracing_disabled()** function ka istemal kar ke tracing ko mukammal tor par disable bhi kar sakte hain.

Python

```
from agents import set_tracing_disabled

set_tracing_disabled(True)
```

Debug Logging

SDK mein do Python loggers hain jin mein koi handlers set nahi hotay. Default roop se, iska matlab hai ke warnings aur errors **stdout** par bheje jate hain, lekin doosre logs suppress kiye jate hain. Verbose logging ko enable karne ke liye, **enable_verbose_stdout_logging()** function istemal karein.

Python

```
from agents import enable_verbose_stdout_logging

enable_verbose_stdout_logging()
```

Mutabadil ke tor par, aap handlers, filters, formatters, waghera shamil kar ke logs ko customize kar sakte hain. Aap **Python logging guide** mein mazeed parh sakte hain.

Python

```
import logging

logger = logging.getLogger("openai.agents") # Ya openai.agents.tracing Tracing logger ke liye

# Tamaam logs ko show karne ke liye
logger.setLevel(logging.DEBUG)
# Info aur us se upar ke logs ko show karne ke liye
logger.setLevel(logging.INFO)
# Warning aur us se upar ke logs ko show karne ke liye
logger.setLevel(logging.WARNING)
# waghera
# Aap isay zaroorat ke mutabiq customize kar sakte hain, lekin yeh default roop se `stderr` par output karega
logger.addHandler(logging.StreamHandler())
```

Sensitive Data in Logs (Logs Mein Hassas Data)

Kuch logs mein mumkina tor par hassas data shamil ho sakta hai (masalan, user data). Agar aap is data ko log hone se rokna chahte hain, toh darj zail environment variables set karein. LLM inputs aur outputs ki logging ko disable karne ke liye:

Bash

```
export OPENAI_AGENTS_DONT_LOG_MODEL_DATA=1
```

Tool inputs aur outputs ki logging ko disable karne ke liye:

Bash

```
export OPENAI_AGENTS_DONT_LOG_TOOL_DATA=1
```

Agent Visualization

Agent Visualization (Agent Ka Tasawwur)

Agent visualization aapko Graphviz ka istemal karte huay agents aur unke rishton ki aik structured graphical representation banane ki ijazat deta hai. Yeh samjhne ke liye mufeed hai ke agents, tools, aur handoffs aik application ke andar kaise interact karte hain.

Installation

Optional `viz` dependency group install karein:

Bash

```
pip install "openai-agents[viz]"
```

Generating a Graph (Graph Banana)

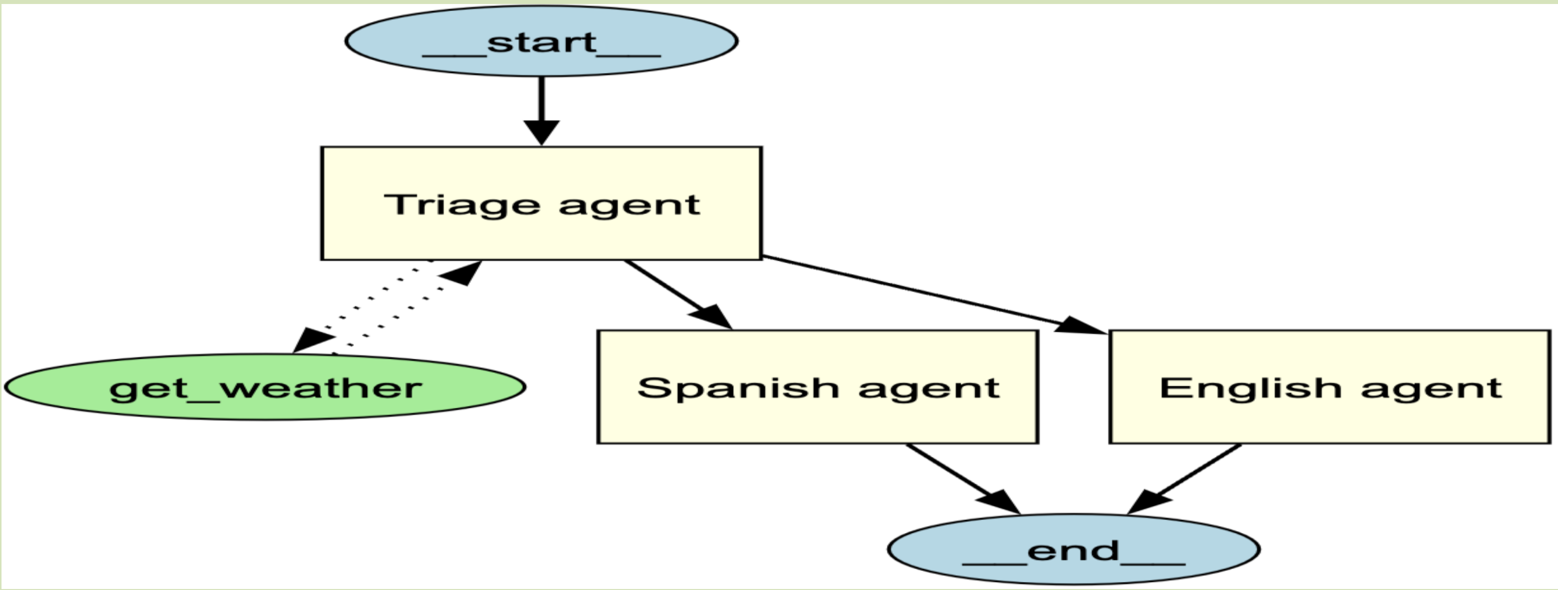
Aap **draw_graph** function ka istemal kar ke agent visualization bana sakte hain. Yeh function aik directed graph banata hai jahan:

- **Agents** ko peele boxes se represent kiya jata hai.
- **Tools** ko sabz ellipses se represent kiya jata hai.
- **Handoffs** aik agent se doosre agent tak directed edges hotay hain.

Example Usage (Misaal)

Python

```
from agents import Agent, function_tool
from agents.extensions.visualization import draw_graph

@function_tool
def get_weather(city: str) -> str:
    return f"The weather in {city} is sunny."

spanish_agent = Agent(
    name="Spanish agent",
    instructions="Aap sirf Spanish bolte hain.",
)

english_agent = Agent(
    name="English agent",
    instructions="Aap sirf English bolte hain",
)

triage_agent = Agent(
    name="Triage agent",
    instructions="Request ki zaban ki buniyad par munasib agent ko handoff karein.",
    handoffs=[spanish_agent, english_agent],
    tools=[get_weather],
)

draw_graph(triage_agent)
```



Yeh aik graph banata hai jo **triage agent** ki structure aur uske sub-agents aur tools se connections ko vizually represent karta hai.

Understanding the Visualization (Visualization Ko Samjhna)

Bana hua graph shamil karta hai:

- Aik **start node** (`__start__`) jo entry point ki nishandahi karta hai.
- **Agents** ko peele rang se bhare hue **rectangles** ke tor par represent kiya jata hai.
- **Tools** ko sabz rang se bhare hue **ellipses** ke tor par represent kiya jata hai.
- Directed edges interactions ki nishandahi karte hain:
    - **Solid arrows** agent-to-agent handoffs ke liye.
    - **Dotted arrows** tool invocations ke liye.
- Aik **end node** (`__end__`) jo batata hai ke execution kahan khatam hoti hai.

Customizing the Graph (Graph Ko Customize Karna)

Showing the Graph (Graph Ko Dikhana)

By default, `draw_graph` graph ko inline display karta hai. Graph ko aik alag window mein dikhane ke liye, darj zail likhein:

Python

```
draw_graph(triage_agent).view()
```

Saving the Graph (Graph Ko Save Karna)

By default, `draw_graph` graph ko inline display karta hai. Isay file ke tor par save karne ke liye, filename specify karein:

Python

```
draw_graph(triage_agent, filename="agent_graph")
```

Yeh working directory mein **agent_graph.png** banayega.

Release process

Yeh project semantic versioning ka thora sa tabdeel shuda version follow karta hai, **0.Y.Z** ki shakal mein. Shuru ka **0** is baat ki nishandahi karta hai ke SDK abhi tezi se evolve ho raha hai. Components ko darj zail tareeqay se badhaein:

Minor (Y) Versions

Hum **minor versions Y** ko kisi bhi public interfaces mein **breaking changes** ke liye badhayenge jo beta ke tor par mark nahi kiye gaye hain. Masalan, **0.0.x** se **0.1.x** tak jane mein breaking changes shamil ho sakte hain. Agar aap breaking changes nahi chahte, toh hum aapke project mein **0.0.x versions** par pin karne ki sifarish karte hain.

Patch (Z) Versions

Hum **Z** ko non-breaking changes ke liye badhayenge:

- Bug fixes
- Naye features
- Private interfaces mein tabdeelian
- Beta features mein updates

Voice Agents

Voice Quickstart: Agents Ko Bolne Par Majboor Karna

Agar aap Agents SDK ke saath apne AI ko zabaan ki salahiyat dena chahte hain, toh yeh quickstart aapki rehnumai karega.

Prerequisites (Buniyadi Zaruriyat)

Sab se pehle, yakeeni banayein ke aapne Agents SDK ke liye **base quickstart instructions** par amal kiya hai aur aik virtual environment set up kiya hai. Iske baad, SDK se optional voice dependencies install karein:

Bash

```
pip install 'openai-agents[voice]'
```

Concepts (Buniyadi Tasawwurat)

Sab se ahem concept jise aapko samajhna hai, woh **VoicePipeline** hai. Yeh aik 3-step process hai:

1. **Run a speech-to-text model**: Audio ko text mein tabdeel karne ke liye speech-to-text model chalana.
2. **Run your code (agentic workflow)**: Aapka code chalana, jo aam tor par aik agentic workflow hota hai, taa ke nateeja paida ho.
3. **Run a text-to-speech model**: Nateeje ke text ko dobarah audio mein tabdeel karne ke liye text-to-speech model chalana.

```
          +---------------------+
           |   □ Audio Input    |
          +----------|----------+
                     |
                     v
          +---------------------+
 | Transcribe          |   (Speech-to-Text)
          +----------|----------+
                     |
                     v
          +---------------------+
| Your Code            |   (Agentic Workflow)
          +----------|----------+
                     |
                     v
          +---------------------+
          | Text-to-Speech      |
          +----------|----------+
                     |
                     v
          +---------------------+
           |   □ Audio Output   |
          +---------------------+
```

Agents (Agents Ki Setting)

Sab se pehle, kuch Agents set up karte hain. Agar aapne is SDK ke saath koi agents banaye hain, toh yeh aapko jaana pehchana lagega. Hamare paas kuch Agents, aik handoff, aur aik tool hoga.

Python

```
import asyncio
import random

from agents import (
    Agent,
    function_tool,
)
from agents.extensions.handoff_prompt import prompt_with_handoff_instructions

@function_tool
def get_weather(city: str) -> str:
    """Get the weather for a given city."""
    print(f"[debug] get_weather called with city: {city}")
    choices = ["sunny", "cloudy", "rainy", "snowy"]
    return f"The weather in {city} is {random.choice(choices)}."

spanish_agent = Agent(
    name="Spanish",
    handoff_description="Aik Spanish bolne wala agent.", # A Spanish speaking agent.
    instructions=prompt_with_handoff_instructions(
        "Aap aik insaan se baat kar rahe hain, isliye polite aur concise rahen. Spanish mein
baat karein.", # You're speaking to a human, so be polite and concise. Speak in Spanish.
    ),
    model="gpt-4o-mini",
)

agent = Agent(
    name="Assistant",
    instructions=prompt_with_handoff_instructions(
        "Aap aik insaan se baat kar rahe hain, isliye polite aur concise rahen. Agar user
Spanish mein baat kare, toh Spanish agent ko handoff karein.", # You're speaking to a human, so
be polite and concise. If the user speaks in Spanish, handoff to the Spanish agent.
    ),
    model="gpt-4o-mini",
    handoffs=[spanish_agent],
    tools=[get_weather],
)
```

Voice Pipeline (Voice Pipeline Banana)

Hum `SingleAgentVoiceWorkflow` ko workflow ke tor par istemal karte huay aik simple voice pipeline set up karein ge.

Python

```
from agents.voice import SingleAgentVoiceWorkflow, VoicePipeline
pipeline = VoicePipeline(workflow=SingleAgentVoiceWorkflow(agent))
```

Run the Pipeline (Pipeline Ko Chalana)

Python

```
import numpy as np
import sounddevice as sd
from agents.voice import AudioInput

# Sadgi ke liye, hum sirf 3 seconds ki khamoshi banayenge
# Asal mein, aap microphone data hasil karein ge
buffer = np.zeros(24000 * 3, dtype=np.int16)
audio_input = AudioInput(buffer=buffer)

result = await pipeline.run(audio_input)

# `sounddevice` ka istemal karte huay aik audio player banayein
player = sd.OutputStream(samplerate=24000, channels=1, dtype=np.int16)
player.start()

# Audio stream ko jaisay hi aata hai chalayein
async for event in result.stream():
    if event.type == "voice_stream_event_audio":
        player.write(event.data)
```

Put It All Together (Sab Ko Ikhatta Karna)

Python

```
import asyncio
import random

import numpy as np
import sounddevice as sd

from agents import (
    Agent,
    function_tool,
    set_tracing_disabled,
)
from agents.voice import (
    AudioInput,
    SingleAgentVoiceWorkflow,
    VoicePipeline,
)
from agents.extensions.handoff_prompt import prompt_with_handoff_instructions


@function_tool
def get_weather(city: str) -> str:
    """Get the weather for a given city."""
    print(f"[debug] get_weather called with city: {city}")
    choices = ["sunny", "cloudy", "rainy", "snowy"]
    return f"The weather in {city} is {random.choice(choices)}."


spanish_agent = Agent(
    name="Spanish",
    handoff_description="Aik Spanish bolne wala agent.",
    instructions=prompt_with_handoff_instructions(
        "Aap aik insaan se baat kar rahe hain, isliye polite aur concise rahen. Spanish mein
baat karein.",
    ),
    model="gpt-4o-mini",
)

agent = Agent(
    name="Assistant",
    instructions=prompt_with_handoff_instructions(
        "Aap aik insaan se baat kar rahe hain, isliye polite aur concise rahen. Agar user
Spanish mein baat kare, toh Spanish agent ko handoff karein.",
    ),
    model="gpt-4o-mini",
    handoffs=[spanish_agent],
    tools=[get_weather],
)


async def main():
    pipeline = VoicePipeline(workflow=SingleAgentVoiceWorkflow(agent))
    buffer = np.zeros(24000 * 3, dtype=np.int16) # Yahan, hum sirf 3 seconds ki khamoshi
istemal kar rahe hain. Asal mein, aap microphone se audio input lenge.
    audio_input = AudioInput(buffer=buffer)

    result = await pipeline.run(audio_input)

    # `sounddevice` ka istemal karte huay aik audio player banayein
    player = sd.OutputStream(samplerate=24000, channels=1, dtype=np.int16)
    player.start()

    # Audio stream ko jaisay hi aata hai chalayein
    async for event in result.stream():
        if event.type == "voice_stream_event_audio":
            player.write(event.data)


if __name__ == "__main__":
    asyncio.run(main())
```

Agar aap is misaal ko chalayenge, toh agent aap se baat karega! `examples/voice/static` mein misaal dekhein taa ke aik demo dekh sakein jahan aap khud agent se baat kar sakte hain.

Pipelines and workflows

Pipelines aur Workflows (Pipelines aur Workflows)

VoicePipeline aik class hai jo aapke agentic workflows ko voice app mein tabdeel karna aasan banata hai. Aapko run karne ke liye aik workflow pass karna hota hai, aur pipeline audio input ko transcribe karne, audio ke khatam hone ka pata lagane, sahih waqt par aapke workflow ko call karne, aur workflow output ko dobarah audio mein tabdeel karne ka khayal rakhta hai.

```
+----------------------+
|   □ Audio Input     |
+----------|----------+
           |
           v
+----------------------+
| Transcribe          |   (Speech-to-Text)
+----------|----------+
           |
           v
+----------------------+
| Your Code           |   (Agentic Workflow)
+----------|----------+
           |
           v
+----------------------+
| Text-to-Speech      |
+----------|----------+
           |
           v
+----------------------+
|   □ Audio Output    |
+----------------------+
```

Configuring a Pipeline (Pipeline Ko Configure Karna)

Jab aap aik pipeline banate hain, toh aap kuch cheezein set kar sakte hain:

- **The workflow**: Woh code jo har baar naye audio ke transcribe hone par chalta hai.
- **The speech-to-text and text-to-speech models used**: Istemal hone wale speech-to-text aur text-to-speech models.
- **The config**: Jo aapko cheezein configure karne deta hai jaisay:
  - Aik model provider, jo model names ko models se map kar sakta hai.
  - Tracing, jis mein tracing ko disable karna, audio files ka upload hona, workflow name, trace IDs waghera shamil hain.
  - TTS (Text-to-Speech) aur STT (Speech-to-Text) models par settings, jaisay prompt, zaban aur istemal hone wale data types.

Running a Pipeline (Pipeline Ko Chalana)

Aap `run()` method ke zariye pipeline ko chala sakte hain, jo aapko audio input ko do tareeqon se pass karne deta hai:

- **AudioInput**: Isay tab istemal kiya jata hai jab aapke paas aik mukammal audio transcript ho, aur aap sirf uske liye nateeja paida karna chahte hon. Yeh un mamlaat mein mufeed hai jahan aapko yeh pata lagane ki zaroorat nahi hoti ke speaker bolna kab khatam karta hai; masalan, jab aapke paas pehle se record shuda audio ho ya push-to-talk apps mein jahan yeh wazeh hota hai ke user ne bolna kab khatam kiya hai.
- **StreamedAudioInput**: Isay tab istemal kiya jata hai jab aapko yeh pata lagane ki zaroorat ho sakti hai ke user ne bolna kab khatam kiya hai. Yeh aapko audio chunks ko jaisay hi detect hon push karne ki ijazat deta hai, aur voice pipeline "activity detection" namak aik process ke zariye agent workflow ko sahih waqt par automatically chalayega.

Results (Nateejay)

Voice pipeline run ka nateeja aik **StreamedAudioResult** hota hai. Yeh aik object hai jo aapko events ko jaisay hi woh hotay hain stream karne deta hai. **VoiceStreamEvent** ki kuch qismein hain, jin mein shamil hain:

- `VoiceStreamEventAudio`: Jis mein audio ka aik chunk hota hai.
- `VoiceStreamEventLifecycle`: Jo aapko lifecycle events ke baray mein inform karta hai jaisay turn ka shuru hona ya khatam hona.
- `VoiceStreamEventError`: Yeh aik error event hai.

Python

```
result = await pipeline.run(input)

async for event in result.stream():
    if event.type == "voice_stream_event_audio":
        # audio chalayein (play audio)
    elif event.type == "voice_stream_event_lifecycle":
        # lifecycle event handle karein
    elif event.type == "voice_stream_event_error":
        # error event handle karein
    # ...
```

Best Practices (Behtareen Tareeqay)

Interruptions (Dakhlandazi)

Agents SDK filhal **StreamedAudioInput** ke liye koi built-in interruptions support nahi karta. Iske bajaye har detect shuda turn ke liye yeh aapke workflow ka aik alag run trigger karega. Agar aap apni application ke andar interruptions ko handle karna chahte hain toh aap `VoiceStreamEventLifecycle` events ko sun sakte hain. `turn_started` yeh batayega ke aik naya turn transcribe ho gaya hai aur processing shuru ho rahi hai. `turn_ended` tab trigger hoga jab aik mutaliqa turn ke liye saara audio dispatch ho chuka hoga. Aap in events ka istemal speaker ke microphone ko mute karne ke liye kar sakte hain jab model aik turn shuru karta hai aur microphone ko unmute kar sakte hain jab aapne us turn ke liye saara mutaliqa audio flush kar diya ho.

Tracing

Tracing (Tracing)

Bilkul isi tarah jaise **agents ko trace kiya jata hai**, voice pipelines ko bhi automatically trace kiya jata hai.

Buniyadi tracing maloomat ke liye aap upar di gayi tracing doc parh sakte hain, lekin aap `VoicePipelineConfig` ke zariye aik pipeline ki tracing ko mazeed configure kar sakte hain.

Tracing se mutaliq ahem fields darj zail hain:

- `tracing_disabled`: Yeh control karta hai ke tracing disable ki gayi hai ya nahi. By default, tracing enable hoti hai.

- `trace_include_sensitive_data`: Yeh control karta hai ke traces mein mumkina tor par hassas data shamil hai ya nahi, jaisay audio transcripts. Yeh khaas tor par voice pipeline ke liye hai, aur us har cheez ke liye nahi jo aapke Workflow ke andar hoti hai.
- `trace_include_sensitive_audio_data`: Yeh control karta hai ke traces mein audio data shamil hai ya nahi.
- `workflow_name`: Trace workflow ka naam.
- `group_id`: Trace ka `group_id`, jo aapko mutadid traces ko link karne deta hai.
- `trace_metadata`: Trace ke saath shamil karne ke liye mazeed metadata.

API Reference

1) AGENTS
2) TRACING
3) VOICE
4) EXTENSION

AGENTS

Agents module

set_default_openai_key

set_default_openai_key(

key: str, use_for_tracing: bool = True

) -> None

"OpenAI API key ko default ke tor par set karein jo LLM requests ke liye istemal hogi (aur ikhtiyari tor par tracing ke liye bhi). Yeh sirf tab zaroori hai jab `OPENAI_API_KEY` environment variable pehle se set na ho. Agar yeh key faraham ki jati hai, toh isay `OPENAI_API_KEY` environment variable ki bajaye istemal kiya jayega."

Parameters:

Name	Type	Description	Default
client	AsyncOpenAI	The OpenAI client to use.	*required*
use_for_tracing	bool	Whether to use the API key from this client for uploading traces. If False, you'll either need to set the OPENAI_API_KEY environment variable or call set_tracing_export_api_key() with the API key you want to use for tracing.	True

Source code in src/agents/__init__.py

Agents Module Ke Functions

`set_default_openai_api`

Yeh function aapko yeh tay karne deta hai ke aapka SDK OpenAI LLM (Large Language Model) requests ke liye konsa API istemal karega. By default, SDK "responses" API ko use karta hai, jo ke naya standard hai. Lekin agar aapko zarurat ho toh aap isay "chat_completions" API par bhi set kar sakte hain.

Parameters:

- **`api`** (`Literal["chat_completions", "responses"]`):
    - **Tafseel:** Woh API type jo aap OpenAI LLM requests ke liye istemal karna chahte hain. Aap "chat_completions" ya "responses" mein se koi aik chun sakte hain.

`set_tracing_export_api_key`

Yeh function khaas tor par us **OpenAI API key** ko set karta hai jo sirf tracing data ko OpenAI ke backend par export karne ke liye istemal hogi. Yeh us waqt faida mand hai jab aap apne main LLM operations ke liye koi aur key istemal kar rahe hon aur tracing ke liye koi alag key rakhna chahte hon.

`set_tracing_disabled`

Yeh function aapki poori Agents SDK application mein tracing ki functionality ko **globally disable ya enable** karne ka option deta hai. Agar aap isay `True` par set karte hain, toh koi bhi trace data jama nahi hoga aur na hi bheja jayega. Tracing by default enable hoti hai.

`set_trace_processors`

Yeh aik taaqatwar function hai jo aapko SDK ke default trace processors ki list ko mukammal tor par **tabdeel (replace)** karne ki ijazat deta hai. Trace processors woh components hote hain jo jama kiye gaye trace data ko handle aur export karte hain. Apni `TracingProcessor` objects ki list faraham karke, aap traces ko kisi aur jagah (masalan, apne custom monitoring system) redirect kar sakte hain ya traces ko process karne ke liye apni khaas logic bana sakte hain. Yaad rakhein, agar aapki custom list mein koi aisa processor shamil nahi hai jo data OpenAI ke backend par bhejta hai, toh traces OpenAI ko nahi bheje jayenge.

`enable_verbose_stdout_logging`

Yeh aik asaan utility function hai jo aapke standard output (aam tor par aapke console) par **tafseeli logging output** ko enable karta hai. By default, SDK shayad sirf warnings aur errors hi dikhaye. Is function ko call karna development aur debugging ke dauran bohat faida mand hai kyunke yeh aapke agents ke andaruni kaam aur flow ke baray mein bohat ziyada maloomat faraham karta hai.

Agents

ToolsToFinalOutputFunction module-attribute

```
ToolsToFinalOutputFunction: TypeAlias = Callable[

    [RunContextWrapper[TContext], list[FunctionToolResult]],

    MaybeAwaitable[ToolsToFinalOutputResult],

]
```

Yeh aik aisa function hai jo **run context** aur **tool results ki list** leta hai, aur phir **`ToolsToFinalOutputResult`** return karta hai.

Is function ka buniyadi maqsad yeh tay karna hai ke jab aapke tools koi kaam mukammal kar len, toh unke nateejay ko seedha agent ka aakhri output samjha jaye, ya LLM (Large Language Model) ko mazeed processing aur jawab dene ke liye wapas bheja jaye.

`ToolsToFinalOutputResult` Dataclass (ToolsToFinalOutputResult Dataclass)

`ToolsToFinalOutputResult` aik khaas kism ka **dataclass** hai. Yeh tab istemal hota hai jab aapke agent ne koi tool call kiya ho, aur ab aapko yeh tay karna ho ke us tool ke nateeje (output) ko kaise handle kiya jaye. Kya woh nateeja seedha agent ka aakhri jawab ban jayega, ya LLM (Large Language Model) ko mazeed processing ke liye wapas bheja jayega?

Is dataclass mein do ahem hisse hote hain:

- **`is_final_output`** (`bool`)
  - **Yeh aik boolean value hai** jo batati hai ke kya tool ka nateeja seedha **final output** hai.
  - Agar yeh `True` hai, toh tool ka output hi agent ka aakhri jawab ban jayega.
  - Agar yeh `False` hai, toh iska matlab hai ke LLM dobara chalega. Woh tool ke output ko input ke tor par receive karega aur phir uski buniyad par mazeed jawab dega.
- **`final_output`** (`Any | None`, Default: `None`)
  - Yeh woh **aakhri output** hai jo agent faraham karega.
  - Agar `is_final_output` `False` hai, toh yeh `None` ho sakta hai (yaani abhi final output nahi bana).
  - Agar `is_final_output` `True` hai, toh is `final_output` mein koi value honi chahiye, aur woh agent ke `output_type` se match karni chahiye (yaani jo data type agent se output mein aanay ki tawaqqo ki ja rahi hai).

`StopAtTools` (TypedDict)

`StopAtTools` aik khaas kism ka configuration object hai jo `TypedDict` ki buniyad par bana hai. Iska maqsad yeh tay karna hai ke aapka agent kab mazeed kaam karna band kar de, khaas tor par jab woh tools ka istemal kar raha ho.

Is mein sirf aik khaas attribute hota hai:

- **`stop_at_tool_names`** (`list[str]`)
    - Yeh **tool names ki aik list** hai.
    - Agar agent in mein se kisi bhi tool ko call karta hai, toh **agent wahi ruk jayega** aur mazeed processing nahi karega.
    - Is surat mein, jo tool call hua hai, uska output hi seedha agent ka **final output** ban jata hai, aur Large Language Model (LLM) us nateeje ko mazeed process nahi karta.

`MCPConfig` (TypedDict)

`MCPConfig` aik khaas qism ki configuration hai jo **MCP servers** ke liye istemal hoti hai. Yeh aik `TypedDict` ki buniyad par banaya gaya hai, jiska matlab hai ke yeh data structure type-hinted keys aur unki values ko define karta hai.

Is configuration mein sirf aik khaas attribute hota hai:

- **`convert_schemas_to_strict`** (`NotRequired[bool]`, Default: `False`)
    - Agar aap isay `True` par set karte hain, toh SDK **MCP schemas ko strict-mode schemas mein tabdeel karne ki koshish karega**.
    - Iska maqsad schemas ko zyada sakht (strict) banana hai taake data validation behtar ho sake.
    - Lekin yaad rahe, yeh aik **"best-effort conversion"** hai. Iska matlab hai ke yeh system apni poori koshish karega, magar ho sakta hai ke kuch schemas puri tarah se tabdeel na ho sakein.
    - By default, yeh `False` par set hota hai, yaani schemas ko strict-mode mein tabdeel nahi kiya jata.

`Agent` Dataclass (Agent Dataclass)

`Agent` aik **AI model** hai jise instructions, tools, guardrails, handoffs aur bahut kuch ke saath configure kiya jata hai. Yeh aapke AI application ka dil hai.

Hum is mein **`instructions`** (jo agent ke liye "system prompt" ka kaam karta hai) ko shamil karne ki sakhti se sifarish karte hain. Iske ilawa, aap **`handoff_description`** bhi de sakte hain, jo agent ki insani-qabil-e-faham tafseel hoti hai aur tab istemal hoti hai jab agent tools/handoffs ke andar use hota hai.

Agents `context type` par generic hote hain. **Context** aik object hai jo aap banate hain. Yeh tools functions, handoffs, guardrails, waghera ko pass kiya jata hai.

Agent Ke Ahem Attributes

- **`name`** (`str`):
    - Agent ka **naam**.
- **`instructions`** (`str | Callable | None`):
    - Agent ke liye **hidayaat**. Jab yeh agent invoke hoga, toh yeh "system prompt" ke tor par istemal hoga. Yeh batata hai ke agent ko kya karna chahiye aur kaise jawab dena chahiye.
    - Yeh aik seedhi `string` ho sakti hai, ya phir aik `function` jo dynamically (chalte waqt) agent ke liye instructions banaye. Agar aap function dete hain, toh woh context aur agent instance ke saath call hoga aur usay aik `string` return karni hogi.
- **`prompt`** (`Prompt | DynamicPromptFunction | None`):

  - Aik **prompt object** (ya aik function jo `Prompt` return kare). Prompts aapko instructions, tools aur doosri config ko apne code se bahar dynamically configure karne dete hain. Yeh sirf OpenAI models ke saath, Responses API ka istemal karte huay, mumkin hai.
- **`handoff_description`** (`str | None`):
  - Agent ki **tafseel**. Yeh tab istemal hota hai jab agent ko handoff ke tor par use kiya jata hai, taake aik LLM ko pata chale ke yeh kya karta hai aur kab isay invoke karna hai.
- **`handoffs`** (`list[Agent[Any] | Handoff[TContext]]`):
  - **Handoffs woh sub-agents hain** jinhein agent apna kaam delegate kar sakta hai. Aap handoffs ki list faraham kar sakte hain, aur agent mutaliqa hone par unhein kaam sonpne ka faisla kar sakta hai. Yeh kaam ki taqseem (separation of concerns) aur modularity (tukron mein kaam) ki ijazat deta hai.
- **`model`** (`str | Model | None`):
  - LLM (Large Language Model) ko invoke karte waqt istemal hone wali **model implementation**.
  - By default, agar yeh set na ho, toh agent `openai_provider.DEFAULT_MODEL` (filhal "gpt-4o") mein configure shuda default model istemal karega.
- **`model_settings`** (`ModelSettings`):
  - Model-specific tuning parameters (masalan, `temperature`, `top_p`) ko configure karta hai.
- **`tools`** (`list[Tool]`):
  - Woh **tools ki list** jinhein agent istemal kar sakta hai.
- **`mcp_servers`** (`list[MCPServer]`):
  - `Model Context Protocol` (MCP) servers ki list jinhein agent istemal kar sakta hai. Jab bhi agent chalta hai, toh woh in servers se tools ko available tools ki list mein shamil karega.
  - **NOTE:** Aapko in servers ke **lifecycle ko manage karna hoga**. Khaas tor par, aapko `server.connect()` call karna hoga agent ko pass karne se pehle, aur `server.cleanup()` tab call karna hoga jab server ki zaroorat na ho.
- **`mcp_config`** (`MCPConfig`):
  - MCP servers ke liye **configuration**.
- **`input_guardrails`** (`list[InputGuardrail[TContext]]`):
  - Checks ki list jo agent ke execution ke parallel mein chalti hain, jawab generate karne se pehle. Yeh sirf tab chalti hain jab agent chain mein pehla agent ho.
- **`output_guardrails`** (`list[OutputGuardrail[TContext]]`):
  - Checks ki list jo agent ke final output par chalti hain, jawab generate karne ke baad. Yeh sirf tab chalti hain jab agent aik final output paida karta hai.
- **`output_type`** (`type[Any] | AgentOutputSchemaBase | None`):
  - Output object ki type. Agar faraham na ki jaye, toh output `str` hoga. Zyada tar cases mein, aapko aik regular Python type (masalan, dataclass, Pydantic model, TypedDict, waghera) pass karni chahiye. Aap isay do tareeqon se customize kar sakte hain:
    1. Agar aap **non-strict schemas** chahte hain, toh `AgentOutputSchema`

       `(MyClass, strict_json_schema=False)` pass karein.

    2. Agar aap **custom JSON schema** (yaani SDK ke automatic schema creation ke baghair) istemal karna chahte hain, toh `AgentOutputSchemaBase` subclass karein aur pass karein.

- **`hooks`** (`AgentHooks[TContext] | None`):
  - Aik class jo is agent ke mukhtalif **lifecycle events par callbacks receive karti hai**.
- **`tool_use_behavior`** (`Literal["run_llm_again", "stop_on_first_tool"] | StopAtTools | ToolsToFinalOutputFunction`):
  - Yeh aapko tay karne deta hai ke **tools ko kaise handle kiya jaye**.
    - `"run_llm_again"`: Yeh default behavior hai. Tools chalaye jate hain, aur phir LLM nateejay receive karta hai aur jawab deta hai.

- `"stop_on_first_tool"`: Pehle tool call ka output hi final output ke tor par istemal hota hai. Iska matlab hai ke LLM tool call ke nateeje ko process nahi karta.
- `StopAtTools` (tools ke naam ki list): Agent chalna band kar dega agar list mein se koi bhi tool call hota hai. Pehle matching tool call ka output hi final output hoga. LLM tool call ke nateeje ko process nahi karta.
- `ToolsToFinalOutputFunction` (aik function): Agar aap function pass karte hain, toh woh run context aur tool results ki list ke saath call hoga. Usay `ToolsToFinalOutputResult` return karna hoga, jo tay karta hai ke tool calls se final output hasil hota hai ya nahi.

NOTE: Yeh configuration sirf `FunctionTools` ke liye hai. Hosted tools, jaisay file search, web search, waghera hamesha LLM ke zariye process hote hain.

`reset_tool_choice` (`bool`):

Kya tool call hone ke baad **tool choice ko default value par reset karna hai**. Default `True` hai. Yeh yakeeni banata hai ke agent tool usage ke infinite loop mein na phanse.

Agent Ke Methods

- **`clone(**kwargs: Any) -> Agent[TContext]`**
  - Yeh method agent ki **copy banata hai**, diye gaye arguments ko tabdeel karte huay. Masalan, aap kar sakte hain:

    Python

    ```
    new_agent = agent.clone(instructions="Naye instructions")
    ```

  - Yeh aapko aik maujooda agent ko asani se customize karne deta hai baghair asal object ko tabdeel kiye.
- **`as_tool(tool_name: str | None, tool_description: str | None, custom_output_extractor: Callable[[RunResult], Awaitable[str]] | None = None) -> Tool`**
  - Yeh method is agent ko aik **tool mein tabdeel karta hai**, jise doosre agents call kar sakte hain.
  - Yeh **handoffs se do tareeqon se mukhtalif hai**:

1. **Handoffs** mein, naya agent conversation history receive karta hai. Is **tool** mein, naya agent generated input receive karta hai.
2. **Handoffs** mein, naya agent conversation ko sambhal leta hai. Is **tool** mein, naya agent aik tool ke tor par call hota hai, aur conversation asal agent ke zariye jari rehti hai.

- **Parameters:**

Name	Type	Description	Default
tool_name	str | None	The name of the tool. If not provided, the agent's name will be used.	*required*

Name	Type	Description	Default
tool_description	str | None	The description of the tool, which should indicate what it does and when to use it.	*required*
custom_output_extractor	Callable[[RunResult], Awaitable[str]] | None	A function that extracts the output from the agent. If not provided, the last message from the agent will be used.	None

- Source code in src/agents/agent.py

- ***get_system_prompt async***

- get_system_prompt(
-     run_context: RunContextWrapper[TContext],
-) -> str | None
- Get the system prompt for the agent.
- Source code in src/agents/agent.py

- ***get_prompt async***

- get_prompt(
-     run_context: RunContextWrapper[TContext],
-) -> ResponsePromptParam | None
- Get the prompt for the agent.
- Source code in src/agents/agent.py

- ***get_mcp_tools async***

- get_mcp_tools() -> list[Tool]
- Fetches the available tools from the MCP servers.
- Source code in src/agents/agent.py

- ***get_all_tools async***

- get_all_tools(
- run_context: RunContextWrapper[Any],
-) -> list[Tool]
- All agent tools, including MCP tools and function tools.
- Source code in src/agents/agent.py

RUNNER

Runner (Chalanay Wala)

`Runner` module Agents SDK mein workflows ko chalane ke liye bunyadi tareeqay faraham karta hai. Yeh woh jagah hai jahan aap apne define kiye gaye agents aur unke interactions ko asal mein kaam karte huay dekhte hain.

`run` (async classmethod)

Yeh method aik workflow ko diye gaye agent se shuru karta hai aur usay chalata hai. Agent aik loop mein chalega jab tak aik final output paida na ho jaye.

Yeh loop is tarah kaam karta hai:

1. Agent ko diye gaye input ke saath invoke kiya jata hai.
2. Agar koi final output mil jata hai (yani agent `agent.output_type` ki qism ka kuch paida karta hai), toh loop khatam ho jata hai.
3. Agar handoff hota hai, toh hum naye agent ke saath dobara loop chalate hain.
4. Warna, hum tool calls (agar koi hain) chalate hain, aur loop ko dobara chalate hain.

Do sooraton mein, agent exception raise kar sakta hai:

1. Agar `max_turns` had se barh jaye, toh `MaxTurnsExceeded` exception raise hota hai.
2. Agar koi guardrail tripwire trigger ho jaye, toh `GuardrailTripwireTriggered` exception raise hota hai. **Note:** Sirf pehle agent ke input guardrails chalte hain.

Parameters (Args):

- `starting_agent`: Woh shuruati agent jise chalana hai.
- `input`: Agent ko diya jane wala ibtedai input. Aap user message ke liye aik single string, ya input items ki list pass kar sakte hain.
- `context`: Agent ko chalane ke liye context.
- `max_turns`: Agent ko chalane ke liye zyada se zyada turns ki tadaad. Aik turn ko aik AI invocation (jis mein koi bhi tool calls shamil ho sakte hain) ke tor par define kiya gaya hai.

- `hooks`: Aik object jo mukhtalif lifecycle events par callbacks receive karta hai.
- `run_config`: Poore agent run ke liye global settings.
- `previous_response_id`: Pichle response ki ID. Agar aap OpenAI models ko Responses API ke zariye istemal kar rahe hain, toh yeh aapko pichle turn se input pass karne se bachata hai.

Returns (Nateeja): Aik run result jis mein tamam inputs, guardrail results aur aakhri agent ka output shamil hota hai. Agents handoffs kar sakte hain, isliye hum output ki khaas type nahi jante.

`run_sync` (classmethod)

Yeh method diye gaye agent se shuru karte huay aik workflow ko **synchronously** chalata hai.

Note: Yeh sirf `run` method ko wrap karta hai, isliye agar pehle se koi event loop chal raha ho (masalan async function ke andar, ya Jupyter notebook mein, ya FastAPI jaisay async context mein), toh yeh kaam nahi karega. Un mamlaat ke liye, `run` method istemal karein.

Baaki ka behavior (`loop` aur `exceptions`) `run` method jaisa hi hai.

Parameters (Args): (Parameters `run` method jaisay hi hain)

Returns (Nateeja): (Nateeja `run` method jaisa hi hai)

`run_streamed` (classmethod)

Yeh method diye gaye agent se shuru karte huay aik workflow ko **streaming mode** mein chalata hai. Return hone wala result object aik method deta hai jise aap semantic events ko stream karne ke liye istemal kar sakte hain jaisay hi woh generate hote hain.

Baaki ka behavior (`loop` aur `exceptions`) `run` method jaisa hi hai.

Parameters (Args): (Parameters `run` method jaisay hi hain)

Returns (Nateeja): Aik result object jis mein run ke baray mein data hota hai, sath hi events ko stream karne ka method bhi.

`RunConfig` (dataclass)

`RunConfig` poore agent run ke liye settings ko configure karta hai. Yeh global control deta hai jo individual agent settings ko override kar sakta hai.

Attributes:

- **`model`** (`str | Model | None`):

  - Poore agent run ke liye istemal hone wala model. Agar set kiya jaye, toh yeh har agent par set kiye gaye model ko override karega. Neeche pass kiya gaya `model_provider` is model name ko resolve karne ke qabil hona chahiye.
- **`model_provider`** (`ModelProvider`):
  - String model names ko lookup karte waqt istemal hone wala model provider. Default `OpenAI` hai.
- **`model_settings`** (`ModelSettings | None`):
  - Global model settings configure karein. Koi bhi non-null values agent-specific model settings ko override karengi.
- **`handoff_input_filter`** (`HandoffInputFilter | None`):
  - Tamaam handoffs par lagoo hone wala global input filter. Agar `Handoff.input_filter` set kiya gaya hai, toh woh tarjeeh lega. Input filter aapko naye agent ko bheje jane wale inputs ko edit karne deta hai. Mazeed tafseelat ke liye `Handoff.input_filter` mein documentation dekhein.
- **`input_guardrails`** (`list[InputGuardrail[Any]] | None`):
  - Initial run input par chalane ke liye input guardrails ki list.
- **`output_guardrails`** (`list[OutputGuardrail[Any]] | None`):
  - Run ke final output par chalane ke liye output guardrails ki list.
- **`tracing_disabled`** (`bool`):
  - Kya agent run ke liye tracing disable hai. Agar disable ho, toh hum agent run ko trace nahi karengi. Default `False` hai.
- **`trace_include_sensitive_data`** (`bool`):
  - Kya hum traces mein mumkina tor par hassas data (masalan: tool calls ya LLM generations ke inputs/outputs) shamil karte hain. Agar `False` ho, toh hum in events ke liye spans tab bhi banayenge, lekin hassas data shamil nahi hoga. Default `True` hai.
- **`workflow_name`** (`str`):
  - Run ka naam, jo tracing ke liye istemal hota hai. Run ke liye aik logical naam hona chahiye, jaisay "Code generation workflow" ya "Customer support agent". Default `Agent workflow` hai.
- **`trace_id`** (`str | None`):
  - Tracing ke liye istemal karne wali aik custom trace ID. Agar faraham na ki jaye, toh hum aik naya trace ID generate karengi.
- **`group_id`** (`str | None`):
  - Tracing ke liye istemal hone wala aik grouping identifier, jo aik hi conversation ya process se mutadid traces ko link karta hai. Masalan, aap chat thread ID istemal kar sakte hain.
- **`trace_metadata`** (`dict[str, Any] | None`):
  - Trace ke saath shamil karne ke liye mazeed metadata ka aik ikhtiyari dictionary.

REPL

REPL Module (REPL Module)

REPL (Read-Eval-Print Loop) module aapko apne agents ko command line se asani se test aur debug karne ki sahulat deta hai.

`run_demo_loop` (async function)

Yeh function diye gaye agent ke saath aik simple REPL (Read-Eval-Print Loop) chalata hai.

Yeh utility aapko command line se aik agent ki tez manual testing aur debugging ki ijazat deti hai. Conversation ki state har turn mein mehfooz rehti hai. Loop ko rokne ke liye **`exit`** ya **`quit`** type karein.

Parameters:

- **`agent`** (`Agent[Any]`)
  - **Tafseel:** Woh shuruati agent jise chalana hai. (Zaroori)
- **`stream`** (`bool`, Default: `True`)
  - **Tafseel:** Kya agent ke output ko stream karna hai ya nahi.

TOOLS

Tools (Auzaar / Tools)

`Tools` module Agents SDK mein agents ke liye bahar ki duniya se interact karne ya khaas kaam anjaam dene ke liye mukhtalif functionalities faraham karta hai.

`MCPToolApprovalFunction` (Module Attribute)

Yeh aik function hai jo aik **tool call ko manzoor (approve) ya radd (reject)** karta hai.

`LocalShellExecutor` (Module Attribute)

Yeh aik function hai jo **shell par aik command execute** karta hai.

`Tool` (Module Attribute - Union Type)

`Tool` aik general type hai jo aik agent mein istemal ho sakne wale mukhtalif qism ke tools ko represent karta hai. Yeh in mein se koi bhi ho sakta hai:

- `FunctionTool`
- `FileSearchTool`
- `WebSearchTool`
- `ComputerTool`
- `HostedMCPTool`

- `LocalShellTool`
- `ImageGenerationTool`
- `CodeInterpreterTool`

`FunctionToolResult` (Dataclass)

Yeh dataclass tab istemal hota hai jab aik `FunctionTool` chalta hai aur uska nateeja (result) paida hota hai. Is mein yeh tafseelat hoti hain:

- **`tool`** (`FunctionTool`): Woh tool jo chala tha.
- **`output`** (`Any`): Tool ka output.
- **`run_item`** (`RunItem`): Tool call ke nateeje mein paida hone wala run item.

`FunctionTool` (Dataclass)

`FunctionTool` aik aisa tool hai jo **kisi function ko wrap** karta hai. Zyada tar mamlaat mein, aapko `function_tool` helpers ko istemal karna chahiye `FunctionTool` banane ke liye, kyunki woh aapko aasani se Python function ko wrap karne dete hain.

Attributes:

- **`name`** (`str`):
  - Tool ka naam, jaisa ke LLM ko dikhaya jata hai. Aam tor par yeh function ka naam hota hai.
- **`description`** (`str`):
  - Tool ki tafseel, jaisa ke LLM ko dikhaya jata hai.
- **`params_json_schema`** (`dict[str, Any]`):
  - Tool ke parameters ke liye JSON schema.
- **`on_invoke_tool`** (`Callable[[ToolContext[Any], str], Awaitable[Any]]`):
  - Aik function jo diye gaye context aur parameters ke saath tool ko invoke karta hai. Jo parameters pass kiye jate hain woh hain:
    1. Tool run context.
    2. LLM se arguments, aik JSON string ke tor par.
  - Aapko tool output ki string representation return karni chahiye, ya koi aisi cheez jis par hum `str()` call kar saken. Ghalati (errors) ki surat mein, aap ya toh `Exception` raise kar sakte hain (jis se run fail ho jayega) ya phir aik string error message return kar sakte hain (jo LLM ko wapas bheja jayega).
- **`strict_json_schema`** (`bool`, Default: `True`):
  - Kya JSON schema strict mode mein hai. Hum isay `True` par set karne ki **sakhti se sifarish** karte hain, kyunki yeh sahi JSON input ke imkaan ko barhata hai.
- **`is_enabled`** (`bool | Callable[[RunContextWrapper[Any], Agent[Any]], MaybeAwaitable[bool]]`, Default: `True`):
  - Kya tool enable hai. Yeh ya toh aik `bool` ho sakta hai ya aik `Callable` jo run context aur agent leta hai aur batata hai ke tool enable hai ya nahi. Aap isay dynamically (run time par) tool ko enable/disable karne ke liye istemal kar sakte hain, apne context/state ki buniyad par.

`FileSearchTool` (Dataclass)

Yeh aik **hosted tool** hai jo LLM ko vector store mein search karne deta hai. Filhal yeh sirf OpenAI models ke saath supported hai, Responses API ka istemal karte huay.

Attributes:

- **`vector_store_ids`** (`list[str]`):
  - Woh vector stores ki IDs jin mein search karna hai.
- **`max_num_results`** (`int | None`):
  - Wapas kiye jane wale results ki zyada se zyada tadaad.
- **`include_search_results`** (`bool`, Default: `False`):
  - Kya LLM ke zariye paida kiye gaye output mein search results ko shamil karna hai.
- **`ranking_options`** (`RankingOptions | None`):
  - Search ke liye ranking options.
- **`filters`** (`Filters | None`):
  - File attributes ki buniyad par lagoo karne wala filter.

`WebSearchTool` (Dataclass)

Yeh aik **hosted tool** hai jo LLM ko web search karne deta hai. Filhal yeh sirf OpenAI models ke saath supported hai, Responses API ka istemal karte huay.

Attributes:

- **`user_location`** (`UserLocation | None`):
  - Search ke liye ikhtiyari location. Yeh aapko results ko kisi khaas location se mutaliq banane deta hai.
- **`search_context_size`** (`Literal["low", "medium", "high"]`, Default: `"medium"`):
  - Search ke liye istemal hone wale context ki miqdar.

`ComputerTool` (Dataclass)

Yeh aik **hosted tool** hai jo LLM ko computer control karne deta hai.

Attributes:

- **`computer`** (`Computer | AsyncComputer`):
  - Computer implementation, jo computer ke environment aur dimensions ko bayan karti hai, sath hi click, screenshot, waghera jaisay computer actions ko implement karti hai.

`MCPToolApprovalRequest` (Dataclass)

Yeh aik request hai jo **tool call ko manzoor (approve) karne** ke liye hoti hai.

Attributes:

- **`ctx_wrapper`** (`RunContextWrapper[Any]`):
    - Run context.
- **`data`** (`McpApprovalRequest`):
    - MCP tool approval request se data.

`MCPToolApprovalFunctionResult` (TypedDict)

Yeh aik MCP tool approval function ka nateeja hai.

Attributes:

- **`approve`** (`bool`):
    - Kya tool call ko manzoor (approve) karna hai.
- **`reason`** (`NotRequired[str]`):
    - Aik ikhtiyari wajah (reason), agar reject kiya gaya ho.

`HostedMCPTool` (Dataclass)

Yeh aik tool hai jo LLM ko aik **remote MCP server istemal karne** ki ijazat deta hai. LLM khud-ba-khud tools ko list aur call karega, aapke code mein wapas jaane ki zaroorat ke baghair. Agar aap MCP servers ko locally (masalan stdio ke zariye), ya kisi VPC ya doosre non-publicly-accessible environment mein chalana chahte hain, ya aap sirf tool calls ko locally chalana pasand karte hain, toh aap `agents.mcp` mein servers ko istemal kar sakte hain aur agent ko `Agent(mcp_servers=[...])` pass kar sakte hain.

Attributes:

- **`tool_config`** (`Mcp`):
    - MCP tool config, jis mein server URL aur doosri settings shamil hain.
- **`on_approval_request`** (`MCPToolApprovalFunction | None`):
    - Aik ikhtiyari function jo tab call hoga agar MCP tool ke liye approval request ki jaye. Agar faraham na kiya jaye, toh aapko manually input mein approvals/rejections add karne honge aur `Runner.run(...)` ko dobara call karna hoga.

`CodeInterpreterTool` (Dataclass)

Yeh aik tool hai jo LLM ko **sandboxed environment mein code execute** karne ki ijazat deta hai.

Attributes:

- **`tool_config`** (`CodeInterpreter`):
    - Tool config, jis mein container aur doosri settings shamil hain.

`ImageGenerationTool` (Dataclass)

Yeh aik tool hai jo LLM ko **images generate** karne ki ijazat deta hai.

Attributes:

- **`tool_config`** (`ImageGeneration`):
  - Tool config, jis mein image generation settings shamil hain.

`LocalShellCommandRequest` (Dataclass)

Yeh shell par aik command execute karne ki request hai.

Attributes:

- **`ctx_wrapper`** (`RunContextWrapper[Any]`):
  - Run context.
- **`data`** (`LocalShellCall`):
  - Local shell tool call se data.

`LocalShellTool` (Dataclass)

Yeh aik tool hai jo LLM ko **shell par commands execute** karne ki ijazat deta hai.

Attributes:

- **`executor`** (`LocalShellExecutor`):
  - Aik function jo shell par command execute karta hai.

`default_tool_error_function`

Yeh default tool error function hai, jo sirf aik **generic error message** return karta hai.

`function_tool` (Decorator/Function)

Yeh aik **decorator** hai jo kisi function se `FunctionTool` banane ke liye istemal hota hai. By default, yeh:

1. Function signature ko parse karta hai taake tool ke parameters ke liye JSON schema banaya ja sake.
2. Function ke docstring ko tool ki tafseel ke liye istemal karta hai.
3. Function ke docstring ko argument descriptions ke liye istemal karta hai.

Docstring style khud-ba-khud detect ho jati hai, lekin aap isay override kar sakte hain.

Agar function pehle argument ke tor par `RunContextWrapper` leta hai, toh usay us agent ke context type se **match karna zaroori hai** jo is tool ko istemal karta hai.

Parameters:

- **`func`** (`ToolFunction[...] | None`):
  - Woh function jise wrap karna hai. (Optional, agar decorator ke tor par istemal na ho)
- **`name_override`** (`str | None`):
  - Agar faraham kiya jaye, toh function ke naam ki bajaye is naam ko tool ke liye istemal karein.
- **`description_override`** (`str | None`):
  - Agar faraham kiya jaye, toh function ke docstring ki bajaye is tafseel ko tool ke liye istemal karein.
- **`docstring_style`** (`DocstringStyle | None`):
  - Agar faraham kiya jaye, toh tool ke docstring ke liye is style ko istemal karein. Agar faraham na kiya jaye, toh hum style ko auto-detect karne ki koshish karengi.
- **`use_docstring_info`** (`bool`, Default: `True`):
  - Agar `True` ho, toh function ke docstring ko tool ki tafseel aur argument descriptions ke liye istemal karein.
- **`failure_error_function`** (`ToolErrorFunction | None`, Default: `default_tool_error_function`):
  - Agar faraham kiya jaye, toh is function ko istemal karein jab tool call fail ho jaye. Error message LLM ko bheja jata hai. Agar aap `None` pass karte hain, toh koi error message nahi bheja jayega aur iske bajaye `Exception` raise hoga.
- **`strict_mode`** (`bool`, Default: `True`):
  - Kya tool ke JSON schema ke liye strict mode ko enable karna hai. Hum isay `True` par set karne ki **sakhti se sifarish** karte hain, kyunki yeh sahi JSON input ke imkaan ko barhata hai. Agar `False` ho, toh yeh non-strict JSON schemas ki ijazat deta hai. Masalan, agar kisi parameter ki default value hai, toh woh optional hoga, additional properties ki ijazat hogi, waghera. Mazeed tafseelat ke liye: https://platform.openai.com/docs/guides/structured-outputs?api-mode=responses#supported-schemas
- **`is_enabled`** (`bool | Callable[[RunContextWrapper[Any], Agent[Any]], MaybeAwaitable[bool]]`, Default: `True`):
  - Kya tool enable hai. Yeh aik `bool` ho sakta hai ya aik `callable` jo run context aur agent leta hai aur batata hai ke tool enable hai ya nahi. Disabled tools runtime par LLM se chipe (hidden) hote hain.

RESULTS

Results Module (Nateeje / Results Module)

`Results` module mein woh classes aur methods shamil hain jo agent ke run ke nateejay ko handle karne aur us tak rasai hasil karne mein madad karte hain. Yeh batata hai ke run ke baad aapko kya maloomat mil sakti hain.

`RunResultBase` (Dataclass - Abstract Base Class)

Yeh agent run ke tamam nateejon ke liye aik **buniyadi abstract class** hai. Yani, baqi result classes isi par mabni hoti hain.

Attributes:

- **`input`** (`str | list[TResponseInputItem]`):
    - Woh **asal input items** jo `run()` call hone se pehle the. Yeh input ka tabdeel shuda version bhi ho sakta hai, agar koi handoff input filters input ko tabdeel karte hain.
- **`new_items`** (`list[RunItem]`):
    - Agent run ke dauran paida hone wale **naye items**. Is mein naye messages, tool calls aur unke outputs, waghera shamil hote hain.
- **`raw_responses`** (`list[ModelResponse]`):
    - Agent run ke dauran model ke zariye paida kiye gaye **raw LLM responses**.
- **`final_output`** (`Any`):
    - **Aakhri agent ka output**.
- **`input_guardrail_results`** (`list[InputGuardrailResult]`):
    - Input messages ke liye **Guardrail results**.
- **`output_guardrail_results`** (`list[OutputGuardrailResult]`):
    - Agent ke final output ke liye **Guardrail results**.
- **`context_wrapper`** (`RunContextWrapper[Any]`):
    - Agent run ke liye **context wrapper**.
- **`last_agent`** (Abstract Method Property):
    - Woh **aakhri agent jo chala tha**. Yeh aik abstract property hai, isliye isay concrete subclasses mein implement karna zaroori hai.
- **`last_response_id`** (Property):
    - Aakhri model response ki **response ID** hasil karne ka aik asaan tareeqa.

`final_output_as` (Method)

Yeh aik asaan tareeqa hai **final output ko aik khaas type mein cast karne** ke liye.

Parameters:

- **`cls`** (`type[T]`):
    - Woh type jis mein final output ko cast karna hai. (Zaroori)
- **`raise_if_incorrect_type`** (`bool`, Default: `False`):
    - Agar `True` ho, toh agar final output di gayi type ka nahi hoga toh `TypeError` raise kiya jayega.

Returns:

- **`T`**: Final output ko di gayi type mein cast kar ke wapas kiya jata hai.

`to_input_list` (Method)

Yeh aik **naya input list banata hai**, asal input ko paida kiye gaye tamam naye items ke saath merge karte huay.

`RunResult` (Dataclass)

`RunResult` aik concrete dataclass hai jo **`RunResultBase` par mabni hai**. Yeh aik non-streaming agent run ka nateeja represent karta hai. Is mein wohi attributes aur methods hain jo `RunResultBase` mein bayan kiye gaye hain.

`RunResultStreaming` (Dataclass)

`RunResultStreaming` bhi `RunResultBase` par mabni aik dataclass hai, lekin yeh **streaming mode mein agent run ka nateeja** hai. Aap `stream_events` method ko istemal kar sakte hain taake semantic events (jaise woh paida hon) ko stream kiya ja sake.

Jab streaming method chalti hai, toh yeh exceptions raise kar sakti hai:

- `MaxTurnsExceeded` exception agar agent `max_turns` limit ko paar kar jaye.
- `GuardrailTripwireTriggered` exception agar koi guardrail trigger ho jaye.

Attributes (Additional/Specific to Streaming):

- **`current_agent`** (`Agent[Any]`):
  - Woh **current agent jo chal raha hai**.
- **`current_turn`** (`int`):
  - **Current turn number**.
- **`max_turns`** (`int`):
  - Agent ke chalne ke liye **zyada se zyada turns ki tadaad**.
- **`final_output`** (`Any`):
  - Agent ka **final output**. Yeh `None` hota hai jab tak agent apna kaam mukammal nahi kar leta.
- **`is_complete`** (`bool`, Default: `False`):
  - Kya agent apna kaam mukammal kar chuka hai.
- **`last_agent`** (Property):
  - Woh **aakhri agent jo chala tha**. Yeh agent run ke barhne ke saath update hota rehta hai, isliye sahi aakhri agent sirf agent run mukammal hone ke baad hi dastiyab hota hai.

`cancel` (Method - `RunResultStreaming` ke liye)

Yeh streaming run ko **cancel** karta hai, tamam background tasks ko rokta hai aur run ko mukammal ke tor par mark karta hai.

`stream_events` (Async Method - `RunResultStreaming` ke liye)

Yeh **naye items ke deltas ko stream karta hai** jaisay woh paida hote hain. Hum OpenAI Responses API se types istemal kar rahe hain, isliye yeh **semantic events** hain: har event mein aik `type` field hota hai jo event ki type ko bayan karta hai, sath hi us event ke liye data bhi hota hai.

Yeh bhi exceptions raise kar sakta hai:

- `MaxTurnsExceeded` exception agar agent `max_turns` limit ko paar kar jaye.
- `GuardrailTripwireTriggered` exception agar koi guardrail trigger ho jaye.

Streaming events

Streaming Events (Streaming Events)

`Streaming Events` woh tareeqa hai jisse aapko agent ke chalte waqt real-time mein updates miltay hain. Jab aapka agent koi kaam kar raha hota hai, toh yeh events aapko batate hain ke kya ho raha hai, masalan naye messages, tool calls, ya agent ki tabdeeli.

`StreamEvent` (Type Alias)

Yeh aik **general type** hai jo agent se aanay wale tamam streaming events ko shamil karta hai. Yeh teen mukhtalif qism ke events mein se koi bhi ho sakta hai:

- `RawResponsesStreamEvent`
- `RunItemStreamEvent`
- `AgentUpdatedStreamEvent`

`RawResponsesStreamEvent` (Dataclass)

Yeh LLM (Large Language Model) se aanay wale **'raw' events** hote hain. Iska matlab hai ke yeh directly LLM se pass kiye jate hain, in mein koi mazeed processing nahi hoti.

- **`data`** (`TResponseStreamEvent`): LLM se aanay wala raw responses streaming event.
- **`type`** (`Literal['raw_response_event']`): Event ki qism, jo hamesha `'raw_response_event'` hoti hai.

`RunItemStreamEvent` (Dataclass)

Yeh woh streaming events hain jo aik **`RunItem` ko wrap karte hain**. Jab agent LLM response ko process karta hai, toh woh yeh events paida karta hai naye messages, tool calls, tool outputs, handoffs, waghera ke liye. Yeh aapko agent ke internal flow ki tafseeli maloomat dete hain.

- **`name`** (`Literal[...]`): Event ka naam. Is mein mukhtalif types shamil ho sakti hain jaisay:
    - `"message_output_created"` (naya message paida hua)
    - `"handoff_requested"` (handoff ki darkhwast ki gayi)
    - `"handoff_occured"` (handoff hua)
    - `"tool_called"` (tool call kiya gaya)
    - `"tool_output"` (tool ka output mila)
    - `"reasoning_item_created"` (soch-vichar ka item paida hua)

  - `"mcp_approval_requested"` (MCP approval ki darkhwast ki gayi)
  - `"mcp_list_tools"` (MCP tools ki list)
- **`item`** (`RunItem`): Woh item jo paida kiya gaya tha.

`AgentUpdatedStreamEvent` (Dataclass)

Yeh aik event hai jo yeh **ittela (notify) deta hai ke aik naya agent chal raha hai**. Yeh tab hota hai jab agent kisi doosre agent ko handoff karta hai.

- **`new_agent`** (`Agent[Any]`): Woh naya agent jo ab chal raha hai.

Handoffs

Handoffs (Ikhtiyaar Sonpna / Handoffs)

`Handoffs` woh tareeqa hai jahan aik agent apna koi kaam ya zimmedari kisi **doosre agent ko sonp deta hai**. Masalan, customer support mein, aik "triage agent" yeh tay karta hai ke user ki request ko konsa agent sambhalega, aur phir billing ya account management jaisay khaas mahir agents ko kaam sonpa jata hai.

`HandoffInputFilter` (Module Attribute)

Yeh aik **function** hai jo naye agent ko bheje jane wale **input data ko filter** karta hai.

`HandoffInputData` (Dataclass)

Yeh dataclass woh data represent karta hai jo handoff ke waqt naye agent ko input ke tor par milta hai. Is mein yeh tafseelat hoti hain:

- **`input_history`** (`str | tuple[TResponseInputItem, ...]`):
  - Woh **input history** jo `Runner.run()` call hone se pehle thi.
- **`pre_handoff_items`** (`tuple[RunItem, ...]`):
  - Woh **items jo agent turn se pehle paida hue the**, jis mein handoff invoke kiya gaya tha.
- **`new_items`** (`tuple[RunItem, ...]`):
  - **Current agent turn ke dauran paida hone wale naye items**, jis mein woh item bhi shamil hai jis ne handoff ko trigger kiya aur woh tool output message jo handoff output se response ko represent karta hai.

`Handoff` (Dataclass)

`Handoff` woh object hai jo aik agent se doosre agent ko ikhtiyaar sonpne (delegation) ko define karta hai.

Attributes:

- **`tool_name`** (`str`):
  - Us **tool ka naam** jo handoff ko represent karta hai (jaisa ke LLM ko dikhaya jata hai).
- **`tool_description`** (`str`):
  - Us **tool ki tafseel** jo handoff ko represent karta hai (jaisa ke LLM ko dikhaya jata hai).
- **`input_json_schema`** (`dict[str, Any]`):
  - Handoff input ke liye **JSON schema**. Agar handoff koi input nahi leta, toh yeh khali (empty) ho sakta hai.
- **`on_invoke_handoff`** (`Callable[[RunContextWrapper[Any], str], Awaitable[Agent[TContext]]]`):
  - Woh **function jo handoff ko invoke karta hai**. Is mein diye gaye parameters hain:
    1. Handoff run context.
    2. LLM se arguments, aik JSON string ke tor par. Agar `input_json_schema` khali ho toh yeh empty string hogi.
  - Is function ko **aik agent return karna lazmi hai**.
- **`agent_name`** (`str`):
  - Us **agent ka naam** jise ikhtiyaar sonpa ja raha hai.
- **`input_filter`** (`HandoffInputFilter | None`):
  - Aik function jo **naye agent ko bheje jane wale inputs ko filter** karta hai. By default, naya agent poori conversation history dekhta hai. Kuch mamlaat mein, aap inputs ko filter karna chahenge, masalan purane inputs ko hatane ke liye, ya maujooda inputs se tools ko remove karne ke liye.
  - Is function ko ab tak ki poori conversation history milegi, jis mein woh input item bhi shamil hoga jis ne handoff ko trigger kiya aur aik tool call output item jo handoff tool ke output ko represent karta hai.
  - Aap input history ya naye items ko apni marzi ke mutabiq tabdeel kar sakte hain. Agla agent jo chalega usay `handoff_input_data.all_items` milega.
  - **IMPORTANT:** Streaming mode mein, is function ke nateeje mein hum kuch bhi stream nahi karengi. Jo items pehle paida ho chuke honge woh pehle hi stream ho chuke honge.
- **`strict_json_schema`** (`bool`, Default: `True`):
  - Kya input JSON schema strict mode mein hai. Hum isay `True` par set karne ki **sakhti se sifarish** karte hain, kyunki yeh sahi JSON input ke imkaan ko barhata hai.

`handoff` (Function/Decorator)

Yeh function aik agent se **handoff banane** ke liye istemal hota hai. Yeh kayi mukhtalif tareeqon se istemal kiya ja sakta hai (overloaded signatures).

Parameters:

- **`agent`** (`Agent[TContext]`):
  - Woh **agent jise ikhtiyaar sonpna hai**, ya aik function jo agent return karta hai. (Zaroori)
- **`tool_name_override`** (`str | None`):
  - Handoff ko represent karne wale tool ke naam ke liye **ikhtiyari override**.
- **`tool_description_override`** (`str | None`):
  - Handoff ko represent karne wale tool ki tafseel ke liye **ikhtiyari override**.
- **`on_handoff`** (`OnHandoffWithInput[THandoffInput] | OnHandoffWithoutInput | None`):
  - Aik **function jo tab chalta hai jab handoff invoke hota hai**.
- **`input_type`** (`type[THandoffInput] | None`):

  - Handoff ke input ki type. Agar faraham ki jaye, toh input ko is type ke mutabiq validate kiya jayega. Yeh sirf tab relevant hai agar aap aik function pass karte hain jo input leta hai.
- **`input_filter`** (`Callable[[HandoffInputData], HandoffInputData] | None`):
  - Aik function jo naye agent ko bheje jane wale inputs ko filter karta hai.

Lifecycle

Lifecycle (Zindagi Ka Daur)

`Lifecycle` se murad agent ke chalne ke mukammal dauran hone wale mukhtalif marhalay (stages) aur events hain. Agents SDK aapko in events par respond karne ki ijazat deta hai taake aap customization, logging, ya monitoring jaisay kaam kar saken. Yeh do ahem classes ke zariye kiya jata hai: `RunHooks` aur `AgentHooks`.

`RunHooks`

Yeh aik aisi class hai jo **poore agent run mein mukhtalif lifecycle events par callbacks receive karti hai**. Aap is class ko subclass kar ke apni zaroorat ke mutabiq methods ko override kar sakte hain. Jab aap ise `Runner.run()` method mein pass karte hain, toh yeh poore workflow (chahe kitne bhi agents involve hon) ke events ko monitor karta hai.

- **`on_agent_start`** (async method)
  - Yeh method **agent ke invoke hone se pehle** call hota hai. Jab bhi current agent tabdeel hota hai (yaani naya agent chalna shuru karta hai), tab yeh method call hota hai.
  - **Parameters:** `context` (run context), `agent` (jo agent shuru ho raha hai).
- **`on_agent_end`** (async method)
  - Yeh method tab call hota hai jab **agent apna final output paida karta hai**.
  - **Parameters:** `context` (run context), `agent` (jo agent khatam ho raha hai), `output` (agent ka final output).
- **`on_handoff`** (async method)
  - Yeh method tab call hota hai jab **handoff hota hai**. Yani, jab aik agent apna kaam kisi doosre agent ko sonp deta hai.
  - **Parameters:** `context` (run context), `from_agent` (woh agent jo kaam sonp raha hai), `to_agent` (woh agent jise kaam sonpa ja raha hai).
- **`on_tool_start`** (async method)
  - Yeh method **tool ke invoke hone se pehle** call hota hai.
  - **Parameters:** `context` (run context), `agent` (jis agent ne tool ko call kiya), `tool` (woh tool jo chalne wala hai).
- **`on_tool_end`** (async method)
  - Yeh method **tool ke invoke hone ke baad** call hota hai.
  - **Parameters:** `context` (run context), `agent` (jis agent ne tool ko call kiya), `tool` (woh tool jo chala), `result` (tool ka output aik string ke tor par).

`AgentHooks`

Yeh aik aisi class hai jo **kisi khaas agent ke lifecycle events par callbacks receive karti hai**. Aap ise `agent.hooks` attribute par set kar sakte hain taake sirf us khaas agent se mutaliq events hasil kar saken. Jabke `RunHooks` poore run ko monitor karta hai, `AgentHooks` aik individual agent par focused hota hai.

- **`on_start`** (async method)
  - Yeh method **agent ke invoke hone se pehle** call hota hai. Yeh har bar tab call hota hai jab running agent ko is agent mein tabdeel kiya jata hai.
- **`on_end`** (async method)
  - Yeh method tab call hota hai jab **agent apna final output paida karta hai**.
- **`on_handoff`** (async method)
  - Yeh method tab call hota hai jab **is agent ko ikhtiyaar sonpa ja raha ho**. Yani, jab koi doosra agent is agent ko kaam deta hai.
  - **Parameters:** `context` (run context), `agent` (yeh khud agent), `source` (woh agent jo is agent ko ikhtiyaar sonp raha hai).
- **`on_tool_start`** (async method)
  - Yeh method **tool ke invoke hone se pehle** call hota hai (agar yeh agent tool call kar raha ho).
- **`on_tool_end`** (async method)
  - Yeh method **tool ke invoke hone ke baad** call hota hai (agar yeh agent tool call kar raha ho).

`RunHooks` aur `AgentHooks` aapko apne agent workflows ke behaviour ko zyada gehrayi se control aur observe karne ki sahulat dete hain.

Items

Items (Ashya / Items)

`Items` module agent ke run ke dauran paida hone wali mukhtalif qism ki maloomat (data entities) ko define karta hai. Yeh messages, tool calls, unke outputs, aur dusri events ko represent karte hain, jo agent ke andarونی amal (internal workings) aur interactions ko zahir karte hain.

Type Aliases (Types ke Mukhtasar Naam)

Yeh kuch mukhtasar naam hain jo OpenAI SDK se mukhtalif types ko refer karte hain:

- **`TResponse`**: `Response` type ke liye mukhtasar naam.
- **`TResponseInputItem`**: `ResponseInputItemParam` type ke liye mukhtasar naam.
- **`TResponseOutputItem`**: `ResponseOutputItem` type ke liye mukhtasar naam.
- **`TResponseStreamEvent`**: `ResponseStreamEvent` type ke liye mukhtasar naam.

`ToolCallItemTypes` (Type Alias)

Yeh aik type hai jo **tool call item ki mukhtalif aqsaam** ko represent karta hai. Yani, jab agent koi tool call karta hai, toh woh in mein se kisi bhi type ka ho sakta hai:

- `ResponseFunctionToolCall`
- `ResponseComputerToolCall`
- `ResponseFileSearchToolCall`
- `ResponseFunctionWebSearch`
- `McpCall`
- `ResponseCodeInterpreterToolCall`
- `ImageGenerationCall`
- `LocalShellCall`

`RunItem` (Type Alias)

`RunItem` **aik agent ke zariye paida hone wala koi bhi item** ho sakta hai. Yeh aik "union type" hai, jis ka matlab hai ke yeh in mein se kisi bhi type ka ho sakta hai:

- `MessageOutputItem` (LLM se message)
- `HandoffCallItem` (handoff ka tool call)
- `HandoffOutputItem` (handoff ka nateeja)
- `ToolCallItem` (tool ka call)
- `ToolCallOutputItem` (tool call ka nateeja)
- `ReasoningItem` (agent ki soch-vichar)
- `MCPListToolsItem` (MCP server se tools list karne ka call)
- `MCPApprovalRequestItem` (MCP approval ki darkhwast)
- `MCPApprovalResponseItem` (MCP approval darkhwast ka jawab)

`RunItemBase` (Dataclass - Abstract Base Class)

Yeh tamam `RunItem` types ke liye aik **buniyadi abstract class** hai. Har `RunItem` isi par mabni hota hai.

Attributes:

- **`agent`** (`Agent[Any]`):
  - Woh **agent jis ke run ki wajah se yeh item paida hua** hai.
- **`raw_item`** (`T`):
  - Run se hasil hone wala **raw Responses item**. Yeh hamesha ya toh output item (`openai.types.responses.ResponseOutputItem`) hoga ya input item (`openai.types.responses.ResponseInputItemParam`).

Methods:

- **`to_input_item() -> TResponseInputItem`**:
  - Is item ko **input item mein tabdeel karta hai**, jo model ko pass karne ke liye munasib ho.

`MessageOutputItem` (Dataclass)

Yeh LLM (Large Language Model) se aanay wale **message ko represent** karta hai.

- **`raw_item`** (`ResponseOutputMessage`): Raw response output message.
- **(Inherits `agent` and `to_input_item` from `RunItemBase`)**

`HandoffCallItem` (Dataclass)

Yeh aik agent se doosre agent ko hone wale **handoff ke liye tool call ko represent** karta hai.

- **`raw_item`** (`ResponseFunctionToolCall`): Handoff ko represent karne wala raw response function tool call.
- **(Inherits `agent` and `to_input_item` from `RunItemBase`)**

`HandoffOutputItem` (Dataclass)

Yeh **handoff ke output ko represent** karta hai.

- **`raw_item`** (`TResponseInputItem`): Woh raw input item jo handoff ke hone ko represent karta hai.
- **`source_agent`** (`Agent[Any]`): Woh **agent jis ne handoff kiya**.
- **`target_agent`** (`Agent[Any]`): Woh **agent jise ikhtiyaar sonpa ja raha hai**.
- **(Inherits `agent` and `to_input_item` from `RunItemBase`)**

`ToolCallItem` (Dataclass)

Yeh aik **tool call ko represent** karta hai, masalan aik function call ya computer action call.

- **`raw_item`** (`ToolCallItemTypes`): Raw tool call item.
- **(Inherits `agent` and `to_input_item` from `RunItemBase`)**

`ToolCallOutputItem` (Dataclass)

Yeh **tool call ke output ko represent** karta hai.

- **`raw_item`** (`FunctionCallOutput | ComputerCallOutput | LocalShellCallOutput`): Model se aanay wala raw item.
- **`output`** (`Any`): Tool call ka output. Yeh woh value hai jo tool call ne return ki; `raw_item` mein output ki string representation hoti hai.
- **(Inherits `agent` and `to_input_item` from `RunItemBase`)**

`ReasoningItem` (Dataclass)

Yeh **agent ki soch-vichar (reasoning) item ko represent** karta hai.

- **`raw_item`** (`ResponseReasoningItem`): Raw reasoning item.
- **(Inherits `agent` and `to_input_item` from `RunItemBase`)**

`MCPListToolsItem` (Dataclass)

Yeh **MCP server ko tools list karne ke liye call ko represent** karta hai.

- **`raw_item`** (`McpListTools`): Raw MCP list tools call.
- **(Inherits `agent` and `to_input_item` from `RunItemBase`)**

`MCPApprovalRequestItem` (Dataclass)

Yeh **MCP approval ke liye darkhwast (request) ko represent** karta hai.

- **`raw_item`** (`McpApprovalRequest`): Raw MCP approval request.
- **(Inherits `agent` and `to_input_item` from `RunItemBase`)**

`MCPApprovalResponseItem` (Dataclass)

Yeh **MCP approval request ke jawab (response) ko represent** karta hai.

- **`raw_item`** (`McpApprovalResponse`): Raw MCP approval response.
- **(Inherits `agent` and `to_input_item` from `RunItemBase`)**

`ModelResponse` (Dataclass)

Yeh LLM (Large Language Model) se aanay wale mukammal jawab (response) ko represent karta hai.

- **`output`** (`list[TResponseOutputItem]`): Model ke zariye paida kiye gaye outputs (messages, tool calls, waghera) ki list.
- **`usage`** (`Usage`): Response ke liye usage ki maloomat.
- **`response_id`** (`str | None`): Response ke liye aik ID jise model ko baad ke calls mein refer karne ke liye istemal kiya ja sakta hai. Tamam model providers isay support nahi karte. Agar OpenAI models ko Responses API ke zariye istemal kiya ja raha hai, toh yeh `response_id` parameter hai, aur isay `Runner.run` ko pass kiya ja sakta hai.

Methods:

- `to_input_items() -> list[TResponseInputItem]`:
    - Output ko **input items ki list mein tabdeel karta hai**, jo model ko pass karne ke liye munasib ho.

`ItemHelpers` (Class)

Yeh class `RunItem` objects ko process karne aur unse maloomat nikalne ke liye madadgar (utility) methods faraham karti hai.

- `extract_last_content(message: TResponseOutputItem) -> str`:
    - Aik message se **aakhri text content ya refusal extract** karta hai.
- `extract_last_text(message: TResponseOutputItem) -> str | None`:
    - Aik message se **aakhri text content extract** karta hai, agar koi ho. Refusals ko nazar andaz karta hai.
- `input_to_new_input_list(input: str | list[TResponseInputItem]) -> list[TResponseInputItem]`:
    - Aik string ya input items ki list ko **input items ki list mein tabdeel** karta hai.
- `text_message_outputs(items: list[RunItem]) -> str`:
    - Message output items ki list se **tamam text content ko yakja (concatenate) karta hai**.
- `text_message_output(message: MessageOutputItem) -> str`:
    - Aik single message output item se **tamam text content extract** karta hai.
- `tool_call_output_item(tool_call: ResponseFunctionToolCall, output: str) -> FunctionCallOutput`:
    - Aik **tool call aur uske output se tool call output item banata hai**.

Run context

RunContextWrapper (Run Context Wrapper)

`RunContextWrapper` aik **dataclass** hai jo aapke diye gaye **context object ko wrap karta hai**, jo aap `Runner.run()` ko pass karte hain. Is mein agent run ke dauran ab tak ke istemal (usage) ki maloomat bhi shamil hoti hai.

Zaroori Baat: Contexts LLM (Large Language Model) ko pass nahi kiye jate. Yeh sirf aapke implement kiye gaye code (jaise tool functions, callbacks, hooks, waghera) ko dependencies aur data pass karne ka aik tareeqa hain.

Attributes

- `context` (`TContext`):
    - Yeh woh **context object** hai (ya `None` ho sakta hai), jo aapne khud `Runner.run()` method ko diya tha. Aap is mein koi bhi data ya objects rakh sakte hain jinhein aap apni tool functions, guardrails, ya hooks mein istemal karna chahte hain.
- `usage` (`Usage`, Default: `Usage()`):
    - Yeh **agent run ka ab tak ka istemal (usage)** batata hai (masalan, tokens ki tadaad).

  - Streaming responses ke liye, `usage` ki maloomat tab tak **purani (stale)** rahegi jab tak stream ka aakhri chunk process na ho jaye. Jab stream mukammal ho jayegi, tab yeh mutadid (accurate) ho jayega.

`RunContextWrapper` aapko apne agent ke andar aik **dynamic aur mutable state** banaye rakhne mein madad karta hai, jise aapke mukhtalif components access aur update kar sakte hain.

Usage

Usage (Istemal)

`Usage` dataclass aapke agent run ke dauran Large Language Model (LLM) API ke **istemal ki tafseelat** ko track karta hai. Yeh aapko ye samajhne mein madad karta hai ke aapka agent kitne resources (jaise tokens aur requests) istemal kar raha hai.

Attributes

- **`requests`** (`int`, Default: `0`):
  - LLM API ko ki gayi **total requests** ki tadaad. Har baar jab agent LLM se baat karta hai, toh yeh count barhta hai.
- **`input_tokens`** (`int`, Default: `0`):
  - Tamaam requests mein bheje gaye **total input tokens** ki tadaad. Input tokens woh hain jo aap LLM ko sawal ya context ke tor par bhejte hain.
- **`input_tokens_details`** (`InputTokensDetails`):
  - Input tokens ke baray mein **mazeed tafseelat**, jo OpenAI Responses API ke usage details se mutabiqat rakhti hain. Is mein filhal `cached_tokens` shamil hain.
  - **`cached_tokens`**: Cache se istemal hone wale tokens ki tadaad.
- **`output_tokens`** (`int`, Default: `0`):
  - Tamaam requests mein receive kiye gaye **total output tokens** ki tadaad. Output tokens woh hain jo LLM aapko jawab ke tor par wapas karta hai.
- **`output_tokens_details`** (`OutputTokensDetails`):
  - Output tokens ke baray mein **mazeed tafseelat**, jo OpenAI Responses API ke usage details se mutabiqat rakhti hain. Is mein filhal `reasoning_tokens` shamil hain.
  - **`reasoning_tokens`**: Woh tokens jo LLM ne apni soch-vichar (reasoning) ke liye istemal kiye.
- **`total_tokens`** (`int`, Default: `0`):
  - Tamaam requests mein bheje aur receive kiye gaye **total tokens** (input + output) ki tadaad.

`Usage` object aapko apne agent ki performance aur cost ko monitor karne mein madad karta hai.

Exceptions

Exceptions (Istisnaat / Exceptions)

`Exceptions` module Agent SDK mein hone wali mukhtalif qism ki ghaltiyon (errors) ko handle karne aur unhein bayan karne ke liye classes faraham karta hai. Jab agent run ke dauran koi anchaahi (unexpected) surat-e-haal paish aati hai, toh yeh exceptions raise hoti hain taake aap usay theek tareeqay se manage kar saken.

`RunErrorDetails` (Dataclass)

Yeh dataclass **agent run ke dauran exception hone par jama ki gayi maloomat** ko store karta hai. Yeh exception ke baray mein mazeed tafseelat faraham karta hai.

`AgentsException` (Base Class)

Yeh Agent SDK mein **tamam exceptions ke liye buniyadi class** hai. Jitne bhi khaas exceptions neeche bayan kiye gaye hain, woh sab isi class se inherit karte hain.

`MaxTurnsExceeded` (Exception)

Yeh exception tab raise hota hai jab agent run ke dauran **turns ki zyada se zyada tadaad (maximum number of turns) paar ho jati hai**. Yani, agent apni di gayi turn limit ke andar kaam mukammal nahi kar pata.

`ModelBehaviorError` (Exception)

Yeh exception tab raise hota hai jab **model koi anchaahi harkat (unexpected behavior) karta hai**, masalan:

- Kisi aise tool ko call karna jo maujood nahi hai.
- Ghalat JSON format faraham karna (malformed JSON).
- Ya koi bhi aisi harkat jo model se mutawaqqi na ho.

`UserError` (Exception)

Yeh exception tab raise hota hai jab **user SDK istemal karte waqt koi ghalti karta hai**. Misal ke tor par, galat parameters pass karna ya SDK ki guidelines ki khilaf warzi karna.

`InputGuardrailTripwireTriggered` (Exception)

Yeh exception tab raise hota hai jab **input guardrail ka tripwire trigger ho jata hai**. Yani, agent ko diye gaye input ne kisi security, policy, ya dusri guardrail shart (condition) ki khilaf warzi ki ho.

- **`guardrail_result`**: Us guardrail ka result data jo trigger hua tha.

`OutputGuardrailTripwireTriggered` (Exception)

Yeh exception tab raise hota hai jab **output guardrail ka tripwire trigger ho jata hai**. Yani, agent ke paida kiye gaye final output ne kisi guardrail shart ki khilaf warzi ki ho.

- **`guardrail_result`**: Us guardrail ka result data jo trigger hua tha.

Yeh exceptions aapko agent run ke dauran hone wali ghaltiyon ko pehchanne, unhein debug karne, aur munasib tareeqay se handle karne mein madad karti hain.

Guardrails

Guardrails (Nigehbani / Guardrails)

`Guardrails` Agent SDK mein woh check mechanisms hain jo agent ke execution ke dauran input aur output par control rakhte hain. Yeh aapko security, content moderation, ya khaas business rules ko enforce karne ki ijazat dete hain. Agar koi guardrail "trigger" ho jaye, toh agent ka execution rooka ja sakta hai.

`GuardrailFunctionOutput` (Dataclass)

Yeh aik **guardrail function ka output** hai. Har guardrail function ko yahi type return karni chahiye.

- **`output_info`** (`Any`):
  - Guardrail ke output ke bare mein **ikhtiyari maloomat**. Masalan, guardrail un checks ke bare mein maloomat shamil kar sakti hai jo isne kiye the aur unke mukammal nateejay.
- **`tripwire_triggered`** (`bool`):
  - Kya **tripwire trigger hua tha**. Agar `True` ho, toh agent ka execution fauran ruk jayega aur aik exception raise hoga.

`InputGuardrailResult` (Dataclass)

Yeh **input guardrail run ka nateeja** hai.

- **`guardrail`** (`InputGuardrail[Any]`):
  - Woh **input guardrail jo chala tha**.
- **`output`** (`GuardrailFunctionOutput`):
  - **Guardrail function ka output** (jis mein `tripwire_triggered` ki maloomat shamil hoti hai).

`OutputGuardrailResult` (Dataclass)

Yeh **output guardrail run ka nateeja** hai.

- **`guardrail`** (`OutputGuardrail[Any]`):
  - Woh **output guardrail jo chala tha**.
- **`agent_output`** (`Any`):
  - **Agent ka output** jise guardrail ne check kiya tha.
- **`agent`** (`Agent[Any]`):
  - Woh **agent jise guardrail ne check kiya tha**.
- **`output`** (`GuardrailFunctionOutput`):
  - **Guardrail function ka output**.

`InputGuardrail` (Dataclass)

`Input guardrails` woh checks hain jo **agent ke execution ke mutawazi (parallel) chalte hain**. Inhein mukhtalif maqasid ke liye istemal kiya ja sakta hai, masalan:

- Yeh check karna ke kya input messages off-topic hain.
- Agar koi anchaaha input mil jaye toh agent ke execution ka control sambhaal lena.

Aap `@input_guardrail()` decorator ko istemal kar ke kisi function ko `InputGuardrail` mein tabdeel kar sakte hain, ya phir `InputGuardrail` ko manually bana sakte hain.

Guardrails aik `GuardrailFunctionOutput` return karte hain. Agar `result.tripwire_triggered` `True` ho, toh agent execution fauran ruk jayega aur aik `InputGuardrailTripwireTriggered` exception raise hoga.

Attributes:

- **`guardrail_function`** (`Callable[...]`):
  - Aik function jo **agent input aur context receive karta hai**, aur aik `GuardrailFunctionOutput` return karta hai. Nateeja batata hai ke kya tripwire trigger hua tha, aur ikhtiyari tor par guardrail ke output ke baray mein maloomat shamil kar sakta hai.
- **`name`** (`str | None`):
  - Guardrail ka naam, jo tracing ke liye istemal hota hai. Agar faraham na kiya jaye, toh hum guardrail function ka naam istemal karengi.

`OutputGuardrail` (Dataclass)

`Output guardrails` woh checks hain jo **agent ke final output par chalte hain**. Inhein yeh check karne ke liye istemal kiya ja sakta hai ke kya output kuch validation criteria ko pura karta hai.

Aap `@output_guardrail()` decorator ko istemal kar ke kisi function ko `OutputGuardrail` mein tabdeel kar sakte hain, ya phir `OutputGuardrail` ko manually bana sakte hain.

Guardrails aik `GuardrailFunctionOutput` return karte hain. Agar `result.tripwire_triggered` `True` ho, toh aik `OutputGuardrailTripwireTriggered` exception raise hoga.

Attributes:

- **`guardrail_function`** (`Callable[...]`):
  - Aik function jo **final agent, uska output, aur context receive karta hai**, aur aik `GuardrailFunctionOutput` return karta hai. Nateeja batata hai ke kya tripwire trigger hua tha, aur ikhtiyari tor par guardrail ke output ke baray mein maloomat shamil kar sakta hai.
- **`name`** (`str | None`):
  - Guardrail ka naam, jo tracing ke liye istemal hota hai. Agar faraham na kiya jaye, toh hum guardrail function ka naam istemal karengi.

`input_guardrail` (Decorator)

Yeh aik decorator hai jo aik sync ya async function ko **`InputGuardrail` mein tabdeel** karta hai. Isay barah-e-raast (direct) bhi istemal kiya ja sakta hai ya keyword arguments ke saath bhi, masalan:

Python

```
@input_guardrail
def my_sync_guardrail(...):
    ...

@input_guardrail(name="guardrail_name")
async def my_async_guardrail(...):
    ...
```

`output_guardrail` (Decorator)

Yeh aik decorator hai jo aik sync ya async function ko **`OutputGuardrail` mein tabdeel** karta hai. Isay barah-e-raast (direct) bhi istemal kiya ja sakta hai ya keyword arguments ke saath bhi, masalan:

Python

```
@output_guardrail
def my_sync_guardrail(...):
    ...

@output_guardrail(name="guardrail_name")
async def my_async_guardrail(...):
    ...
```

Guardrails aapke agents ko ziada robust aur safe banane ke liye aik powerful feature hain.

Model settings

ModelSettings (Model Settings)

`ModelSettings` dataclass mein woh **ikhtiyari (optional) configuration parameters** hote hain jo Large Language Model (LLM) ko call karte waqt istemal hote hain. Jaise ke temperature, top_p, penalties, truncation, waghera.

Yeh baat zehen mein rakhna zaroori hai ke tamam models/providers in parameters ko support nahi karte, isliye aap jis khaas model aur provider ko istemal kar rahe hain, uski API documentation check kar len.

Attributes

- **`temperature`** (`float | None`):
  - Model ko call karte waqt istemal hone wala **temperature**. Yeh model ki creativity aur randomness ko control karta hai. Higher values zyada random output deti hain.
- **`top_p`** (`float | None`):
  - Model ko call karte waqt istemal hone wala **top_p value**. Yeh bhi output ki randomness ko control karta hai, lekin top probability mass ki buniyad par.
- **`frequency_penalty`** (`float | None`):
  - Model ko call karte waqt istemal hone wali **frequency penalty**. Yeh un tokens ki frequency ko control karta hai jo LLM ne pehle generate kiye hain. Zyada value baar baar aane wale tokens ko kam karti hai.
- **`presence_penalty`** (`float | None`):
  - Model ko call karte waqt istemal hone wali **presence penalty**. Yeh is baat ko control karta hai ke LLM naye topics ya concepts kitni baar introduce karta hai. Zyada value naye topics ko encourage karti hai.
- **`tool_choice`** (`Literal["auto", "required", "none"] | str | None`):
  - Model ko call karte waqt istemal hone wala **tool choice**. Yeh control karta hai ke LLM tools ko kaise istemal karega:
    - `"auto"`: Model khud decide karega.
    - `"required"`: Model ko tool call karna lazmi hai.
    - `"none"`: Model koi tool call nahi karega.
    - Aap kisi khaas tool ka naam bhi de sakte hain.
- **`parallel_tool_calls`** (`bool | None`):
  - Kya model ko call karte waqt **parallel tool calls istemal karne hain**. Agar faraham nahi kiya gaya toh default `False` hota hai.
- **`truncation`** (`Literal['auto', 'disabled'] | None`):
  - Model ko call karte waqt istemal hone wali **truncation strategy**.
    - `'auto'`: Model khud content ko truncate karega agar woh context window se barh jaye.
    - `'disabled'`: Truncation nahi ki jayegi.
- **`max_tokens`** (`int | None`):
  - Paida kiye jane wale **output tokens ki zyada se zyada tadaad**.
- **`reasoning`** (`Reasoning | None`):
  - **Reasoning models ke liye configuration options**.

- **`metadata`** (`dict[str, str] | None`):
  - Model response call ke saath shamil karne ke liye **metadata**.
- **`store`** (`bool | None`):
  - Kya generated model response ko baad mein retrieve karne ke liye **store karna hai**. Agar faraham nahi kiya gaya toh default `True` hota hai.
- **`include_usage`** (`bool | None`):
  - Kya usage chunk ko **shamil karna hai**. Agar faraham nahi kiya gaya toh default `True` hota hai.
- **`extra_query`** (`Query | None`):
  - Request ke saath faraham karne ke liye **additional query fields**. Agar faraham nahi kiya gaya toh default `None` hota hai.
- **`extra_body`** (`Body | None`):
  - Request ke saath faraham karne ke liye **additional body fields**. Agar faraham nahi kiya gaya toh default `None` hota hai.
- **`extra_headers`** (`Headers | None`):
  - Request ke saath faraham karne ke liye **additional headers**. Agar faraham nahi kiya gaya toh default `None` hota hai.
- **`extra_args`** (`dict[str, Any] | None`):
  - Model API call ko pass karne ke liye **arbitrary keyword arguments**. Yeh directly underlying model provider ki API ko pass kiye jayenge. Ehtiyat se istemal karein kyunki tamam models tamam parameters ko support nahi karte.

`resolve` (Method)

- **`resolve(override: ModelSettings | None) -> ModelSettings`**:
  - Is instance ke **non-None values ko override ke non-None values ke upar overlay karke** aik naya `ModelSettings` object paida karta hai. Yani, agar `override` mein koi setting set hai, toh woh is instance ki setting ko override kar degi.

`ModelSettings` aapko apne agents ke behaviour ko LLM level par barik-bini se control karne ki sahulat deta hai.

Agent output

Agent Output (Agent Output)

`Agent Output` module is baat ko define karta hai ke agent se kis qism ka output mutawaqqi hai aur us output ko validate aur parse kaise kiya jaye. Yeh is baat ko yakeeni banata hai ke LLM (Large Language Model) se milne wala data sahi format aur structure mein ho.

`AgentOutputSchemaBase` (Abstract Base Class)

Yeh output ke JSON schema ko capture karne, aur LLM se paida hone wale JSON ko output type mein validate/parse karne ke liye aik **buniyadi abstract class** hai.

Abstract Methods (Jinhein subclass mein implement karna lazmi hai):

- **`is_plain_text() -> bool`**:
  - Kya output type **plain text hai (ya JSON object)**?
- **`name() -> str`**:
  - Output type ka **naam**.
- **`json_schema() -> dict[str, Any]`**:
  - Output ka **JSON schema return karta hai**. Yeh method sirf tab call hoga agar output type plain text nahi ho.
- **`is_strict_json_schema() -> bool`**:
  - Kya JSON schema **strict mode mein hai**? Strict mode JSON schema features ko mehdood karta hai, lekin sahi JSON ki zamanat deta hai. Mazeed tafseelat ke liye: https://platform.openai.com/docs/guides/structured-outputs#supported-schemas
- **`validate_json(json_str: str) -> Any`**:
  - Aik JSON string ko output type ke mukabil **validate karta hai**. Aapko validated object return karna hoga, ya agar JSON invalid ho toh `ModelBehaviorError` raise karna hoga.

`AgentOutputSchema` (Dataclass)

`AgentOutputSchema` aik concrete dataclass hai jo `AgentOutputSchemaBase` se inherit karta hai. Yeh LLM ke zariye paida hone wale JSON ko output type mein validate aur parse karta hai.

Attributes:

- **`output_type`** (`type[Any]`):
  - Output ki **type**.

Constructor (`__init__`) Parameters:

- **`output_type`** (`type[Any]`):
  - Output ki type. (Zaroori)
- **`strict_json_schema`** (`bool`, Default: `True`):
  - Kya JSON schema strict mode mein ho. Hum isay `True` par set karne ki **sakhti se sifarish** karte hain, kyunki yeh sahi JSON input ke imkaan ko barhata hai.

Methods (Implementations of `AgentOutputSchemaBase` methods):

- **`is_plain_text() -> bool`**:
  - Kya output type plain text hai (ya JSON object).
- **`is_strict_json_schema() -> bool`**:
  - Kya JSON schema strict mode mein hai.
- **`json_schema() -> dict[str, Any]`**:
  - Output type ka JSON schema.
- **`validate_json(json_str: str) -> Any`**:
  - Aik JSON string ko output type ke mukabil validate karta hai. Validated object return karta hai, ya agar JSON invalid ho toh `ModelBehaviorError` raise karta hai.

- `name() -> str`:
  - Output type ka naam.

`AgentOutputSchema` ka istemal is baat ko yakeeni banane ke liye hota hai ke agent se milne wala output aapke mutawaqqi format aur structure mein ho, khaas tor par jab aap structured data (JSON) ke saath kaam kar rahe hon.

Function schema

Function Schema (Function Schema)

`Function Schema` module is baat se mutaliq hai ke aik **Python function ki schema ko kaise capture kiya jaye**, taake use LLM (Large Language Model) ko aik tool ke tor par bhejne ke liye tayar kiya ja sake. Yeh LLM ko batata hai ke function kya karta hai, kin parameters ko leta hai, aur un parameters ki types kya hain.

`FuncSchema` (Dataclass)

Yeh dataclass aik **Python function ke schema ko capture** karta hai.

Attributes:

- **`name`** (`str`):
  - Function ka **naam**.
- **`description`** (`str | None`):
  - Function ki **tafseel**.
- **`params_pydantic_model`** (`type[BaseModel]`):
  - Aik **Pydantic model** jo function ke parameters ko represent karta hai. Yeh type validation aur parsing ke liye istemal hota hai.
- **`params_json_schema`** (`dict[str, Any]`):
  - Function ke parameters ke liye **JSON schema**, jo Pydantic model se hasil kiya gaya hai. LLM is schema ko samajh kar sahi parameters generate karta hai.
- **`signature`** (`Signature`):
  - Function ka **signature** (yani parameters, return type, waghera ki maloomat).
- **`takes_context`** (`bool`, Default: `False`):
  - Kya function **`RunContextWrapper` argument leta hai** (agar leta hai, toh yeh pehla argument hona chahiye).
- **`strict_json_schema`** (`bool`, Default: `True`):
  - Kya JSON schema **strict mode mein hai**. Hum isay `True` par set karne ki **sakhti se sifarish** karte hain, kyunki yeh sahi JSON input ke imkaan ko barhata hai.

Methods:

- **`to_call_args(data: BaseModel) -> tuple[list[Any], dict[str, Any]]`**:
  - Pydantic model se validated data ko `(args, kwargs)` mein tabdeel karta hai, jo asal function ko call karne ke liye munasib ho.

`FuncDocumentation` (Dataclass)

Yeh dataclass aik **Python function ke baray mein metadata** shamil karta hai, jo uske docstring se nikala gaya hai.

Attributes:

- **`name`** (`str`):
  - Function ka naam, `__name__` ke zariye.
- **`description`** (`str | None`):
  - Function ki tafseel, docstring se hasil shuda.
- **`param_descriptions`** (`dict[str, str] | None`):
  - Function ke parameter ki tafseelat, docstring se hasil shuda.

`generate_func_documentation` (Function)

Yeh function **kisi function ke docstring se metadata extract** karta hai, taake use LLM ko aik tool ke tor par bhejne ke liye tayar kiya ja sake.

Parameters:

- **`func`** (`Callable[..., Any]`):
  - Woh function jis se documentation extract karni hai. (Zaroori)
- **`style`** (`DocstringStyle | None`):
  - Parsing ke liye istemal hone wale docstring ka style. Agar faraham na kiya jaye, toh hum style ko auto-detect karne ki koshish karengi.

Returns:

- **`FuncDocumentation`**: Aik `FuncDocumentation` object jis mein function ka naam, tafseel, aur parameter ki tafseelat shamil hoti hain.

`function_schema` (Function)

Yeh function **diye gaye Python function se aik `FuncSchema` extract** karta hai, jis mein naam, tafseel, parameter tafseelat, aur doosra metadata shamil hota hai.

Parameters:

- **`func`** (`Callable[..., Any]`):
  - Woh function jis se schema extract karna hai. (Zaroori)
- **`docstring_style`** (`DocstringStyle | None`):
  - Parsing ke liye istemal hone wale docstring ka style. Agar faraham na kiya jaye, toh hum style ko auto-detect karne ki koshish karengi.
- **`name_override`** (`str | None`):
  - Agar faraham kiya jaye, toh function ke `__name__` ki bajaye is naam ko istemal karein.

- **`description_override`** (`str | None`):
    - Agar faraham kiya jaye, toh docstring se hasil shuda tafseel ki bajaye is tafseel ko istemal karein.
- **`use_docstring_info`** (`bool`, Default: `True`):
    - Agar `True` ho, toh tafseel aur parameter tafseelat generate karne ke liye docstring ko istemal karein.
- **`strict_json_schema`** (`bool`, Default: `True`):
    - Kya JSON schema strict mode mein ho. Agar `True` ho, toh hum yakeeni banayengi ke schema "strict" standard ke mutabiq ho jaisa ke OpenAI API expect karta hai. Hum isay `True` par set karne ki **sakhti se sifarish** karte hain, kyunki yeh LLM ko sahi JSON input faraham karne ke imkaan ko barhata hai.

Returns:

- **`FuncSchema`**: Aik `FuncSchema` object jis mein function ka naam, tafseel, parameter tafseelat, aur doosra metadata shamil hota hai.

`function_schema` module aapko LLMs ko azeem tools banne mein madad karta hai, kyunki yeh LLM ko aapke code se interact karne ke liye zaroori structure aur maloomat faraham karta hai.

Model interface

Model Interface (Model Interface)

`Model Interface` module Agent SDK mein Large Language Models (LLMs) ke saath baat cheet (interaction) ke liye buniyadi tareeqay (interfaces) ko bayan karta hai. Yeh is baat ko yakeeni banata hai ke mukhtalif LLM providers aur models ko aik consistent tareeqay se istemal kiya ja sake.

`ModelTracing` (Enum)

`ModelTracing` aik Enum hai jo LLM calls ke liye tracing ki configuration ko define karta hai. Tracing aapko agent aur model ke darmiyan hone wali bat cheet ko monitor aur debug karne mein madad karta hai.

- **`DISABLED = 0`**: Tracing mukammal tor par **disabled** hai. Koi data record nahi kiya jayega.
- **`ENABLED = 1`**: Tracing **enabled** hai, aur tamam data (inputs/outputs shamil hain) record kiya jayega.
- **`ENABLED_WITHOUT_DATA = 2`**: Tracing **enabled** hai, lekin inputs/outputs ko shamil nahi kiya jayega. Sirf call ki maloomat (metadata) record ki jayegi.

`Model` (Abstract Base Class)

`Model` LLM ko call karne ke liye **buniyadi interface** hai. Koi bhi khaas LLM (masalan, OpenAI ka GPT model ya Google ka Gemini model) is interface ko implement karega.

Abstract Methods (Jinhein concrete model classes mein implement karna lazmi hai):

- **`get_response`** (async method):
  - Model se **mukammal jawab** hasil karta hai. Yeh aik non-streaming call hai.
  - **Parameters:**
    - **`system_instructions`** (`str | None`): Istemal ki jane wali system instructions.
    - **`input`** (`str | list[TResponseInputItem]`): Model ko diye jane wale input items, OpenAI Responses format mein.
    - **`model_settings`** (`ModelSettings`): Istemal ki jane wali model settings (temperature, max_tokens, waghera).
    - **`tools`** (`list[Tool]`): Model ke liye dastiyab tools.
    - **`output_schema`** (`AgentOutputSchemaBase | None`): Istemal ki jane wali output schema (masalan, JSON structure).
    - **`handoffs`** (`list[Handoff]`): Model ke liye dastiyab handoffs.
    - **`tracing`** (`ModelTracing`): Tracing configuration.
    - **`previous_response_id`** (`str | None`): Pichle response ki ID.
    - **`prompt`** (`ResponsePromptParam | None`): Model ke liye istemal ki jane wali prompt config.
  - **Returns:**
    - **`ModelResponse`**: Mukammal model response.
- **`stream_response`** (abstract method):
  - Model se jawab ko **stream** karta hai. Yeh streaming calls ke liye hai, jahan jawab hisson mein aata hai.
  - **Parameters:** `get_response` jaisay hi parameters.
  - **Returns:**
    - **`AsyncIterator[TResponseStreamEvent]`**: Response stream events ka aik iterator, OpenAI Responses format mein.

`ModelProvider` (Abstract Base Class)

`ModelProvider` aik **model provider ke liye buniyadi interface** hai. Model provider ka kaam model ko uske naam se dhoondhna hai. Misal ke tor par, aik OpenAI provider `gpt-4` ya `gpt-3.5-turbo` jaisay models faraham karega.

Abstract Methods:

- **`get_model(model_name: str | None) -> Model`**:
  - Naam ke zariye aik **model hasil karta hai**.
  - **Parameters:**
    - **`model_name`** (`str | None`): Hasil kiye jane wale model ka naam.
  - **Returns:**
    - **`Model`**: Talab kiya gaya model.

Yeh interfaces Agent SDK ko mukhtalif LLM providers ke saath lachakdar tareeqay se kaam karne ki ijazat dete hain, jabke developers ke liye aik consistent API faraham karte hain.

OpenAI Chat Completions model

OpenAI Chat Completions Model (OpenAI Chat Completions Model)

`OpenAIChatCompletionsModel` Agent SDK mein aik khaas model class hai jo **OpenAI ke Chat Completions API ke saath interact karti hai**. Yeh `Model` base interface ko implement karti hai, iska matlab hai ke yeh LLM se jawab hasil karne aur stream karne ke buniyadi tareeqon ko faraham karti hai.

`stream_response` (Async Method)

Yeh method **model se jawab ko stream karta hai**, yani jawab ke parts aur usage ki maloomat jaisay woh generate hoti hain, yield karta hai.

Parameters:

- **`system_instructions`** (`str | None`): Woh system instructions jo chat completion request mein istemal ki jayengi.
- **`input`** (`str | list[TResponseInputItem]`): Model ko diya jane wala input, jo OpenAI Responses format mein hota hai. Is mein user messages, assistant replies, tool calls, waghera shamil ho sakte hain.
- **`model_settings`** (`ModelSettings`): Model ki settings, jismein temperature, max_tokens, aur dusre parameters shamil hain jo OpenAI Chat Completions API ko pass kiye jayenge.
- **`tools`** (`list[Tool]`): Woh tools jo model ke liye dastiyab hain. Yeh tools functions ke tor par LLM ko diye jate hain taake woh zaroorat ke mutabiq unhein call kar sake.
- **`output_schema`** (`AgentOutputSchemaBase | None`): Agar model se structured output (masalan, JSON object) mutawaqqi ho toh istemal ki jane wali output schema.
- **`handoffs`** (`list[Handoff]`): Woh handoffs jo model ke liye dastiyab hain. Handoffs bhi tools ki tarah represent kiye ja sakte hain.
- **`tracing`** (`ModelTracing`): Tracing ki configuration, jo yeh tay karti hai ke model ke calls ko kaise monitor kiya jaye.
- **`previous_response_id`** (`str | None`): Pichle response ki ID, agar koi ho. Yeh OpenAI Responses API mein context maintain karne ke liye istemal ho sakti hai.
- **`prompt`** (`ResponsePromptParam | None = None`): Model ke liye istemal ki jane wali prompt configuration.

Returns:

- **`AsyncIterator[TResponseStreamEvent]`**: `TResponseStreamEvent` types ka aik async iterator. Yeh stream karte waqt mukhtalif events (jaise message deltas, tool call deltas) faraham karta hai.

Functionality (Karkardagi):

`OpenAIChatCompletionsModel` ka buniyadi kaam yeh hai ke yeh Agent SDK ke generalized `Model` interface aur OpenAI Chat Completions API ke darmiyan aik bridge ke tor par kaam karta hai. Jab aapka agent OpenAI model istemal karta hai, toh yeh class OpenAI API calls ko handle karti hai, input ko sahi format mein tabdeel karti hai, settings apply karti hai, aur phir OpenAI se hasil hone wale raw jawab ko Agent SDK ke `StreamEvent` format mein convert karti hai.

OpenAI Responses model

OpenAI Responses Model (OpenAI Responses Model)

`OpenAIResponsesModel` Agent SDK mein aik khaas model class hai jo **OpenAI Responses API ko istemal karti hai**. Yeh bhi `Model` base interface ko implement karti hai, bilkul `OpenAIChatCompletionsModel` ki tarah, lekin iska focus OpenAI ki Responses API par hai.

`stream_response` (Async Method)

Yeh method **model se jawab ko stream karta hai**, bilkul `OpenAIChatCompletionsModel` ke `stream_response` method ki tarah. Yani, yeh jawab ke hisson aur usage ki maloomat ko generate hone ke saath-saath provide karta hai.

Parameters:

- **`system_instructions`** (`str | None`): Woh system instructions jo request mein istemal ki jayengi.
- **`input`** (`str | list[TResponseInputItem]`): Model ko diya jane wala input, jo OpenAI Responses format mein hota hai. Is mein user messages, assistant replies, tool calls, waghera shamil ho sakte hain.
- **`model_settings`** (`ModelSettings`): Model ki settings, jismein temperature, max_tokens, aur dusre parameters shamil hain jo OpenAI Responses API ko pass kiye jayenge.
- **`tools`** (`list[Tool]`): Woh tools jo model ke liye dastiyab hain.
- **`output_schema`** (`AgentOutputSchemaBase | None`): Agar model se structured output (masalan, JSON object) mutawaqqi ho toh istemal ki jane wali output schema.
- **`handoffs`** (`list[Handoff]`): Woh handoffs jo model ke liye dastiyab hain.
- **`tracing`** (`ModelTracing`): Tracing ki configuration, jo yeh tay karti hai ke model ke calls ko kaise monitor kiya jaye.
- **`previous_response_id`** (`str | None`): Pichle response ki ID, agar koi ho. Yeh Responses API mein context maintain karne ke liye khaas tor par ahem hai.
- **`prompt`** (`ResponsePromptParam | None = None`): Model ke liye istemal ki jane wali prompt configuration.

Returns:

- **`AsyncIterator[ResponseStreamEvent]`**: `ResponseStreamEvent` types ka aik async iterator. Yeh stream karte waqt mukhtalif events (jaise message deltas, tool call deltas) faraham karta hai.

`Converter` (Class/Module)

`Converter` ek utility class ya module hai jo `OpenAIResponsesModel` ke andar ya uske saath istemal hota hai. Iska buniyadi maqsad **data ko Agent SDK ke internal formats aur OpenAI Responses API ke formats ke darmiyan convert karna** hai. Yeh conversions is baat ko yakeeni banate hain ke Agent SDK aur OpenAI API ke darmiyan data ka tabadlah (exchange) sahi tareeqay se ho sake.

Functionality (Karkardagi):

OpenAIResponsesModel ka kaam OpenAI ki **Responses API ki capabilities ko Agent SDK mein layana** hai. Jabke ChatCompletions API zyada generalized chat interactions ke liye hai, Responses API kuch advance features aur structure provide karti hai. Is model ko istemal karne ka matlab hai ke aap OpenAI ke Responses API ke khaas features (jaise ke response_id se context management) se faida utha sakte hain jab aapka agent OpenAI models ke saath kaam kar raha ho.

MCP Servers

MCP Servers (Model Context Protocol Servers)

MCP Servers module Model Context Protocol (MCP) servers ko define karta hai. MCP aik protocol hai jo agents aur tools ke darmiyan communication ko standardize karta hai. Yeh servers tools provide karte hain jinhein agents istemal kar sakte hain, aur yeh mukhtalif transport mechanisms (jaise Stdio, SSE, Streamable HTTP) ke zariye operate kar sakte hain.

MCPServer (Abstract Base Class)

MCPServer Model Context Protocol servers ke liye **buniyadi abstract class** hai. Koi bhi khaas MCP server is class ko implement karega.

Abstract Properties/Methods:

- **name** (str property):
  - Server ke liye aik **qabil-e-qirat (readable) naam**.
- **connect()** (async method):
  - Server se **connect** hota hai. Masalan, iska matlab subprocess shuru karna ya network connection kholna ho sakta hai. Server se cleanup() call hone tak connected rehne ki tawaqqo ki jati hai.
- **cleanup()** (async method):
  - Server ko **cleanup** karta hai. Masalan, iska matlab subprocess ko band karna ya network connection ko band karna ho sakta hai.
- **list_tools() -> list[Tool]** (async method):
  - Server par dastiyab **tools ki list** return karta hai.
- **call_tool(tool_name: str, arguments: dict[str, Any] | None) -> CallToolResult** (async method):
  - Server par aik **tool invoke** karta hai.

MCPServerStdioParams (TypedDict)

Yeh TypedDict mcp.client.stdio.StdioServerParameters ko mirror karta hai aur aapko **stdio transport par chalne wale server ke parameters** pass karne ki ijazat deta hai.

Parameters:

- **`command`** (`str`):
    - Server shuru karne ke liye chalne wala **executable**. Masalan, `'python'` ya `'node'`.
- **`args`** (`NotRequired[list[str]]`):
    - `command` executable ko pass karne wale **command line arguments**. Masalan, `['foo.py']` ya `['server.js', '--port', '8080']`.
- **`env`** (`NotRequired[dict[str, str]]`):
    - Server ke liye set kiye jane wale **environment variables**.
- **`cwd`** (`NotRequired[str | Path]`):
    - Process ko spawn karte waqt istemal ki jane wali **working directory**.
- **`encoding`** (`NotRequired[str]`, Default: `'utf-8'`):
    - Server ko messages bhejte/receive karte waqt istemal hone wali **text encoding**. Default `utf-8`.
- **`encoding_error_handler`** (`NotRequired[Literal["strict", "ignore", "replace"]]`, Default: `'strict'`):
    - Text encoding **error handler**. Default `strict`. Mazeed tafseelat ke liye Python `codecs` documentation dekhein.

`MCPServerStdio` (Class)

`MCPServerStdio` MCP server ki implementation hai jo **stdio (Standard Input/Output) transport** ko istemal karti hai. Yeh aam tor par un servers ke liye istemal hota hai jo subprocess ke tor par chalte hain aur standard input/output streams ke zariye baat cheet karte hain.

Constructor (`__init__`) Parameters:

- **`params`** (`MCPServerStdioParams`): Server ko configure karne wale parameters. (Zaroori)
- **`cache_tools_list`** (`bool`, Default: `False`):
    - Kya tools list ko **cache karna hai**. Agar `True` ho, toh tools list sirf aik bar server se fetch ki jayegi. Agar `False` ho, toh `list_tools()` ke har call par tools list server se fetch ki jayegi. Cache ko `invalidate_tools_cache()` call karke invalidate kiya ja sakta hai. Yeh `True` set karna chahiye agar aap jante hain ke server apni tools list ko tabdeel nahi karega, kyunki yeh latency ko kafi behtar bana sakta hai.
- **`name`** (`str | None`):
    - Server ke liye aik readable naam. Agar faraham na kiya jaye, toh command se aik naam banaya jayega.
- **`client_session_timeout_seconds`** (`float | None`, Default: `5`):
    - MCP `ClientSession` ko pass kiya jane wala read timeout.

Methods:

- **`create_streams()`**: Server ke liye streams banata hai.
- **`connect()`** (async): Server se connect hota hai.
- **`cleanup()`** (async): Server ko cleanup karta hai.
- **`list_tools()`** (async): Server par dastiyab tools list karta hai.
- **`call_tool()`** (async): Server par aik tool invoke karta hai.
- **`invalidate_tools_cache()`**: Tools cache ko invalidate karta hai.

`MCPServerSseParams` (TypedDict)

Yeh `TypedDict` **HTTP with SSE (Server-Sent Events) transport** istemal karne wale server ke parameters ko mirror karta hai.

Parameters:

- **`url`** (`str`):
  - Server ka **URL**.
- **`headers`** (`NotRequired[dict[str, str]]`):
  - Server ko bheje jane wale **headers**.
- **`timeout`** (`NotRequired[float]`, Default: `5`):
  - HTTP request ke liye **timeout**. Default 5 seconds.
- **`sse_read_timeout`** (`NotRequired[float]`, Default: `300` seconds / 5 minutes):
  - SSE connection ke liye **timeout**. Default 5 minutes.

`MCPServerSse` (Class)

`MCPServerSse` MCP server ki implementation hai jo **HTTP with SSE transport** ko istemal karti hai. SSE aik light-weight protocol hai jo server ko client ko events push karne ki ijazat deta hai.

Constructor Parameters and Methods:

- Iske parameters aur methods `MCPServerStdio` se mushabah hain, lekin `params` `MCPServerSseParams` type ka hota hai aur `name` URL se banaya jata hai agar faraham na kiya jaye.

`MCPServerStreamableHttpParams` (TypedDict)

Yeh `TypedDict` **Streamable HTTP transport** istemal karne wale server ke parameters ko mirror karta hai.

Parameters:

- **`url`** (`str`):
  - Server ka **URL**.
- **`headers`** (`NotRequired[dict[str, str]]`):
  - Server ko bheje jane wale **headers**.
- **`timeout`** (`NotRequired[timedelta | float]`, Default: `5`):
  - HTTP request ke liye **timeout**. Default 5 seconds.
- **`sse_read_timeout`** (`NotRequired[timedelta | float]`, Default: `300` seconds / 5 minutes):
  - SSE connection ke liye **timeout**. Default 5 minutes.
- **`terminate_on_close`** (`NotRequired[bool]`):
  - Kya close hone par **terminate karna hai**.

`MCPServerStreamableHttp` (Class)

`MCPServerStreamableHttp` MCP server ki implementation hai jo **Streamable HTTP transport** ko istemal karti hai. Yeh naye transport spec ke mutabiq design kiya gaya hai jo streamable data ke liye optimized hai.

Constructor Parameters and Methods:

- Iske parameters aur methods `MCPServerSse` se mushabah hain, lekin `params` `MCPServerStreamableHttpParams` type ka hota hai.

MCP Servers aapko apne agents ke liye custom tools provide karne aur unhein mukhtalif communication channels ke zariye expose karne ki flexibility dete hain.

MCP Util

MCP Utilities (MCP Utilities)

`MCPUtil` class **Model Context Protocol (MCP) aur Agents SDK tools ke darmiyan behtar taalmel (interoperability) ke liye utilities ka aik set** faraham karta hai. Yeh aapko MCP servers se tools ko Agents SDK mein integrate karne aur unhein istemal karne mein madad karta hai.

`get_all_function_tools` (Async Classmethod)

Yeh method **MCP servers ki list se tamam function tools hasil karta hai**.

- **Parameters:**
  - **`servers`** (`list[MCPServer]`): MCP servers ki list jin se tools hasil karne hain.
  - **`convert_schemas_to_strict`** (`bool`): Kya tools ke schemas ko strict mode mein convert karna hai. Isay `True` set karne se LLM ko sahi JSON input faraham karne ka imkaan barh jata hai.
- **Returns:**
  - **`list[Tool]`**: Hasil shuda **function tools ki list**.

`get_function_tools` (Async Classmethod)

Yeh method **aik single MCP server se tamam function tools hasil karta hai**.

- **Parameters:**
  - **`server`** (`MCPServer`): Woh MCP server jis se tools hasil karne hain.
  - **`convert_schemas_to_strict`** (`bool`): Kya tools ke schemas ko strict mode mein convert karna hai.
- **Returns:**
  - **`list[Tool]`**: Hasil shuda **function tools ki list**.

`to_function_tool` (Classmethod)

Yeh method **aik MCP tool ko Agents SDK function tool mein tabdeel karta hai**. Yeh conversion zaroori hai taake Agents SDK MCP tools ko samajh sake aur unhein istemal kar sake.

- **Parameters:**
  - **`tool`** (`Tool`): Woh MCP tool jise convert karna hai.
  - **`server`** (`MCPServer`): Woh MCP server jis se tool aaya hai.
  - **`convert_schemas_to_strict`** (`bool`): Kya tool ke schema ko strict mode mein convert karna hai.
- **Returns:**
  - **`FunctionTool`**: Agents SDK ka **FunctionTool** object.

`invoke_mcp_tool` (Async Classmethod)

Yeh method **aik MCP tool ko invoke karta hai aur nateeja string ke tor par return karta hai**. Yeh method agent ko MCP server par maujood kisi bhi tool ko call karne ki ijazat deta hai.

- **Parameters:**
  - **`server`** (`MCPServer`): Woh MCP server jis par tool maujood hai.
  - **`tool`** (`Tool`): Woh tool jise invoke karna hai.
  - **`context`** (`RunContextWrapper[Any]`): Run context, jis mein agent run ki maloomat shamil hoti hai.
  - **`input_json`** (`str`): Tool ko pass kiya jane wala input JSON string ke tor par.
- **Returns:**
  - **`str`**: Tool call ka **nateeja string ke tor par**.

`MCPUtil` class MCP servers par maujood custom tools ko Agents SDK ke andar seamlessly istemal karne ke liye aik ahem bridge faraham karti hai. Isse aap apne agents ki functionality ko MCP-based services aur tools ke zariye expand kar sakte hain.

TRACING

TraceProvider

Bases: ABC

Interface for creating traces and spans.

Source code in src/agents/tracing/provider.py

register_processor *abstractmethod*

register_processor(processor: TracingProcessor) -> None

Add a processor that will receive all traces and spans.

Source code in src/agents/tracing/provider.py

set_processors *abstractmethod*

set_processors(processors: list[TracingProcessor]) -> None

Replace the list of processors with processors.

Source code in src/agents/tracing/provider.py

get_current_trace *abstractmethod*

get_current_trace() -> Trace | None

Return the currently active trace, if any.

Source code in src/agents/tracing/provider.py

get_current_span abstractmethod

get_current_span() -> Span[Any] | None

Return the currently active span, if any.

Source code in src/agents/tracing/provider.py

set_disabled abstractmethod

set_disabled(disabled: bool) -> None

Enable or disable tracing globally.

Source code in src/agents/tracing/provider.py

time_iso abstractmethod

time_iso() -> str

Return the current time in ISO 8601 format.

Source code in src/agents/tracing/provider.py

gen_trace_id abstractmethod

gen_trace_id() -> str

Generate a new trace identifier.

Source code in src/agents/tracing/provider.py

gen_span_id abstractmethod

gen_span_id() -> str

Generate a new span identifier.

Source code in src/agents/tracing/provider.py

gen_group_id *abstractmethod*

```
gen_group_id() -> str
```

Generate a new group identifier.

Source code in src/agents/tracing/provider.py

create_trace *abstractmethod*

```
create_trace(
    name: str,
    trace_id: str | None = None,
    group_id: str | None = None,
    metadata: dict[str, Any] | None = None,
    disabled: bool = False,
) -> Trace
```

Create a new trace.

Source code in src/agents/tracing/provider.py

create_span *abstractmethod*

```
create_span(
    span_data: TSpanData,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[TSpanData]
```

Create a new span.

Source code in src/agents/tracing/provider.py

shutdown *abstractmethod*

```
shutdown() -> None
```

Clean up any resources used by the provider.

Source code in src/agents/tracing/provider.py

TracingProcessor

Bases: ABC

Interface for processing spans.

Source code in src/agents/tracing/processor_interface.py

on_trace_start *abstractmethod*

on_trace_start(trace: Trace) -> None

Called when a trace is started.

Parameters:

Name	Type	Description	Default
trace	Trace	The trace that started.	*required*

Source code in src/agents/tracing/processor_interface.py

on_trace_end *abstractmethod*

on_trace_end(trace: Trace) -> None

Called when a trace is finished.

Parameters:

Name	Type	Description	Default
trace	Trace	The trace that started.	*required*

Source code in src/agents/tracing/processor_interface.py

on_span_start *abstractmethod*

on_span_start(span: Span[Any]) -> None

Called when a span is started.

Parameters:

Name	Type	Description	Default
span	Span[Any]	The span that started.	*required*

Source code in src/agents/tracing/processor_interface.py

on_span_end *abstractmethod*

on_span_end(span: Span[Any]) -> None

Called when a span is finished. Should not block or raise exceptions.

Parameters:

Name	Type	Description	Default
span	Span[Any]	The span that finished.	*required*

Source code in src/agents/tracing/processor_interface.py

shutdown *abstractmethod*

shutdown() -> None

Called when the application stops.

Source code in src/agents/tracing/processor_interface.py

force_flush *abstractmethod*

force_flush() -> None

Forces an immediate flush of all queued spans/traces.

Source code in src/agents/tracing/processor_interface.py

AgentSpanData

Bases: SpanData

Represents an Agent Span in the trace. Includes name, handoffs, tools, and output type.

Source code in src/agents/tracing/span_data.py

CustomSpanData

Bases: SpanData

Represents a Custom Span in the trace. Includes name and data property bag.

Source code in src/agents/tracing/span_data.py

FunctionSpanData

Bases: SpanData

Represents a Function Span in the trace. Includes input, output and MCP data (if applicable).

Source code in src/agents/tracing/span_data.py

GenerationSpanData

Bases: SpanData

Represents a Generation Span in the trace. Includes input, output, model, model configuration, and usage.

Source code in src/agents/tracing/span_data.py

GuardrailSpanData

Bases: SpanData

Represents a Guardrail Span in the trace. Includes name and triggered status.

Source code in src/agents/tracing/span_data.py

HandoffSpanData

Bases: SpanData

Represents a Handoff Span in the trace. Includes source and destination agents.

Source code in src/agents/tracing/span_data.py

MCPListToolsSpanData

Bases: SpanData

Represents an MCP List Tools Span in the trace. Includes server and result.

Source code in src/agents/tracing/span_data.py

ResponseSpanData

Bases: SpanData

Represents a Response Span in the trace. Includes response and input.

Source code in src/agents/tracing/span_data.py

SpanData

Bases: ABC

Represents span data in the trace.

Source code in src/agents/tracing/span_data.py

type *abstractmethod property*

type: str

Return the type of the span.

export *abstractmethod*

export() -> dict[str, Any]

Export the span data as a dictionary.

Source code in src/agents/tracing/span_data.py

SpeechGroupSpanData

Bases: SpanData

Represents a Speech Group Span in the trace.

Source code in src/agents/tracing/span_data.py

SpeechSpanData

Bases: SpanData

Represents a Speech Span in the trace. Includes input, output, model, model configuration, and first content timestamp.

Source code in src/agents/tracing/span_data.py

TranscriptionSpanData

Bases: SpanData

Represents a Transcription Span in the trace. Includes input, output, model, and model configuration.

Source code in src/agents/tracing/span_data.py

Span

Bases: ABC, Generic[TSpanData]

Source code in src/agents/tracing/spans.py

start *abstractmethod*

start(mark_as_current: bool = False)

Start the span.

Parameters:

Name	Type	Description	Default
mark_as_current	bool	If true, the span will be marked as the current span.	False

Source code in src/agents/tracing/spans.py

finish *abstractmethod*

finish(reset_current: bool = False) -> None

Finish the span.

Parameters:

Name	Type	Description	Default
reset_current	bool	If true, the span will be reset as the current span.	False

Source code in src/agents/tracing/spans.py

Trace

A trace is the root level object that tracing creates. It represents a logical "workflow".

Source code in src/agents/tracing/traces.py

trace_id abstractmethod property

trace_id: str

The trace ID.

name abstractmethod property

name: str

The name of the workflow being traced.

start abstractmethod

start(mark_as_current: bool = False)

Start the trace.

Parameters:

Name	Type	Description	Default
mark_as_current	bool	If true, the trace will be marked as the current trace.	False

Source code in src/agents/tracing/traces.py

finish abstractmethod

finish(reset_current: bool = False)

Finish the trace.

Parameters:

Name	Type	Description	Default
reset_current	bool	If true, the trace will be reset as the current trace.	False

Source code in src/agents/tracing/traces.py *export abstractmethod*

```
export() -> dict[str, Any] | None
```

Export the trace as a dictionary.

Source code in src/agents/tracing/traces.py

agent_span

```
agent_span(
    name: str,
    handoffs: list[str] | None = None,
    tools: list[str] | None = None,
    output_type: str | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[AgentSpanData]
```

Create a new agent span. The span will not be started automatically, you should either do with agent_span() ... or call span.start() + span.finish() manually.

Parameters:

Name	Type	Description	Default
name	str	The name of the agent.	*required*
handoffs	list[str] | None	Optional list of agent names to which this agent could hand off control.	None
tools	list[str] | None	Optional list of tool names available to this agent.	None
output_type	str | None	Optional name of the output type produced by the agent.	None
span_id	str | None	The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen_span_id() to generate a span ID,	None

Name	Type	Description	Default
		to guarantee that IDs are correctly formatted.	
parent	Trace | Span[Any] | None	The parent span or trace. If not provided, we will automatically use the current trace/span as the parent.	None
disabled	bool	If True, we will return a Span but the Span will not be recorded.	False

Returns:

Type	Description
Span[AgentSpanData]	The newly created agent span.

Source code in src/agents/tracing/create.py

custom_span

```
custom_span(
    name: str,
    data: dict[str, Any] | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[CustomSpanData]
```

Create a new custom span, to which you can add your own metadata. The span will not be started automatically, you should either do with custom_span() ... or call span.start() + span.finish() manually.

Parameters:

Name	Type	Description	Default
name	str	The name of the custom span.	*required*

Name	Type	Description	Default
data	dict[str, Any] | None	Arbitrary structured data to associate with the span.	None
span_id	str | None	The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen_span_id() to generate a span ID, to guarantee that IDs are correctly formatted.	None
parent	Trace | Span[Any] | None	The parent span or trace. If not provided, we will automatically use the current trace/span as the parent.	None
disabled	bool	If True, we will return a Span but the Span will not be recorded.	False

Returns:

Type	Description
Span[CustomSpanData]	The newly created custom span.

Source code in src/agents/tracing/create.py

function_span

```
function_span(
    name: str,
    input: str | None = None,
    output: str | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[FunctionSpanData]
```

Create a new function span. The span will not be started automatically, you should either do with function_span() ... or call span.start() + span.finish() manually.

Parameters:

Name	Type	Description	Default
name	str	The name of the function.	*required*
input	str | None	The input to the function.	None
output	str | None	The output of the function.	None
span_id	str | None	The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen_span_id() to generate a span ID, to guarantee that IDs are correctly formatted.	None
parent	Trace | Span[Any] | None	The parent span or trace. If not provided, we will automatically use the current trace/span as the parent.	None
disabled	bool	If True, we will return a Span but the Span will not be recorded.	False

Returns:

Type	Description
Span[FunctionSpanData]	The newly created function span.

Source code in src/agents/tracing/create.py

generation_span

```
generation_span(
    input: Sequence[Mapping[str, Any]] | None = None,
    output: Sequence[Mapping[str, Any]] | None = None,
    model: str | None = None,
    model_config: Mapping[str, Any] | None = None,
    usage: dict[str, Any] | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[GenerationSpanData]
```

Create a new generation span. The span will not be started automatically, you should either do `with generation_span() ...` or call `span.start()` + `span.finish()` manually.

This span captures the details of a model generation, including the input message sequence, any generated outputs, the model name and configuration, and usage data. If you only need to capture a model response identifier, use `response_span()` instead.

Parameters:

Name	**Type**	**Description**	**Default**
input	Sequence[Mapping[str, Any]] | None	The sequence of input messages sent to the model.	None
output	Sequence[Mapping[str, Any]] | None	The sequence of output messages received from the model.	None
model	str | None	The model identifier used for the generation.	None
model_config	Mapping[str, Any] | None	The model configuration (hyperparameters) used.	None
usage	dict[str, Any] | None	A dictionary of usage information (input tokens, output tokens, etc.).	None
span_id	str | None	The ID of the span. Optional. If not provided, we will generate an ID. We recommend using `util.gen_span_id()` to generate a span ID, to guarantee that IDs are correctly formatted.	None

Name	Type	Description	Default
parent	Trace | Span[Any] | None	The parent span or trace. If not provided, we will automatically use the current trace/span as the parent.	None
disabled	bool	If True, we will return a Span but the Span will not be recorded.	False

Returns:

Type	Description
Span[GenerationSpanData]	The newly created generation span.

Source code in src/agents/tracing/create.py

get_current_span

get_current_span() -> Span[Any] | None

Returns the currently active span, if present.

Source code in src/agents/tracing/create.py

get_current_trace

get_current_trace() -> Trace | None

Returns the currently active trace, if present.

Source code in src/agents/tracing/create.py

guardrail_span

```
guardrail_span(
    name: str,
    triggered: bool = False,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[GuardrailSpanData]
```

Create a new guardrail span. The span will not be started automatically, you should either do `with guardrail_span() ...` or call `span.start()` + `span.finish()` manually.

Parameters:

Name	Type	Description	Default
name	str	The name of the guardrail.	*required*
triggered	bool	Whether the guardrail was triggered.	False
span_id	str | None	The ID of the span. Optional. If not provided, we will generate an ID. We recommend using `util.gen_span_id()` to generate a span ID, to guarantee that IDs are correctly formatted.	None
parent	Trace | Span[Any] | None	The parent span or trace. If not provided, we will automatically use the current trace/span as the parent.	None
disabled	bool	If True, we will return a Span but the Span will not be recorded.	False

Source code in `src/agents/tracing/create.py`

handoff_span

```
handoff_span(
    from_agent: str | None = None,
    to_agent: str | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[HandoffSpanData]
```

Create a new handoff span. The span will not be started automatically, you should either do with handoff_span() ... or call span.start() + span.finish() manually.

Parameters:

Name	Type	Description	Default
from_agent	str | None	The name of the agent that is handing off.	None
to_agent	str | None	The name of the agent that is receiving the handoff.	None
span_id	str | None	The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen_span_id() to generate a span ID, to guarantee that IDs are correctly formatted.	None
parent	Trace | Span[Any] | None	The parent span or trace. If not provided, we will automatically use the current trace/span as the parent.	None
disabled	bool	If True, we will return a Span but the Span will not be recorded.	False

Returns:

Type	Description
Span[HandoffSpanData]	The newly created handoff span.

Source code in src/agents/tracing/create.py

mcp_tools_span

```
mcp_tools_span(
    server: str | None = None,
    result: list[str] | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[MCPListToolsSpanData]
```

Create a new MCP list tools span. The span will not be started automatically, you should either do with mcp_tools_span() ... or call span.start() + span.finish() manually.

Parameters:

Name	Type	Description	Default
server	str | None	The name of the MCP server.	None
result	list[str] | None	The result of the MCP list tools call.	None
span_id	str | None	The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen_span_id() to generate a span ID, to guarantee that IDs are correctly formatted.	None
parent	Trace | Span[Any] | None	The parent span or trace. If not provided, we will automatically use the current trace/span as the parent.	None
disabled	bool	If True, we will return a Span but the Span will not be recorded.	False

Source code in src/agents/tracing/create.py

response_span

```
response_span(
    response: Response | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[ResponseSpanData]
```

Create a new response span. The span will not be started automatically, you should either do with response_span() ... or call span.start() + span.finish() manually.

Parameters:

Name	Type	Description	Default
response	Response | None	The OpenAI Response object.	None
span_id	str | None	The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen_span_id() to generate a span ID, to guarantee that IDs are correctly formatted.	None
parent	Trace | Span[Any] | None	The parent span or trace. If not provided, we will automatically use the current trace/span as the parent.	None
disabled	bool	If True, we will return a Span but the Span will not be recorded.	False

Source code in src/agents/tracing/create.py

speech_group_span

```
speech_group_span(
    input: str | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[SpeechGroupSpanData]
```

Create a new speech group span. The span will not be started automatically, you should either do with speech_group_span() ... or call span.start() + span.finish() manually.

Parameters:

Name	Type	Description	Default
input	str | None	The input text used for the speech request.	None

Name	Type	Description	Default
span_id	str | None	The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen_span_id() to generate a span ID, to guarantee that IDs are correctly formatted.	None
parent	Trace | Span[Any] | None	The parent span or trace. If not provided, we will automatically use the current trace/span as the parent.	None
disabled	bool	If True, we will return a Span but the Span will not be recorded.	False

Source code in src/agents/tracing/create.py

speech_span

```
speech_span(
    model: str | None = None,
    input: str | None = None,
    output: str | None = None,
    output_format: str | None = "pcm",
    model_config: Mapping[str, Any] | None = None,
    first_content_at: str | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[SpeechSpanData]
```

Create a new speech span. The span will not be started automatically, you should either do with speech_span() ... or call span.start() + span.finish() manually.

Parameters:

Name	Type	Description	Default
model	str | None	The name of the model used for the text-to-speech.	None
input	str | None	The text input of the text-to-speech.	None

Name	Type	Description	Default
output	str | None	The audio output of the text-to-speech as base64 encoded string of PCM audio bytes.	None
output_format	str | None	The format of the audio output (defaults to "pcm").	'pcm'
model_config	Mapping[str, Any] | None	The model configuration (hyperparameters) used.	None
first_content_at	str | None	The time of the first byte of the audio output.	None
span_id	str | None	The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen_span_id() to generate a span ID, to guarantee that IDs are correctly formatted.	None
parent	Trace | Span[Any] | None	The parent span or trace. If not provided, we will automatically use the current trace/span as the parent.	None
disabled	bool	If True, we will return a Span but the Span will not be recorded.	False

Source code in src/agents/tracing/create.py

trace

```
trace(
    workflow_name: str,
    trace_id: str | None = None,
    group_id: str | None = None,
    metadata: dict[str, Any] | None = None,
    disabled: bool = False,
) -> Trace
```

Create a new trace. The trace will not be started automatically; you should either use it as a context manager (with trace(...):) or call trace.start() + trace.finish() manually.

In addition to the workflow name and optional grouping identifier, you can provide an arbitrary metadata dictionary to attach additional user-defined information to the trace.

Parameters:

Name	Type	Description	Default
workflow_name	str	The name of the logical app or workflow. For example, you might provide "code_bot" for a coding agent, or "customer_support_agent" for a customer support agent.	*required*
trace_id	str | None	The ID of the trace. Optional. If not provided, we will generate an ID. We recommend using util.gen_trace_id() to generate a trace ID, to guarantee that IDs are correctly formatted.	None
group_id	str | None	Optional grouping identifier to link multiple traces from the same conversation or process. For instance, you might use a chat thread ID.	None
metadata	dict[str, Any] | None	Optional dictionary of additional metadata to attach to the trace.	None
disabled	bool	If True, we will return a Trace but the Trace will not be recorded. This will not be checked if there's an existing trace and even_if_trace_running is True.	False

Returns:

Type	Description
Trace	The newly created trace object.

Source code in src/agents/tracing/create.py

transcription_span

```
transcription_span(
    model: str | None = None,
    input: str | None = None,
    input_format: str | None = "pcm",
    output: str | None = None,
    model_config: Mapping[str, Any] | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[TranscriptionSpanData]
```

Create a new transcription span. The span will not be started automatically, you should either do with transcription_span() ... or call span.start() + span.finish() manually.

Parameters:

Name	Type	Description	Default
model	str | None	The name of the model used for the speech-to-text.	None
input	str | None	The audio input of the speech-to-text transcription, as a base64 encoded string of audio bytes.	None
input_format	str | None	The format of the audio input (defaults to "pcm").	'pcm'
output	str | None	The output of the speech-to-text transcription.	None
model_config	Mapping[str, Any] | None	The model configuration (hyperparameters) used.	None
span_id	str | None	The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen_span_id() to generate a span ID, to guarantee that IDs are correctly formatted.	None

Name	Type	Description	Default
parent	Trace | Span[Any] | None	The parent span or trace. If not provided, we will automatically use the current trace/span as the parent.	None
disabled	bool	If True, we will return a Span but the Span will not be recorded.	False

Returns:

Type	Description
Span[TranscriptionSpanData]	The newly created speech-to-text span.

Source code in src/agents/tracing/create.py

get_trace_provider

get_trace_provider() -> TraceProvider

Get the global trace provider used by tracing utilities.

Source code in src/agents/tracing/setup.py

set_trace_provider

set_trace_provider(provider: TraceProvider) -> None

Set the global trace provider used by tracing utilities.

Source code in src/agents/tracing/setup.py

gen_span_id

```
gen_span_id() -> str
```

Generate a new span ID.

Source code in src/agents/tracing/util.py

gen_trace_id

```
gen_trace_id() -> str
```

Generate a new trace ID.

Source code in src/agents/tracing/util.py

add_trace_processor

```
add_trace_processor(
    span_processor: TracingProcessor,
) -> None
```

Adds a new trace processor. This processor will receive all traces/spans.

Source code in src/agents/tracing/__init__.py

set_trace_processors

```
set_trace_processors(
    processors: list[TracingProcessor],
) -> None
```

Set the list of trace processors. This will replace the current list of processors.

Source code in src/agents/tracing/__init__.py

set_tracing_disabled

```
set_tracing_disabled(disabled: bool) -> None
```

Set whether tracing is globally disabled.

Source code in src/agents/tracing/__init__.py

set_tracing_export_api_key

set_tracing_export_api_key(api_key: str) -> None

Set the OpenAI API key for the backend exporter.

Source code in src/agents/tracing/__init__.py

Tracing Module (Tracing Module)

Tracing module **Agent SDK** mein **agent ke execution ki monitoring aur debugging** ke liye aik mazboot framework faraham karta hai. Yeh aapko agent ke internal flow, LLM calls, tool usage, aur doosre aham waqiyat (events) ko track karne ki ijazat deta hai. Tracing "traces" aur "spans" ke concept par mabni hai.

- **Trace (Trace):** Aik mukammal "workflow" ya logical operation ko represent karta hai, masalan aik agent run.
- **Span (Span):** Trace ke andar aik chhota, waqt ke lehaz se mukhtasir operation hota hai, masalan aik LLM call, aik tool invocation, ya aik guardrail check. Spans nested ho sakte hain, jo operation hierarchy ko darust karte hain.

`TraceProvider` (Abstract Base Class)

`TraceProvider` traces aur spans banane ke liye **buniyadi interface** hai. Koi bhi khaas tracing implementation is interface ko implement karegi.

Abstract Methods:

- `register_processor(processor: TracingProcessor) -> None`: Aik **processor shamil karta hai** jo tamam traces aur spans receive karega.
- `set_processors(processors: list[TracingProcessor]) -> None`: **Processors ki list ko replace** karta hai.
- `get_current_trace() -> Trace | None`: Maujooda **active trace** return karta hai, agar koi ho.
- `get_current_span() -> Span[Any] | None`: Maujooda **active span** return karta hai, agar koi ho.
- `set_disabled(disabled: bool) -> None`: Tracing ko **globally enable ya disable** karta hai.
- `time_iso() -> str`: Maujooda waqt ko **ISO 8601 format mein** return karta hai.
- `gen_trace_id() -> str`: Aik naya **trace identifier** generate karta hai.
- `gen_span_id() -> str`: Aik naya **span identifier** generate karta hai.
- `gen_group_id() -> str`: Aik naya **group identifier** generate karta hai.
- `create_trace(...) -> Trace`: Aik naya **trace banata hai**.
- `create_span(...) -> Span[TSpanData]`: Aik naya **span banata hai**.
- `shutdown() -> None`: Provider ke zariye istemal hone wale kisi bhi **resources ko clean up** karta hai.

`TracingProcessor` (Abstract Base Class)

`TracingProcessor` spans ko process karne ke liye **interface** hai. Jab aik trace ya span shuru ya khatam hota hai, toh registered processors ko notifications milti hain.

Abstract Methods:

- **`on_trace_start(trace: Trace) -> None`**: Jab aik **trace shuru hota hai** toh call kiya jata hai.
- **`on_trace_end(trace: Trace) -> None`**: Jab aik **trace mukammal hota hai** toh call kiya jata hai.
- **`on_span_start(span: Span[Any]) -> None`**: Jab aik **span shuru hota hai** toh call kiya jata hai.
- **`on_span_end(span: Span[Any]) -> None`**: Jab aik **span mukammal hota hai** toh call kiya jata hai. Isey block nahi karna chahiye ya exceptions raise nahi karni chahiye.
- **`shutdown() -> None`**: Jab **application rukti hai** toh call kiya jata hai.
- **`force_flush() -> None`**: Tamam **queued spans/traces ko fauri tor par flush** karne par majboor karta hai.

Span Data Classes

Yeh classes trace mein mukhtalif qism ke operations ke liye **span data** ko represent karti hain. Har class `SpanData` se inherit karti hai.

- **`AgentSpanData`**: Aik **Agent Span** ko represent karta hai, jismein naam, handoffs, tools, aur output type shamil hain.
- **`CustomSpanData`**: Aik **Custom Span** ko represent karta hai, jismein naam aur data property bag shamil hain.
- **`FunctionSpanData`**: Aik **Function Span** ko represent karta hai, jismein input, output aur MCP data (agar lagu ho) shamil hain.
- **`GenerationSpanData`**: Aik **Generation Span** ko represent karta hai, jismein input, output, model, model configuration, aur usage shamil hain.
- **`GuardrailSpanData`**: Aik **Guardrail Span** ko represent karta hai, jismein naam aur triggered status shamil hain.
- **`HandoffSpanData`**: Aik **Handoff Span** ko represent karta hai, jismein source aur destination agents shamil hain.
- **`MCPListToolsSpanData`**: Aik **MCP List Tools Span** ko represent karta hai, jismein server aur nateeja shamil hain.
- **`ResponseSpanData`**: Aik **Response Span** ko represent karta hai, jismein response aur input shamil hain.
- **`SpanData` (Abstract Base Class)**: Span data ke liye buniyadi class.
  - **`type`** (`str` property): Span ki type return karta hai.
  - **`export() -> dict[str, Any]`**: Span data ko dictionary ke tor par export karta hai.
- **`SpeechGroupSpanData`**: Aik **Speech Group Span** ko represent karta hai.
- **`SpeechSpanData`**: Aik **Speech Span** ko represent karta hai, jismein input, output, model, model configuration, aur first content timestamp shamil hain.
- **`TranscriptionSpanData`**: Aik **Transcription Span** ko represent karta hai, jismein input, output, model, aur model configuration shamil hain.

`Span` (Abstract Base Class)

Span aik **waqt ke lehaz se mukhtasir operation** ko represent karta hai jo trace ke andar hota hai.

Abstract Methods:

- **`start(mark_as_current: bool = False)`**: Span ko **shuru karta hai**.
- **`finish(reset_current: bool = False) -> None`**: Span ko **mukammal karta hai**.

`Trace` (Abstract Base Class)

Trace **root level object** hai jo tracing create karta hai. Yeh aik logical "workflow" ko represent karta hai.

Abstract Properties/Methods:

- **`trace_id`** (`str` property): **Trace ID**.
- **`name`** (`str` property): Trace kiye jane wale **workflow ka naam**.
- **`start(mark_as_current: bool = False)`**: Trace ko **shuru karta hai**.
- **`finish(reset_current: bool = False)`**: Trace ko **mukammal karta hai**.
- **`export() -> dict[str, Any] | None`**: Trace ko **dictionary ke tor par export** karta hai.

Span Creation Functions (`src/agents/tracing/create.py`)

Yeh functions mukhtalif qism ke spans banane ke liye conveniance methods hain. Har function aik `Span` object return karta hai, jise `with` statement ke saath ya manually `start()` aur `finish()` call karke istemal kiya ja sakta hai.

- **`agent_span(...) -> Span[AgentSpanData]`**: Aik naya **agent span** banata hai.
- **`custom_span(...) -> Span[CustomSpanData]`**: Aik naya **custom span** banata hai, jismein aap apni metadata shamil kar sakte hain.
- **`function_span(...) -> Span[FunctionSpanData]`**: Aik naya **function span** banata hai.
- **`generation_span(...) -> Span[GenerationSpanData]`**: Aik naya **generation span** banata hai jo model generation ki tafseelat capture karta hai.
- **`get_current_span() -> Span[Any] | None`**: Maujooda **active span** return karta hai.
- **`get_current_trace() -> Trace | None`**: Maujooda **active trace** return karta hai.
- **`guardrail_span(...) -> Span[GuardrailSpanData]`**: Aik naya **guardrail span** banata hai.
- **`handoff_span(...) -> Span[HandoffSpanData]`**: Aik naya **handoff span** banata hai.
- **`mcp_tools_span(...) -> Span[MCPListToolsSpanData]`**: Aik naya **MCP list tools span** banata hai.
- **`response_span(...) -> Span[ResponseSpanData]`**: Aik naya **response span** banata hai.
- **`speech_group_span(...) -> Span[SpeechGroupSpanData]`**: Aik naya **speech group span** banata hai.
- **`speech_span(...) -> Span[SpeechSpanData]`**: Aik naya **speech span** banata hai.
- **`trace(...) -> Trace`**: Aik naya **trace** banata hai.
- **`transcription_span(...) -> Span[TranscriptionSpanData]`**: Aik naya **transcription span** banata hai.

Setup and Utilities (`src/agents/tracing/setup.py` and `src/agents/tracing/util.py`)

- `get_trace_provider() -> TraceProvider`: Tracing utilities ke zariye istemal hone wala **global trace provider** hasil karta hai.
- `set_trace_provider(provider: TraceProvider) -> None`: Tracing utilities ke zariye istemal hone wala **global trace provider set** karta hai.
- `gen_span_id() -> str`: Aik naya **span ID** generate karta hai.
- `gen_trace_id() -> str`: Aik naya **trace ID** generate karta hai.
- `add_trace_processor(span_processor: TracingProcessor) -> None`: Aik naya **trace processor shamil** karta hai.
- `set_trace_processors(processors: list[TracingProcessor]) -> None`: **Trace processors ki list set** karta hai.
- `set_tracing_disabled(disabled: bool) -> None`: Tracing ko **globally disable ya enable** karta hai.
- `set_tracing_export_api_key(api_key: str) -> None`: Backend exporter ke liye **OpenAI API key set** karta hai.

Tracing module aapko apne agent-based applications ke **complex behaviour ko samajhne, performance issues ko debug karne, aur user interactions ko visualize** karne mein madad karta hai.

Creating Traces and Spans

Traces aur Spans Banana (Creating Traces and Spans)

Tracing modules mein, **traces aur spans** aapke agent ke complex operations ko monitor aur debug karne ke liye buniyadi blocks hain. Traces poore workflow ko darust karte hain, jabkay spans is workflow ke andar mukhtalif, chhote operations ko record karte hain.

Traces Banana (`trace`)

`trace` function aik naya trace object banata hai jo aapke logical application ya workflow ko represent karta hai.

Python

```python
trace(
    workflow_name: str,
    trace_id: str | None = None,
    group_id: str | None = None,
    metadata: dict[str, Any] | None = None,
    disabled: bool = False,
) -> Trace
```

- **`workflow_name` (zaroori):** Aapke application ya workflow ka naam. Masalan, "code_bot" ya "customer_support_agent".
- **`trace_id` (optional):** Trace ki ID. Agar aap provide nahi karte, toh system khud generate karega. `util.gen_trace_id()` istemal karna behtar hai.
- **`group_id` (optional):** Mutaddid traces ko aik hi conversation ya process se jorne ke liye istemal kiya jaane wala grouping identifier. Masalan, chat thread ID.

- **`metadata` (optional):** Trace ke saath attach karne ke liye additional user-defined information ki dictionary.
- **`disabled` (optional, default: `False`):** Agar `True` ho, toh trace record nahi hoga.

Istemal ke tareeqay:

1. **Context Manager ke tor par (Recommended):** Yeh sabse aasan tareeqa hai kyunki yeh trace ko khud ba khud start aur finish kar deta hai.

   Python

   ```
   from agents.tracing import trace

   async def my_agent_workflow():
       with trace(workflow_name="customer_onboarding", metadata={"user_id": "abc-123"}):
           # Is block ke andar ke tamam operations is trace ka hissa honge.
           print("Customer onboarding workflow shuru ho gaya.")
           # Yahan doosre spans banaye ja sakte hain
           await some_agent_step()
           print("Customer onboarding workflow mukammal ho gaya.")
   ```

2. **Manually Start aur Finish karna:**

   Python

   ```
   from agents.tracing import trace

   async def my_manual_workflow():
       my_trace = trace(workflow_name="data_processing")
       try:
           my_trace.start()
           print("Data processing shuru ho gaya.")
           # Doosre operations
           await process_data()
       finally:
           my_trace.finish()
           print("Data processing mukammal ho gaya.")
   ```

Maujooda Trace aur Span Hasil Karna

Yeh functions aapko execution context mein maujooda active trace ya span ko access karne ki ijazat dete hain.

- **`get_current_trace() -> Trace | None`**:
  - Agar koi **active trace** maujood ho toh usay return karta hai, warna `None`.
- **`get_current_span() -> Span[Any] | None`**:
  - Agar koi **active span** maujood ho toh usay return karta hai, warna `None`.

Spans Banana

Tracing module mukhtalif types ke operations ke liye khaas span creation functions faraham karta hai. Yeh sabhi functions `Span` object return karte hain aur unhein `with` statement ke saath ya manually `start()` aur `finish()` call karke istemal kiya ja sakta hai.

1. Agent Span (`agent_span`)

Agent ke actions ko track karta hai.

Python

```
agent_span(
    name: str,
    handoffs: list[str] | None = None,
    tools: list[str] | None = None,
    output_type: str | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[AgentSpanData]
```

- **`name` (zaroori):** Agent ka naam.
- **`handoffs` (optional):** Un agents ki list jinhein yeh agent control de sakta hai.
- **`tools` (optional):** Is agent ke liye dastiyab tool names ki list.
- **`output_type` (optional):** Agent ke zariye paida kiye gaye output type ka naam.

Misal:

Python

```
from agents.tracing import trace, agent_span

with trace(workflow_name="customer_service_bot"):
    with agent_span(name="initial_greeting_agent", handoffs=["faq_agent", "human_agent"]):
        print("Welcome to our service. How can I help you?")
```

2. Function Span (`function_span`)

Kissi bhi generic function call ko trace karta hai.

Python

```
function_span(
    name: str,
    input: str | None = None,
    output: str | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[FunctionSpanData]
```

- **`name` (zaroori):** Function ka naam.
- **`input` (optional):** Function ka input.
- **`output` (optional):** Function ka output.

Misal:

Python

```
from agents.tracing import trace, function_span

with trace(workflow_name="data_analyzer"):
    with function_span(name="calculate_average", input="[10, 20, 30]"):
        result = (10 + 20 + 30) / 3
    # function_span automatically calls finish() when exiting the 'with' block
    print(f"Average: {result}")
```

3. Generation Span (`generation_span`)

LLM model se generation details capture karta hai.

Python

```
generation_span(
    input: Sequence[Mapping[str, Any]] | None = None,
    output: Sequence[Mapping[str, Any]] | None = None,
    model: str | None = None,
    model_config: Mapping[str, Any] | None = None,
    usage: dict[str, Any] | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[GenerationSpanData]
```

- **`input` (optional):** Model ko bheje gaye input messages ki sequence.
- **`output` (optional):** Model se hasil kiye gaye output messages ki sequence.
- **`model` (optional):** Generation ke liye istemal hone wala model identifier.
- **`model_config` (optional):** Istemal ki gayi model configuration (hyperparameters).
- **`usage` (optional):** Usage information (masalan, input tokens, output tokens) ki dictionary.

Misal:

Python

```
from agents.tracing import trace, generation_span

with trace(workflow_name="content_generator"):
    with generation_span(
        input=[{"role": "user", "content": "Write a poem about the sea."}],
        model="gemini-1.5-pro",
        model_config={"temperature": 0.7},
    ) as span:
        # Simulate model call
        generated_poem = "The ocean deep, a mystery vast..."
        span.finish(output=[{"role": "assistant", "content": generated_poem}])
```

4. Response Span (`response_span`)

Aik mukammal OpenAI `Response` object ko trace karta hai.

Python

```
response_span(
    response: Response | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[ResponseSpanData]
```

- **`response` (optional):** OpenAI `Response` object.

Misal:

Python

```
# Assume 'openai_response_object' is an actual OpenAI Response object
from agents.tracing import trace, response_span

with trace(workflow_name="api_interaction"):
    with response_span(response=openai_response_object):
        print("OpenAI API response recorded.")
```

5. Handoff Span (`handoff_span`)

Agents ke darmiyan control ke handoff ko trace karta hai.

Python

```
handoff_span(
    from_agent: str | None = None,
    to_agent: str | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[HandoffSpanData]
```

- **`from_agent` (optional):** Handoff karne wale agent ka naam.
- **`to_agent` (optional):** Handoff receive karne wale agent ka naam.

Misal:

Python

```
from agents.tracing import trace, handoff_span

with trace(workflow_name="customer_support_flow"):
    with handoff_span(from_agent="chatbot", to_agent="human_agent"):
        print("Chatbot se human agent ko handoff.")
```

6. Custom Span (`custom_span`)

Apne khud ke metadata ke saath generalized operations ko trace karta hai.

Python

```
custom_span(
    name: str,
    data: dict[str, Any] | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[CustomSpanData]
```

- **`name` (zaroori):** Custom span ka naam.
- **`data` (optional):** Span ke saath associate karne ke liye arbitrary structured data.

Misal:

Python

```
from agents.tracing import trace, custom_span

with trace(workflow_name="data_pipeline"):
    with custom_span(name="data_validation_step", data={"records_processed": 1000, "errors": 
5}):
        print("Data validation complete.")
```

7. Guardrail Span (`guardrail_span`)

Guardrail checks aur unke results ko trace karta hai.

Python

```
guardrail_span(
    name: str,
    triggered: bool = False,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[GuardrailSpanData]
```

- **`name` (zaroori):** Guardrail ka naam.
- **`triggered` (optional, default: `False`):** Kya guardrail trigger hua.

Misal:

Python

```
from agents.tracing import trace, guardrail_span

with trace(workflow_name="user_input_processing"):
    is_safe = check_for_harmful_content(user_input)
    with guardrail_span(name="content_safety_check", triggered=not is_safe):
        if not is_safe:
            print("Guardrail triggered: Harmful content detected.")
```

8. Transcription Span (`transcription_span`)

Speech-to-text (STT) operations ko trace karta hai.

Python

```python
transcription_span(
    model: str | None = None,
    input: str | None = None,
    input_format: str | None = "pcm",
    output: str | None = None,
    model_config: Mapping[str, Any] | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[TranscriptionSpanData]
```

- **`model` (optional):** STT ke liye istemal hone wala model ka naam.
- **`input` (optional):** Audio input, base64 encoded string ke tor par.
- **`input_format` (optional, default: `"pcm"`):** Audio input ka format.
- **`output` (optional):** Transcription ka output.
- **`model_config` (optional):** Model configuration.

Misal:

Python

```python
from agents.tracing import trace, transcription_span

with trace(workflow_name="voice_assistant"):
    audio_data_b64 = "..." # base64 encoded audio
    with transcription_span(
        model="whisper-1",
        input=audio_data_b64,
        output="Hello, how are you?",
    ):
        print("Audio transcribed.")
```

9. Speech Span (`speech_span`)

Text-to-speech (TTS) operations ko trace karta hai.

Python

```python
speech_span(
    model: str | None = None,
    input: str | None = None,
    output: str | None = None,
    output_format: str | None = "pcm",
    model_config: Mapping[str, Any] | None = None,
    first_content_at: str | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[SpeechSpanData].
```

- `model` **(optional):** TTS ke liye istemal hone wala model ka naam.
- `input` **(optional):** Text input.
- `output` **(optional):** Audio output, base64 encoded string ke tor par.
- `output_format` **(optional, default: `"pcm"`):** Audio output ka format.
- `model_config` **(optional):** Model configuration.
- `first_content_at` **(optional):** Audio output ke pehle byte ka waqt.

Misal:

Python

```
from agents.tracing import trace, speech_span

with trace(workflow_name="voice_assistant"):
    text_to_speak = "I am fine, thank you."
    audio_output_b64 = "..." # base64 encoded audio
    with speech_span(
        model="tts-1",
        input=text_to_speak,
        output=audio_output_b64,
    ):
        print("Text converted to speech.")
```

10. Speech Group Span (`speech_group_span`)

Mutaddid speech operations ko aik logical group mein track karta hai.

Python

```
speech_group_span(
    input: str | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[SpeechGroupSpanData]
```

- `input` **(optional):** Speech request ke liye istemal hone wala input text.

Misal:

Python

```
from agents.tracing import trace, speech_group_span

with trace(workflow_name="interactive_voice_response"):
    with speech_group_span(input="Provide me with account details."):
        # A transcription_span and a speech_span might be nested here
        pass
```

11. MCP Tools Span (`mcp_tools_span`)

MCP server se tools ki list hasil karne ke operation ko trace karta hai.

Python

```
mcp_tools_span(
    server: str | None = None,
    result: list[str] | None = None,
    span_id: str | None = None,
    parent: Trace | Span[Any] | None = None,
    disabled: bool = False,
) -> Span[MCPListToolsSpanData]
```

- **`server` (optional):** MCP server ka naam.
- **`result` (optional):** MCP list tools call ka nateeja (tools ki list).

Misal:

Python

```
from agents.tracing import trace, mcp_tools_span

with trace(workflow_name="tool_discovery"):
    tool_names = ["search_tool", "calculator_tool"]
    with mcp_tools_span(server="my_mcp_server", result=tool_names):
        print("MCP tools listed.")
```

Tracing functions aur spans ka istemal karke, aap apne agent applications ke andar hone wale har important event ki tafseeli log banate hain.

Traces

Trace (Trace)

Tracing module mein, aik **trace** sab se upar (root level) ka object hota hai jo poore logical "workflow" ko represent karta hai. Socho ke yeh aapke agent ya application ke andar hone wale kisi khaas kaam ya operation ki kahani hai. Aik trace ke andar kai chhote-chhote hisse ho sakte hain jinhein "spans" kehte hain.

Trace ke Aham Hissay aur Khasoosiyat

`Trace` aik abstract base class hai, jis ka matlab hai ke yeh sirf aik interface define karta hai, aur iski asal implementation (jaise `TraceImpl`) is interface ko follow karti hai.

- **`trace_id` (property):**
    - Yeh trace ki **munfarid pehchaan (unique identifier)** hai, bilkul jaise kisi file ka naam hota hai. Har trace ka apna aik alag ID hota hai.

- **`name` (property):**
    - Yeh us **workflow ya logical app ka naam** hai jise trace kiya ja raha hai. Misal ke taur par, agar aapka agent coding kar raha hai toh naam "code_bot" ho sakta hai, ya agar woh customer support de raha hai toh "customer_support_agent" hoga. Yeh aapko traces ko categorize karne mein madad karta hai.
- **`start(mark_as_current: bool = False)`:**
    - Yeh method **trace ko shuru karta hai**. Jab aap `start()` ko call karte hain, toh trace recording shuru ho jaati hai.
    - `mark_as_current` parameter ka matlab hai ke agar `True` set kiya jaye, toh yeh trace maujooda "active" trace ban jayega. Is se iske andar banne wale doosre spans automatically is trace se jud jayenge.
- **`finish(reset_current: bool = False)`:**
    - Yeh method **trace ko mukammal karta hai**. Jab workflow khatam ho jaye toh isay call kiya jata hai.
    - `reset_current` parameter ka matlab hai ke agar `True` set kiya jaye, aur yeh trace maujooda active trace ho, toh tracing system isay active state se hata dega.
- **`export() -> dict[str, Any] | None`:**
    - Yeh method **trace ke data ko aik dictionary ki shakal mein export** karta hai. Yeh us waqt mufeed hota hai jab aap trace data ko kahin store karna chahte hain ya analysis ke liye bhejna chahte hain.

Trace ki Implementations

Tracing module mein `Trace` interface ki do khaas implementations hain:

1. **`NoOpTrace` (No-Operation Trace):**
    - Jaisa ke naam se zahir hai, yeh aik **"no-operation" trace** hai. Is ka matlab hai ke agar aap `NoOpTrace` istemal karte hain, toh is trace se mutaliq koi bhi data record nahi hoga.
    - Yeh us waqt mufeed hota hai jab aap tracing ko disable karna chahte hon, ya kuch khaas scenarios mein tracing overhead se bachna chahte hon. Performance testing ya production deployments jahan tracing ki zaroorat na ho, wahan isay istemal kiya ja sakta hai.
2. **`TraceImpl` (Trace Implementation):**
    - Yeh **asal trace implementation** hai jo tracing library ke zariye data record karti hai.
    - Jab aap tracing ko enable karte hain aur `trace()` function istemal karte hain, toh aam taur par background mein `TraceImpl` ka instance hi banaya jata hai. Yeh aapke workflow ke har hisse ko capture karta hai taake aap usay baad mein dekh saken aur analysis kar saken.

Asal Nuqta: Trace aapke agent ke operations ka aik poora tasveer banata hai, jis se aap yeh samajh sakte hain ke agent ne kya kiya, kab kiya, aur uske mukhtalif hisson mein kitna waqt laga. `NoOpTrace` aapko tracing ko asani se band karne ki ijazat deta hai, jabkay `TraceImpl` asal recording ke liye istemal hota hai.

Spans

Span (Span)

Tracing module mein, aik **span** aik trace ke andar aik waqt ke lehaz se mukhtasir operation ko represent karta hai. Jahan aik **trace** aik poore "workflow" ki kahani sunata hai, wahin aik **span** us kahani ke andar ke har chote event ya qadam ko tafseel se bayan karta hai. Spans nested ho sakte hain, jis se operations ki hierarchy aur flow ko samajhna asaan ho jata hai.

Span ke Aham Hissay aur Khasoosiyat

`Span` aik abstract base class hai, jis ka matlab hai ke yeh sirf aik interface define karta hai, aur iski asal implementations (jaise `SpanImpl`) is interface ko follow karti hain.

- **`start(mark_as_current: bool = False):`**
  - Yeh method **span ko shuru karta hai**. Jab aap `start()` ko call karte hain, toh span ki recording shuru ho jati hai.
  - `mark_as_current` parameter ka matlab hai ke agar `True` set kiya jaye, toh yeh span maujooda "active" span ban jayega. Is se iske andar banne wale doosre spans automatically is span ke child ban jayenge. Yeh nesting (teh-ba-teh) ki bunyad hai.
- **`finish(reset_current: bool = False) -> None:`**
  - Yeh method **span ko mukammal karta hai**. Jab woh khaas operation ya event khatam ho jaye toh isay call kiya jata hai.
  - `reset_current` parameter ka matlab hai ke agar `True` set kiya jaye, aur yeh span maujooda active span ho, toh tracing system isay active state se hata dega. Yeh context manager (`with span(...):`) ke istemal mein khud ba khud hota hai.

Span ki Implementations

Tracing module mein `Span` interface ki do khaas implementations hain:

1. **`NoOpSpan` (No-Operation Span):**
   - Jaisa ke naam se zahir hai, yeh aik **"no-operation" span** hai. Is ka matlab hai ke agar aap `NoOpSpan` istemal karte hain, toh is span se mutaliq koi bhi data record nahi hoga.
   - Yeh us waqt mufeed hota hai jab tracing globally disabled ho, ya aap kuch khaas operations ko trace nahi karna chahte hon. Yeh performance overhead ko kam karne mein madad karta hai.
2. **`SpanImpl` (Span Implementation):**
   - Yeh **asal span implementation** hai jo tracing library ke zariye data record karti hai.
   - Jab aap tracing ko enable karte hain aur `agent_span()`, `function_span()`, ya kisi bhi doosre span creation function ka istemal karte hain, toh aam taur par background mein `SpanImpl` ka instance hi banaya jata hai. Yeh aapke agent ke har chote amal ki tafseelat ko capture karta hai, masalan LLM call ka input/output, tool invocation, ya guardrail check.

Asal Nuqta: Spans traces ke andar ke building blocks hain jo aapko apne agent ke andar ke har mukhtasir event ko monitor karne ki ijazat dete hain. Yeh aapke agent ke run-time behaviour ki aik daqeeq aur tafseeli tasveer faraham karte hain, jo debugging aur performance analysis ke liye behtareen hai.

Processor interface

TracingProcessor (Tracing Processor)

`TracingProcessor` Agents SDK ke tracing module mein aik **interface** hai jo aapko traces aur spans ke lifecycle events par **custom logic add karne** ki ijazat deta hai. Socho ke yeh aik "event listener" hai jo tracing system mein hone wale aham waqiyat (events) ko sunta hai aur un par amal karta hai.

Aham Khasoosiyat aur Methods

`TracingProcessor` aik abstract base class hai, jis ka matlab hai ke aapko iski sub-classes banani hongi aur in sab abstract methods ko implement karna hoga.

- **`on_trace_start(trace: Trace) -> None`**:
  - Yeh method us waqt **call kiya jata hai jab koi naya trace shuru hota hai**.
  - **`trace`**: Woh `Trace` object jo abhi shuru hua hai.
  - Aap is method ko istemal kar sakte hain trace ke shuru hone par koi initial setup karne ke liye, ya trace ki maloomat (masalan `trace_id` aur `name`) ko log karne ke liye.
- **`on_trace_end(trace: Trace) -> None`**:
  - Yeh method us waqt **call kiya jata hai jab koi trace mukammal hota hai**.
  - **`trace`**: Woh `Trace` object jo abhi khatam hua hai.
  - Yeh method trace ke khatam hone par final processing ya data collection ke liye mufeed hai, masalan poore trace ki tafseelat ko database mein save karna.
- **`on_span_start(span: Span[Any]) -> None`**:
  - Yeh method us waqt **call kiya jata hai jab koi naya span shuru hota hai**.
  - **`span`**: Woh `Span` object jo abhi shuru hua hai.
  - Yahan aap span ke shuru hone par uski maloomat ko log kar sakte hain ya performance metrics collect karna shuru kar sakte hain.
- **`on_span_end(span: Span[Any]) -> None`**:
  - Yeh method us waqt **call kiya jata hai jab koi span mukammal hota hai**.
  - **`span`**: Woh `Span` object jo abhi khatam hua hai.
  - **Ahem Note**: Yeh method **block nahi karna chahiye aur na hi ismein exceptions raise karni chahiye**. Isay tezi se amal karne ke liye design kiya gaya hai taake yeh application ke performance ko mutasir na kare. Yeh span ke data ko process karne, usay aggregate karne, ya agay ke processing pipeline mein bhejne ke liye istemal hota hai.
- **`shutdown() -> None`**:
  - Yeh method us waqt **call kiya jata hai jab application rukti hai** (ya tracing provider band hota hai).
  - Isay kisi bhi pending resources ko clean up karne, ya buffers mein maujood data ko flush karne ke liye istemal kiya jata hai taake koi data zaya na ho.
- **`force_flush() -> None`**:
  - Yeh method **tamam queued spans/traces ko fauri tor par flush karne par majboor karta hai**.

  - Yeh debugging ya test scenarios mein mufeed ho sakta hai jahan aap data ko foran dekhna chahte hain, ya jab application shutdown hone se pehle data ki guarantee karna chahte hain.

TracingExporter (Tracing Exporter)

`TracingExporter` aik aur **interface** hai jo traces aur spans ko **export karne** ki zimmedari leta hai. Jab `TracingProcessor` events ko receive karta hai aur un par amal karta hai, toh `TracingExporter` un processed events ko bahar ki kisi service ya storage system mein bhejta hai. Masalan, yeh traces ko console par log kar sakta hai, ya kisi monitoring backend (jaise OpenTelemetry collector) par bhej sakta hai.

Aham Khasoosiyat aur Methods

- **`export(items: list[Trace | Span[Any]]) -> None`**:
  - Yeh method **traces aur spans ki list ko export karta hai**.
  - **`items`**: Traces aur spans ki list jinhein export karna hai. Yeh `Trace` objects aur `Span` objects ka mix ho sakta hai.
  - Is method ki implementation yeh tay karti hai ke data kahan aur kaise bheja jaye ga – masalan, files mein save karna, network par bhejna, ya database mein store karna.

Asal Nuqta:

- **`TracingProcessor`** aapko tracing events (trace/span start/end) par **custom logic chalane** ki ijazat deta hai.
- **`TracingExporter`** aapko processed tracing data ko **bahar ke system mein bhejne** (export karne) ki sahoolat deta hai.

Yeh dono interfaces mil kar aik flexible aur powerful tracing system banate hain jo aapko apne agents ki performance aur behaviour ko gehrai se samajhne mein madad karta hai

Processors

Tracing Processors and Exporters (Tracing Processors aur Exporters)

Agents SDK ke tracing module mein, **processors** aur **exporters** do aham components hain jo yeh tay karte hain ke tracing data ko kaise handle aur kahan bheja jaye.

- **Processors:** Traces aur spans ke start/end events ko receive karte hain aur unhein processing ke liye tayar karte hain.
- **Exporters:** Tayar shuda tracing data ko kisi muta'alliq service ya destination tak pahunchate hain.

Exporters (Data Bhejne Wale)

Exporters woh classes hain jo actual tracing data (traces aur spans) ko bahar ki dunya mein bhejti hain, yaani unhein display karti hain, log karti hain, ya kisi backend service par upload karti hain. Yeh `TracingExporter` interface ko implement karte hain.

1. `ConsoleSpanExporter` (Console Span Exporter)

`ConsoleSpanExporter` aik simple exporter hai jo traces aur spans ko **console par print** kar deta hai. Yeh debugging aur local development ke liye behad mufeed hai jab aap fauri taur par dekhna chahte hain ke aapka agent kya kar raha hai.

- **Buniyadi Kaam:** Tracing data ko standard output (console) par dikhana.
- **Istemal:** Sirf dekhtay waqt ke liye, production logging ke liye ideal nahi hai.

2. `BackendSpanExporter` (Backend Span Exporter)

`BackendSpanExporter` traces aur spans ko **remote backend service par bhejta hai**, khaas taur par OpenAI ke tracing ingestion endpoint par. Yeh production environments ke liye design kiya gaya hai jahan aapko centralized tracing data collection aur analysis ki zaroorat hoti hai.

- **`__init__(...)`** (Constructor):
  - `api_key` (optional): **Authorization header ke liye API key**. Agar provide nahi ki jaye, toh `OPENAI_API_KEY` environment variable se utha li jayegi. Yeh wohi key hai jo OpenAI Python client istemal karta hai.
  - `organization` (optional): OpenAI organization ID. Agar provide nahi ki jaye, toh `OPENAI_ORG_ID` environment variable se utha li jayegi.
  - `project` (optional): OpenAI project ID. Agar provide nahi ki jaye, toh `OPENAI_PROJECT_ID` environment variable se utha li jayegi.
  - `endpoint` (default: `"https://api.openai.com/v1/traces/ingest"`): Woh HTTP endpoint jahan traces/spans POST kiye jayenge.
  - `max_retries` (default: `3`): Failures par dobara koshishon ki ziyada se ziyada tadaad.
  - `base_delay` (default: `1.0`): Pehle backoff ke liye buniyadi delay (seconds mein).
  - `max_delay` (default: `30.0`): Backoff growth ke liye ziyada se ziyada delay (seconds mein).
- **`set_api_key(api_key: str)`**: Exporter ke liye OpenAI API key set karta hai.
- **`close()`**: Underlying HTTP client ko band karta hai.
- **Buniyadi Kaam:** Tracing data ko API ke zariye ek centralized backend mein bhejna, jahan ise store, visualize aur analyze kiya ja sake. Ismein network communication, retries aur error handling shamil hai.

Processors (Data Tayar Karne Wale)

Processors `TracingProcessor` interface ko implement karte hain aur yeh tay karte hain ke tracing events (span/trace start/end) ko kaise handle kiya jaye.

1. `BatchTraceProcessor` (Batch Trace Processor)

`BatchTraceProcessor` aik aam istemal hone wala processor hai jo tracing data ko **batches mein jama karta hai aur phir unhein export karta hai**. Yeh is liye kiya jata hai taake performance par kam se kam asar pare, kyunki har span ko alag se network par bhejne se behtar hai ke unka batch bana kar bheja jaye.

- **Implementation Notes:**
  1. **Queue ka Istemal:** Yeh data ko jama karne ke liye `Queue` ka istemal karta hai, jo **thread-safe** hoti hai. Iska matlab hai ke mutaddid threads baghair kisi masle ke ismein data add kar sakti hain.
  2. **Background Thread:** Spans ko export karne ke liye aik **background thread** istemal karta hai. Is se application ki main execution block nahi hoti aur performance issues kam se kam hote hain.
  3. **Memory Storage:** Spans ko export hone tak **memory mein store** kiya jata hai.
- **`__init__(...)`** (Constructor):
  - `exporter` (zaroori): Woh `TracingExporter` instance jise data bhejne ke liye istemal kiya jayega (masalan, `BackendSpanExporter`).
  - `max_queue_size` (default: `8192`): Queue mein store karne ke liye spans ki ziyada se ziyada tadaad. Agar is se ziyada spans aayen toh purane spans drop hona shuru ho jayenge.
  - `max_batch_size` (default: `128`): Aik hi batch mein export karne ke liye spans ki ziyada se ziyada tadaad.
  - `schedule_delay` (default: `5.0`): Naye spans ko export karne ke liye checks ke darmiyan delay (seconds mein).
  - `export_trigger_ratio` (default: `0.7`): Queue size ka woh ratio jis par export trigger kiya jayega. Masalan, agar queue 70% bhar jaye toh batch export shuru ho jayega, bhale hi `schedule_delay` abhi khatam na hua ho.

`shutdown(timeout: float | None = None)`:

- Jab application rukti hai toh call kiya jata hai. Yeh background thread ko rukne ka signal deta hai aur phir uske complete hone ka intazar karta hai.

`force_flush()`:

- Tamam queued spans ko **fauri taur par flush** karta hai. Yeh us waqt mufeed hai jab aap application band karne se pehle yeh yakeeni banana chahte hain ke tamam data bheja ja chuka hai.

Default Configuration (Default Configuration)

Tracing module kuch conveniance functions bhi faraham karta hai:

- **`default_exporter() -> BackendSpanExporter`:**
  - Woh **default exporter** return karta hai jo traces aur spans ko backend mein batches ki shakal mein export karta hai. Aam taur par yeh `BackendSpanExporter` ka aik pre-configured instance hoga.
- **`default_processor() -> BatchTraceProcessor`:**
  - Woh **default processor** return karta hai jo traces aur spans ko backend mein batches ki shakal mein export karta hai. Aam taur par yeh `BatchTraceProcessor` ka aik pre-configured instance hoga jo `default_exporter()` ko istemal karta hai.

Asal Nuqta:

Yeh classes mil kar ek mazboot aur scalable tracing system banati hain. `ConsoleSpanExporter` local debugging ke liye mufeed hai, jabkay `BackendSpanExporter` aur `BatchTraceProcessor` production-ready tracing collection ke liye zaroori hain, jo performance ko optimize karte hain aur data loss ko kam karte hain.

Scope

Scope (Scope)

Tracing module mein, **Scope** ka buniyadi maqsad **current active span aur trace ko manage karna** hai, yaani yeh tay karna ke kisi bhi waqt konsa span ya trace "current" samjha jayega. Yeh tracing mein **context propagation** (context ko agay barhana) ke liye bohat zaroori hai.

Socho ke aap aik kitaab padh rahe hain, aur har chapter aik **trace** hai, jabkay us chapter ke andar ke har paragraph ya jumla aik **span** hai. "Scope" yeh tay karta hai ke aap is waqt kitaab ke kis jumle ya paragraph par hain. Jab aap aik naya paragraph shuru karte hain, toh woh aapka "current" span ban jata hai. Jab aap woh paragraph khatam kar lete hain, toh aap wapis pichle paragraph (parent span) par chale jate hain, ya agar woh chapter ka aakhri paragraph tha toh chapter (trace) khatam ho jata hai.

Kaam aur Zaroorat

Jab aap apne code mein `with trace(...)` ya `with span(...)` istemal karte hain, toh `Scope` class background mein kaam karti hai.

1. **Context Management:** `Scope` yeh yakeeni banata hai ke jab aap `trace()` ya `create_span()` jaisi functions ko call karte hain (khaas tor par jab `parent` argument nahi diya jata), toh naya banne wala span ya trace **automatically sahi parent se connect ho jaye**. Yeh tracing hierarchy ko barqarar rakhta hai.
2. **Thread-Safety (aam taur par):** Behtar tracing systems mein, `Scope` aam taur par **thread-safe** hota hai. Iska matlab hai ke agar aapka application multi-threaded hai (aik hi waqt mein kai kam kar raha hai), toh har thread ka apna alag tracing context (current trace/span) hoga, aur woh aik doosre ke context mein mudakhlat nahi karega.
3. **Automatic Parentage:** Jab koi naya span shuru hota hai, toh `Scope` yeh dekhne mein madad karta hai ke kya koi **maujooda active span ya trace** hai. Agar hai, toh naya span uska **child** ban jata hai. Yeh traces ki tree-like structure banata hai, jahan aap dekh sakte hain ke kon sa operation kis bade operation ka hissa tha.
4. **Cleanup:** Jab aik span ya trace `finish()` hota hai (ya `with` block se bahar nikla jata hai), toh `Scope` yeh yakeeni banata hai ke woh ab "current" nahi rahega, aur control wapis uske parent par chala jayega (agar koi ho).

Misal ke Zariye Samajhna

Jab aap code likhte hain:

Python

```python
from agents.tracing import trace, function_span, generation_span

def do_complex_task():
    with trace(workflow_name="main_workflow"):  # Trace shuru hua, yeh current trace ban gaya
        print("Main workflow started.")

        with function_span(name="prepare_data"): # Function span shuru hua, iska parent
'main_workflow' hai, yeh current span ban gaya
            print("Preparing data...")
            # ... data preparation logic ...
        # function_span khatam hua, 'main_workflow' dobara current trace/parent ban gaya
        with generation_span(model="my_llm"): # Generation span shuru hua, iska parent bhi
'main_workflow' hai
            print("Generating response...")
            # ... LLM call ...
        # generation_span khatam hua

        print("Main workflow finished.")
    # Trace khatam hua, ab koi current trace/span nahi.
```

Upar di gayi misal mein:

- Jab `trace("main_workflow")` shuru hota hai, toh `Scope` isay **current trace** ke tor par set karta hai.
- Jab `function_span("prepare_data")` shuru hota hai, toh `Scope` dekhta hai ke "main_workflow" **current trace** hai, is liye yeh `function_span` ko "main_workflow" ka child banata hai aur `function_span` ko **current span** ke tor par set karta hai.
- Jab `function_span` khatam hota hai, `Scope` isay current span se hata deta hai, aur "main_workflow" dobara **current trace** ban jata hai (kyunkay yeh uska parent tha).
- Yahi amal `generation_span` ke liye bhi dohraya jata hai.
- Jab `trace("main_workflow")` khatam hota hai, toh `Scope` tracing context ko saaf kar deta hai.

Khulasa

Scope tracing system ka woh hissa hai jo **current active trace aur span ko track karta hai**, jis se tracing events ki sahi hierarchy aur context propagation yakeeni hoti hai. Yeh aapke traces ko aik munasib tarteeb aur nested structure mein record karne ke liye zaroori hai.

Setup

Global Trace Provider Ko Set Aur Hasil Karna (Setting and Getting the Global Trace Provider)

Tracing module mein, **Trace Provider** (trace faraham karne wala) aik behtar concept hai jo yeh tay karta hai ke traces aur spans ko asal mein kaise banaya aur manage kiya jaye. Jab aap `trace()` ya `agent_span()` jaisi functions ko call karte hain, toh woh background mein isi global trace provider ko istemal karte hain. Yeh setup functions aapko is global provider ko control karne ki ijazat dete hain.

`set_trace_provider(provider: TraceProvider) -> None`

Yeh function **global trace provider ko set karta hai** jo tracing utilities ke zariye istemal hota hai.

- **`provider`**: Aapko aik aisa object faraham karna hoga jo **`TraceProvider`** **abstract base class ko implement karta ho**.
- **Kaam:** Aapke application ko batata hai ke tracing data banane aur manage karne ke liye kaun si khaas implementation (jaise ke default ya custom) istemal karni hai.
- **Zaroorat:**
    - **Custom Tracing Implementation:** Agar aap apni tracing logic banate hain (masalan, kisi khaas backend ya format ke liye), toh aap apni `TraceProvider` class banayenge aur usay is function ke zariye register karenge.
    - **Testing:** Testing ke dauran, aap aik mock ya dummy `TraceProvider` set kar sakte hain taake actual tracing backend se connection na ho.

    - **Default Behavior Override:** Agar aap default tracing setup ko tabdeel karna chahte hain (masalan, tracing ko mukammal taur par disable karna), toh aap `NoOpTraceProvider` (agar dastiyab ho) set kar sakte hain.

Misal:

Python

```
from agents.tracing.setup import set_trace_provider
from agents.tracing.provider import TraceProvider # Assuming you have an implementation

# Suppose you have a custom TraceProvider implementation
class MyCustomTraceProvider(TraceProvider):
    # ... all abstract methods implemented here ...
    pass

# Initialize your custom provider
my_provider = MyCustomTraceProvider()

# Set it as the global provider
set_trace_provider(my_provider)

# Ab is ke baad jitne bhi traces aur spans banenge, woh 'my_provider' ke zariye handle honge.
```

`get_trace_provider() -> TraceProvider`

Yeh function **global trace provider ko hasil karta hai** jo tracing utilities ke zariye istemal ho raha hai.

- **Returns:** Maujooda active `TraceProvider` object.
- **Kaam:** Aapko yeh janne ki ijazat deta hai ke aapka application tracing ke liye kaun si specific provider implementation istemal kar raha hai.
- **Zaroorat:**
    - **Configuration Check:** Aap run-time par dekh sakte hain ke tracing sahi tarah se configured hai ya nahi.
    - **Advanced Scenarios:** Kuch advanced scenarios mein, aapko directly provider ke methods ko access karne ki zaroorat pad sakti hai.

Misal:

Python

```
from agents.tracing.setup import get_trace_provider

# Get the currently active trace provider
current_provider = get_trace_provider()

# Aap ab is provider ke methods ko access kar sakte hain, masalan:
# current_provider.set_disabled(True)
# print(f"Current trace provider type: {type(current_provider)}")
```

Asal Nuqta:

`set_trace_provider` aur `get_trace_provider` functions aapko tracing mechanism par control dete hain. Aap apni marzi ka tracing setup (console output, backend integration, ya no-op) chun sakte hain aur ise application-wide apply kar sakte hain.

Span data

SpanData Classes (Span Data Classes)

Tracing module mein, **`SpanData`** aik **abstract base class** hai jo trace ke andar hone wale mukhtalif operations ki tafseelat ko represent karti hai. Har **span** (aik chhota operation) ke paas aik makhsus `SpanData` object hota hai jo us operation ke baray mein khaas maloomat rakhta hai. Socho yeh aik report form ki tarah hai, jahan har form (SpanData ki child class) aik mukhtalif qism ki report (span) ke liye khaas field rakhti hai.

`SpanData` (Buniyadi Class)

Yeh tamam khaas `SpanData` classes ke liye buniyadi interface hai.

- **`type` (abstractmethod property): `str`**
  - Har `SpanData` class ko yeh property provide karni chahiye jo us **span ki type** batati hai. Masalan, "agent", "function", "generation", waghera.
- **`export() -> dict[str, Any]` (abstractmethod):**
  - Yeh method **span data ko aik dictionary ki shakal mein export** karta hai. Yeh us waqt mufeed hota hai jab aap span ki maloomat ko log karna chahte hain, ya kisi external system par bhejna chahte hain, jahan unhein asani se process kiya ja sake.

Mukhtalif Qism Ki `SpanData` Classes

Har `SpanData` class aik makhsus qism ke operation ko represent karti hai aur us operation se mutaliq khaas maloomat ko store karti hai.

1. **`AgentSpanData`**
   - **Agent Span** ko represent karta hai.
   - Shamil hain: **`name`** (agent ka naam), **`handoffs`** (un agents ki list jinhein yeh control de sakta hai), **`tools`** (dastiyab tool names ki list), aur **`output_type`** (agent ke zariye paida kiye gaye output ki type).
   - **Misal:** Jab aik agent koi decision leta hai ya koi action karta hai.
2. **`FunctionSpanData`**
   - Aik **Function Span** ko represent karta hai.
   - Shamil hain: **`input`** (function ka input), **`output`** (function ka output), aur **`mcp_data`** (agar Multi-Context Planner se mutaliq ho).
   - **Misal:** Jab agent kisi external ya internal function ko call karta hai.
3. **`GenerationSpanData`**
   - Aik **Generation Span** ko represent karta hai. Yeh khaas taur par **Large Language Models (LLMs)** se honay wali interactions ke liye hai.
   - Shamil hain: **`input`** (model ko bheje gaye input messages ki sequence), **`output`** (model se hasil kiye gaye output messages ki sequence), **`model`** (istemal hone wala model identifier), **`model_config`** (model ki hyperparameters), aur **`usage`** (input/output tokens jaisi usage information).
   - **Misal:** Jab agent LLM se response generate karwata hai.
4. **`ResponseSpanData`**

- Aik **Response Span** ko represent karta hai. Yeh aam taur par kisi API call ke **response object** ko record karta hai.
- Shamil hain: **`response`** (OpenAI `Response` object), aur **`input`**.
- **Misal:** Jab LLM se milne wale mukammal response object ko trace karna ho.

5. **`HandoffSpanData`**
    - Aik **Handoff Span** ko represent karta hai.
    - Shamil hain: **`source`** (agent jo handoff kar raha hai), aur **`destination`** (agent jo handoff receive kar raha hai).
    - **Misal:** Jab aik automated agent user ko human agent ko hand off karta hai.
6. **`CustomSpanData`**
    - Aik **Custom Span** ko represent karta hai. Yeh aapko apni marzi ki maloomat ko trace karne ki ijazat deta hai.
    - Shamil hain: **`name`** (custom span ka naam), aur **`data`** (arbitrary structured data ki dictionary).
    - **Misal:** Kisi bhi aisi custom logic ya operation ko trace karna jiske liye koi mukhtasir `SpanData` class mojood na ho.
7. **`GuardrailSpanData`**
    - Aik **Guardrail Span** ko represent karta hai.
    - Shamil hain: **`name`** (guardrail ka naam), aur **`triggered`** (bool, kya guardrail trigger hua).
    - **Misal:** Jab koi safety ya compliance check kiya jata hai.
8. **`TranscriptionSpanData`**
    - Aik **Transcription Span** ko represent karta hai. Yeh Speech-to-Text (STT) operations ke liye hai.
    - Shamil hain: **`input`** (audio input), **`output`** (transcribed text), **`model`** (STT model ka naam), aur **`model_config`**.
    - **Misal:** Jab audio ko text mein convert kiya jata hai.
9. **`SpeechSpanData`**
    - Aik **Speech Span** ko represent karta hai. Yeh Text-to-Speech (TTS) operations ke liye hai.
    - Shamil hain: **`input`** (text input), **`output`** (generated audio), **`model`** (TTS model ka naam), **`model_config`**, aur **`first_content_at`** (audio output ke pehle byte ka timestamp).
    - **Misal:** Jab text ko audio mein convert kiya jata hai.
10. **`SpeechGroupSpanData`**
    - Aik **Speech Group Span** ko represent karta hai. Yeh mutaddid speech-related spans ko aik logical group mein jama karne ke liye hai.
    - **Misal:** Aik single voice interaction jismein transcription aur phir speech generation shamil ho.
11. **`MCPListToolsSpanData`**
    - Aik **MCP List Tools Span** ko represent karta hai. Yeh Multi-Context Planner (MCP) se tools ki list hasil karne ke operation ko track karta hai.
    - Shamil hain: **`server`** (MCP server ka naam), aur **`result`** (tools ki list).
    - **Misal:** Jab agent ko available tools ki list chahiye hoti hai.

Asal Nuqta: Yeh `SpanData` classes tracing module ko yeh ijazat deti hain ke woh har operation ke baray mein **specific aur structured maloomat** record kar sake. Yeh data baad mein debugging, performance analysis, aur agent ke behaviour ko samajhne ke liye behad qeemti hota hai.

Util

Utility Functions for Tracing (Tracing Ke Liye Utility Functions)

Tracing module mein kuch madadgar utility functions shamil hain jo traces aur spans ke liye IDs generate karne aur waqt ko format karne mein kaam aate hain. Yeh functions tracing data ko consistent aur standard format mein rakhne mein madad karte hain.

`time_iso()`

Python

```
time_iso() -> str
```

Yeh function **maujooda waqt (current time) ko ISO 8601 format mein return karta hai**.

- **Kaam:** Tracing mein, spans aur traces ke start aur end times ko record karna zaroori hota hai. ISO 8601 aik international standard hai jo tareekh aur waqt ko aik makhsus format mein bayan karta hai, masalan `2025-06-18T12:34:21.000Z`. Yeh format data ko parse karna aur alag-alag systems ke darmiyan share karna asaan banata hai.
- **Istemal:** Jab bhi aapko kisi tracing event ka precise timestamp record karna ho, toh aap is function ka istemal kar sakte hain.

`gen_trace_id()`

Python

```
gen_trace_id() -> str
```

Yeh function **aik naya trace ID generate karta hai**.

- **Kaam:** Har trace (jo aik mukammal workflow ko represent karta hai) ko aik **munfarid pehchaan (unique identifier)** ki zaroorat hoti hai. `gen_trace_id()` yeh munfarid ID banata hai, jis se aap specific workflows ko track aur analyze kar sakte hain.
- **Istemal:** Jab aap `trace()` function istemal karte hain aur `trace_id` argument provide nahi karte, toh aam taur par yeh function background mein trace ID generate karne ke liye istemal hota hai. Aap ise manually bhi call kar sakte hain agar aap khud se trace IDs manage karna chahte hain.

gen_span_id()

Python

```
gen_span_id() -> str
```

Yeh function **aik naya span ID generate karta hai**.

- **Kaam:** Trace ke andar, har span (jo aik chhota operation hai) ko bhi aik **munfarid ID** ki zaroorat hoti hai. Yeh ID us span ko uske trace ke andar pehchanne mein madad karti hai.
- **Istemal:** Jab aap `agent_span()`, `function_span()`, ya doosre span creation functions istemal karte hain aur `span_id` argument provide nahi karte, toh yeh function background mein span ID generate karne ke liye istemal hota hai. Agar aap apne span IDs ko customize karna chahte hain toh ise manually call kiya ja sakta hai.

---

gen_group_id()

Python

```
gen_group_id() -> str
```

Yeh function **aik naya group ID generate karta hai**.

- **Kaam:** Group ID aik **optional identifier** hai jo mutaddid traces ko aik hi logical group se jorne ke liye istemal hota hai, masalan aik hi user session, aik chat conversation, ya aik bade background process ke mutaddid runs. Yeh aapko related traces ko aik sath dekhne aur analysis karne mein madad karta hai.
- **Istemal:** Jab aap `trace()` function istemal karte hain aur `group_id` argument provide nahi karte, toh yeh function background mein group ID generate karne ke liye istemal hota hai. Aap ise manually bhi call kar sakte hain.

Asal Nuqta: Yeh utility functions tracing system ke liye **IDs ki generation aur waqt ke standard representation** ko aasan banate hain, jo tracing data ki consistency aur usability ke liye zaroori hain.

VOICE

Pipeline

VoicePipeline (وائس پائپ لائن)

VoicePipeline ek khaas tarah se design kiya gaya voice agent pipeline hai. Iska maqsad audio par mabni agents ke liye mukammal (end-to-end) flow ko manage karna hai. Yeh ek "opinionated" (yani kuch khaas tareeqon ko tarjeeh dena wala) aur structured tareeqa istemal karta hai taake audio input ko handle kiya ja sake, ek workflow chalaya ja sake, aur phir jawab ko audio output mein tabdeel kiya ja sake.

Yeh pipeline teen aham marhalon mein kaam karta hai:

1. **Audio Input ki Transcription (آڈیو اِن پٹ کی ٹرانسکرپشن):** Sab se pehle, yeh aapke bole gaye الفاظ ya audio input ko likhi hui text mein badalta hai.
2. **Faraham Karda Workflow Chalana (فراہم کردہ ورک فلو چلانا):** Iske baad, yeh text ko aapke agent ya kisi bhi diye gaye workflow (jise VoiceWorkflowBase kaha gaya hai) ko deta hai. Yeh workflow text jawabat ka ek silsila paida karta hai.
3. **Text Jawabat ko Streaming Audio Output mein Tabdeel Karna (ٹیکسٹ جوابات کو اسٹریمنگ آڈیو آؤٹ پٹ میں تبدیل کرنا)::** Aakhir mein, jo text jawabat workflow ne paida kiye hain, unhein do□□□□ audio mein tabdeel kiya jata hai, jise streaming ke zariye chalaya ja sakta hai.

__init__ (Constructor)

VoicePipeline ka constructor ise setup karta hai. Yeh batata hai ke kaun sa workflow chalana hai aur kaun se speech models istemal karne hain.

Python

```
__init__(
    *,
    workflow: VoiceWorkflowBase,
    stt_model: STTModel | str | None = None,
    tts_model: TTSModel | str | None = None,
    config: VoicePipelineConfig | None = None,
)
```

- `workflow` (Zaroori):
    - **Type:** `VoiceWorkflowBase`
    - **Tafseel:** Yeh woh **workflow** hai jo audio input se badle gaye text ko milne par pipeline chalayega. Yeh aapke agent ki asal logic hai jo user ke sawal ka jawab deti hai ya koi action karti hai. `VoiceWorkflowBase` ek bunyadi class hai jis se aapke custom voice workflow ko inherit karna hoga.
- `stt_model` (Optional):
    - **Type:** `STTModel` | `str` | `None`
    - **Tafseel:** Yeh woh **Speech-to-Text (STT) model** hai jo audio input ko text mein tabdeel karne ke liye istemal hoga. Agar yeh nahi diya gaya, to pipeline **default OpenAI model** istemal karegi. Aap ek khaas `STTModel` object ya model ka naam (string mein) de sakte hain.
- `tts_model` (Optional):
    - **Type:** `TTSModel` | `str` | `None`
    - **Tafseel:** Yeh woh **Text-to-Speech (TTS) model** hai jo text jawabat ko audio output mein tabdeel karne ke liye istemal hoga. Agar yeh nahi diya gaya, to pipeline **default OpenAI model** istemal karegi. Aap ek khaas `TTSModel` object ya model ka naam (string mein) de sakte hain.
- `config` (Optional):
    - **Type:** `VoicePipelineConfig` | `None`
    - **Tafseel:** Yeh pipeline ki **configuration (tarteebat)** ke liye hai. Agar yeh nahi di gayi, to **default configuration** istemal hogi. Is mein pipeline ke behaviour ko control karne wali mazeed tarteebat shamil ho sakti hain.

`run` (Run)

Yeh `async` (asynchronous) function voice pipeline ko chalata hai aur process shuda audio output wapas karta hai.

Python

```
async run(
    audio_input: AudioInput | StreamedAudioInput,
) -> StreamedAudioResult
```

- `audio_input` (Zaroori):
    - **Type:** `AudioInput` | `StreamedAudioInput`
    - **Tafseel:** Process karne ke liye **audio input**. Yeh ya to ek `AudioInput` instance ho sakta hai, jo ek single **static buffer** hai (yaani ek mukammal audio file jo pehle se available ho), ya ek `StreamedAudioInput` instance ho sakta hai, jo **audio data ka ek stream** hai jis mein aap mazeed data add kar sakte hain (yaani real-time mein aane wala audio).
- **Returns (Wapsi):**
    - **Type:** `StreamedAudioResult`
    - **Tafseel:** Ek `StreamedAudioResult` instance. Aap is object ko **audio events ko stream karne aur unhein play karne** ke liye istemal kar sakte hain. Yeh ek handle hai jis ke zariye aap user ko jawab ki awaaz ko streaming andaaz mein chala sakte hain, jese hi woh tayar hoti hai.

Khulasa (Summary)

`VoicePipeline` Agent SDK mein **awaaz par mabni guftugu ke liye ek mukammal framework** faraham karta hai. Yeh Speech-to-Text, aapke agent ki logic, aur Text-to-Speech ko asaani se jorta hai, jis se aap ko interactive voice agent banane mein madad milti hai. Yeh khaas taur par real-time ya streaming audio ke scenarios ke liye design kiya gaya hai, jahan fori jawab dena zaroori hai.

Workflow

Voice Workflows (فلو ورک وائس)

`VoicePipeline` ka dil (core) us ka **workflow** hota hai. Yeh woh jagah hai jahan aap apne voice agent ki asal "dimaghi" logic likhte hain. `VoiceWorkflowBase` ek bunyadi class hai jo is logic ko define karti hai.

`VoiceWorkflowBase` ()

`VoiceWorkflowBase` ek **abstract base class** hai jo voice workflows ke liye ek structure provide karti hai. Iska matlab hai ke aapko is class ko inherit karke apna custom workflow banana hoga aur iske `run` method ko zaroor implement karna hoga.

- **Workflow kya hai?**
  - Ek "workflow" koi bhi code ho sakta hai jo **audio transcription (matn mein tabdeel shuda awaaz)** ko input ke taur par receive karta hai aur phir **text yield karta hai** (yani paida karta hai). Is text ko baad mein text-to-speech (TTS) model ke zariye awaaz mein badal kar user ko sunaya jayega.
  - Zyadatar cases mein, aap **Agents** banayenge aur unhein `Runner.run_streamed()` ke zariye chalayenge. Is se aapko text events ka ek stream milega, aur aap us stream se kuch ya sara text hasil karke yield kar sakte hain.

`run` (Rana) - Abstract Method

Python

```
run(transcription: str) -> AsyncIterator[str]
```

Yeh `VoiceWorkflowBase` class ka **abstract method** hai, jise aapko apni custom workflow class mein zaroor implement karna hoga.

- **Input:**
  - `transcription: str`: Aapko **user ki awaaz ki transcription** (yani woh text jo user ne bola hai) input ke taur par milegi.
- **Output:**
  - `-> AsyncIterator[str]`: Aapko **text yield karna hoga** jo user ko bola jayega. Iska matlab hai ke aap ek-ek text chunk ko step by step paida kar sakte hain, jo TTS model ko bheja jayega aur user ko real-time mein sunaya jayega.
- **Logic:** Aap is method ke andar apni marzi ki koi bhi logic chala sakte hain.
  - **Common Use Case:** Zyada tar cases mein, aap **`Runner.run_streamed()`** ko call karenge aur us stream se **text events ko yield karenge**. Is ke liye `VoiceWorkflowHelper` class madadgar sabit ho sakti hai.

VoiceWorkflowHelper ()

VoiceWorkflowHelper ek madadgar class hai jo RunResultStreaming objects se text events nikalne mein madad karti hai.

stream_text_from (□□□□□□ □□□□□ □□□□) - Async Class Method

Python

```
async stream_text_from(
    result: RunResultStreaming,
) -> AsyncIterator[str]
```

- Yeh ek RunResultStreaming object ko wrap karta hai aur us stream se **text events ko yield karta hai**.
- **Kaam:** Jab aap Runner.run_streamed() istemal karte hain, to woh RunResultStreaming object return karta hai jismein mukhtalif qism ke events (text, tool calls, etc.) shamil ho sakte hain. Yeh helper function sirf text events ko filter karke aapko asani se deta hai, taake aap unhein TTS model ko bhej saken.

SingleAgentWorkflowCallbacks ()

Yeh ek class hai jo SingleAgentVoiceWorkflow ke dauran **callbacks** provide karti hai. Callbacks woh functions hote hain jo kisi khaas event ke hone par automatic call ho jate hain.

on_run ()

Python

```
on_run(
    workflow: SingleAgentVoiceWorkflow, transcription: str
) -> None
```

- Yeh method us waqt **call kiya jata hai jab workflow run hota hai**.
- **Inputs:**
  - workflow: SingleAgentVoiceWorkflow: Woh workflow instance jo chal raha hai.
  - transcription: str: Maujooda audio transcription.
- **Kaam:** Aap is callback ko istemal kar sakte hain workflow ke run hone par koi custom action lene ke liye, masalan logging, monitoring, ya kisi external system ko inform karna.

SingleAgentVoiceWorkflow ()

SingleAgentVoiceWorkflow ek **simple voice workflow** hai jo sirf **ek single agent ko chalata hai**. Yeh VoiceWorkflowBase ko inherit karta hai.

- Har transcription aur uske natije ko input history mein add kiya jata hai.
- **Kab istemal karen:** Agar aapka workflow simple hai aur ismein sirf ek shuruati agent hai aur koi custom logic ki zaroorat nahi hai (masalan, custom message history ya custom configs), to aap SingleAgentVoiceWorkflow ko directly istemal kar sakte hain.
- **Complex workflows ke liye:** Agar aapko zyada complex workflows (masalan, Runner ke kayi calls, custom message history, custom logic, custom configs) ki zaroorat ho, to aap VoiceWorkflowBase ko subclass karen aur apni logic implement karen.

`__init__` (Constructor)

Python

```
__init__(
    agent: Agent[Any],
    callbacks: SingleAgentWorkflowCallbacks | None = None,
)
```

- **`agent`** (Zaroori):
  - **Type:** `Agent[Any]`
  - **Tafseel:** Woh **agent** jise is workflow ke andar chalana hai.
- **`callbacks`** (Optional):
  - **Type:** `SingleAgentWorkflowCallbacks` | `None`
  - **Tafseel:** Workflow ke dauran call karne ke liye optional callbacks.

Asal Nuqta: Voice workflows woh code hain jo user ki awaaz ko text mein badalne ke baad process karte hain, agent ko chalate hain, aur phir jawab ko dobara awaaz mein tabdeel karne ke liye text paida karte hain. `SingleAgentVoiceWorkflow` simple use cases ke liye ready-to-use solution provide karta hai, jabke `VoiceWorkflowBase` complex aur customizable logic ke liye flexibility deta hai.

Input

Voice Pipeline ke Liye Audio Input (وائس پائپ لائن کے لیے آڈیو اِن پٹ)

`VoicePipeline` ko audio input do mukhtalif tareeqon se di ja sakti hai: **static audio** (jo pehle se available ho) aur **streaming audio** (jo real-time mein aa raha ho). Yeh classes in dono qism ke inputs ko handle karti hain.

`AudioInput` () - Static Audio ke Liye

`AudioInput` ek **dataclass** hai jo **static audio** ko represent karta hai, yani woh audio jo mukammal taur par ek hi baar mein available ho aur use `VoicePipeline` ke input ke taur par istemal kiya ja sake.

Attributes ():

- **`buffer`**: `NDArray[int16 | float32]`
  - Yeh woh **numpy array** hai jismein asal audio data mojood hota hai. Is mein `int16` ya `float32` type ke integers ho sakte hain, jo audio samples ko represent karte hain.
- **`frame_rate`**: `int = DEFAULT_SAMPLE_RATE` (Default: `24000`)
  - Audio data ki **sample rate** (samples per second). Zyada sample rate behtar audio quality deti hai lekin data size bhi barha deti hai. Default 24000 Hz hai.
- **`sample_width`**: `int = 2` (Default: `2`)
  - Har audio sample ki **width (bytes mein)**. Default 2 bytes (16-bit audio) hai.
- **`channels`**: `int = 1` (Default: `1`)
  - Audio data mein **channels ki tadaad**. Default 1 (mono audio) hai. Stereo audio ke liye 2 channels honge.

Methods ():

- **`to_audio_file() -> tuple[str, BytesIO, str]`**
  - Yeh method audio data ko aik **tuple** ki shakal mein wapas karta hai jismein `(filename, bytes, content_type)` shamil hote hain. Yeh us waqt mufeed hai jab aap audio ko file format mein export karna chahte hain.
- **`to_base64() -> str`**
  - Yeh method audio data ko **base64 encoded string** ke taur par wapas karta hai. Base64 encoding data ko text format mein represent karne ka aik tareeqa hai, jo use APIs ya network par asani se transfer karne mein madad karta hai.

`StreamedAudioInput` () - Streaming Audio ke Liye

`StreamedAudioInput` audio input ko **audio data ke stream** ke taur par represent karta hai. Yeh us waqt istemal hota hai jab audio data real-time mein mukhtalif hisson (chunks) mein aa raha ho, masalan microphone se. Aap is object ko `VoicePipeline` ko pass kar sakte hain aur phir `add_audio` method ka istemal karte hue queue mein mazeed audio data push kar sakte hain.

Method ():

- **`add_audio(audio: NDArray[int16 | float32])` (Async)**
  - Yeh `async` method stream mein mazeed audio data **add karta hai**.
  - **audio**: `NDArray[int16 | float32]`
    - Woh audio data jo add karna hai. Isay bhi numpy array of `int16` ya `float32` hona chahiye.
  - **Kaam:** Jab user bol raha ho, to microphone se aane wale audio chunks ko is method ke zariye `StreamedAudioInput` object mein shamil kiya jata hai. Pipeline phir is stream ko process karti rehti hai.

Khulasa (Summary)

- **`AudioInput`** un scenarios ke liye hai jahan aapke paas **mukammal audio file** pehle se mojood hai.
- **`StreamedAudioInput`** □□ scenarios ke liye hai jahan audio data **real-time mein, chote-chote hisson mein** (stream) aa raha ho, jaisa ke interactive voice applications mein hota hai.

Yeh dono classes `VoicePipeline` ko flexible input options provide karti hain, jo mukhtalif voice agent use cases ko support karti hain.

Result

StreamedAudioResult ()

StreamedAudioResult VoicePipeline se hasil hone wala **output** hai. Yeh audio data aur related events ko jese jese woh generate hote hain, **stream** karta hai. Iska matlab hai ke aapko poore jawab ke liye intezar nahi karna parta; awaaz aur us se mutaliq maloomat (events) foran milna shuru ho jati hain, jesa ke real-time phone calls ya live assistants mein hota hai.

__init__ (Constructor)

StreamedAudioResult ka constructor iske kaam karne ke liye zaroori components ko setup karta hai.

Python

```
__init__(
    tts_model: TTSModel,
    tts_settings: TTSModelSettings,
    voice_pipeline_config: VoicePipelineConfig,
)
```

- **tts_model** (Zaroori):
  - **Type:** TTSModel
  - **Tafseel:** Yeh woh **Text-to-Speech (TTS) model** hai jo text ko awaaz mein badalne ke liye istemal hoga. Yehi model actual audio generate karega.
- **tts_settings** (Zaroori):
  - **Type:** TTSModelSettings
  - **Tafseel:** TTS model ke liye khaas **settings** (tarteebat) ko represent karta hai. Is mein awaaz ki pitch, speed, ya mukhtalif awaazon (voices) ke intekhab jaisi cheezein shamil ho sakti hain. Yeh TTS output ko customize karne mein madad karta hai.
- **voice_pipeline_config** (Zaroori):
  - **Type:** VoicePipelineConfig
  - **Tafseel:** Poori voice pipeline ki **configuration** (tarteebat) ko refer karta hai. Is mein mukhtalif stage ki settings ya global parameters shamil ho sakte hain jo pipeline ke overall behaviour ko mutasir karte hain.

---

stream () - Async Method

Python

```
async stream() -> AsyncIterator[VoiceStreamEvent]
```

Yeh `async` method **events aur audio data ko stream karta hai jese jese woh generate hote hain.**

- **Returns:** `AsyncIterator[VoiceStreamEvent]`
    - Yeh aik **asynchronous iterator** return karta hai, jis se aap `for` loop ya `async for` loop istemal karte hue **`VoiceStreamEvent`** objects ko receive kar sakte hain.
    - **`VoiceStreamEvent`** mukhtalif qism ke events ko represent karta hai, masalan:
        - **Audio Chunks:** Woh asal audio data jo play kiya ja sakta hai.
        - **Text Events:** Woh text jo TTS model ne awaaz mein tabdeel kiya hai (useful for displaying subtitles or processing further).
        - **Other Metadata:** Mazeed tafseelat jo stream ke dauran paida ho sakti hain.
- **Kaam:** Jab aap `StreamedAudioResult` par `stream()` method call karte hain, to yeh data bhejna shuru kar deta hai. Client (jo is method ko call kar raha hai) phir in events ko receive karta rehta hai jese jese woh available hote hain, aur unhein real-time mein play ya process kar sakta hai.

Khulasa (Summary)

`StreamedAudioResult` `VoicePipeline` ke liye **real-time output mechanism** hai. Yeh sirf final audio file dene ke bajaye, awaaz aur us se mutaliq events ko step-by-step stream karta hai. Yeh approach interactive applications ke liye behtar hai jahan lateness (der) kam se kam honi chahiye, jaisa ke voice assistants ya telecommunication systems mein.

Pipeline Config

VoicePipelineConfig ()

`VoicePipelineConfig` ek **dataclass** hai jo `VoicePipeline` ke liye mukhtalif tarteebat (configurations) ko define karta hai. Yeh aapko pipeline ke andar ke mukhtalif components aur unke rawaiye (behavior) ko control karne ki sahoolat deta hai, khaas taur par model selection aur tracing ke hawale se.

Configuration Options ()

- **`model_provider`**: `VoiceModelProvider = field(default_factory=OpenAIVoiceModelProvider)`
    - Yeh batata hai ke pipeline ke liye kaun sa **voice model provider** istemal hoga. Default ke taur par **OpenAI** use hota hai. Agar aap kisi aur provide se models lena chahte hain, to yahan apni custom `VoiceModelProvider` class de sakte hain.
- **`tracing_disabled`**: `bool = False`
    - Agar `True` set kiya jaye to pipeline ki **tracing disable** ho jayegi. Default `False` hai, yani tracing enable rehti hai.
- **`trace_include_sensitive_data`**: `bool = True`
    - Agar `True` ho to traces mein **sensitive data** shamil kiya jayega. Default `True` hai.
    - **Ahem Note:** Yeh setting khaas taur par voice pipeline ke liye hai, aapke `Workflow` ke andar hone wali tracing par iska asar nahi hoga. Aapko apne workflow mein sensitive data handling ka khayal khud rakhna hoga.
- **`trace_include_sensitive_audio_data`**: `bool = True`

  - Agar `True` ho to traces mein **audio data** shamil kiya jayega. Default `True` hai. Audio data sensitive ho sakta hai, lehaza production mein is option ko disable karna behtar ho sakta hai.
- **`workflow_name`**: `str = 'Voice Agent'`
  - Tracing ke liye istemal hone wale **workflow ka naam**. Default naam `'Voice Agent'` hai.
- **`group_id`**: `str = field(default_factory=gen_group_id)`
  - Tracing ke liye ek **grouping identifier**. Yeh ek hi conversation ya process se mutalliq mukhtalif traces ko jorne ke liye istemal hota hai. Agar aap provide nahi karte, to system khud ek random group ID generate karega (`gen_group_id()` function ke zariye).
- **`trace_metadata`**: `dict[str, Any] | None = None`
  - Trace ke saath shamil karne ke liye **additional metadata** ki ek optional dictionary. Aap yahan koi bhi user-defined information shamil kar sakte hain.
- **`stt_settings`**: `STTModelSettings = field(default_factory=STTModelSettings)`
  - **Speech-to-Text (STT) model** ke liye istemal hone wali settings. Ismein STT model ki khaas tarteebat shamil hongi, masalan language ya model version.
- **`tts_settings`**: `TTSModelSettings = field(default_factory=TTSModelSettings)`
  - **Text-to-Speech (TTS) model** ke liye istemal hone wali settings. Ismein TTS model ki khaas tarteebat shamil hongi, masalan awaaz (voice) ka intekhab, speed, ya pitch.

Khulasa (Summary)

`VoicePipelineConfig` aapko `VoicePipeline` ke andar models, tracing behavior, aur digar parameters ko bari asani se customize karne ki ijazat deta hai. Yeh aapke voice agent ko mukhtalif scenarios aur requirements ke mutabiq ڈھالنے mein madad karta hai, khaas taur par sensitive data aur performance tracing ko manage karne mein.

Events

VoiceStreamEvent ()

`VoiceStreamEvent` `VoicePipeline` se aane wale **events ka majmooa (union)** hai, jo `StreamedAudioResult.stream()` ke zariye stream kiye jate hain. Jaisa ke humne pehle dekha, `VoicePipeline` real-time mein data output karta hai. Yeh `VoiceStreamEvent` aapko batate hain ke us stream mein kis qism ka data ya information aa rahi hai.

Yeh teen mukhtalif types ke events ho sakte hain:

1. `VoiceStreamEventAudio` (ایونٹ آڈیو)
2. `VoiceStreamEventLifecycle` (ایونٹ سائیکل لائف)
3. `VoiceStreamEventError` (ایونٹ ایرر)

`VoiceStreamEventAudio` ()

Yeh `VoicePipeline` se aane wala **streaming event** hai jo **asal audio data** ko carry karta hai.

- **data**: `NDArray[int16 | float32] | None`
    - Yeh **audio data** hai, jo ek numpy array ki shakal mein hota hai, jismein `int16` ya `float32` type ke samples ho sakte hain. Jab aap is event ko receive karte hain, to aap is `data` ko play kar sakte hain.
- **type**: `Literal["voice_stream_event_audio"] = "voice_stream_event_audio"`
    - Event ki type ko identify karta hai, jo "voice_stream_event_audio" hogi.

`VoiceStreamEventLifecycle` ()

Yeh `VoicePipeline` se aane wala **streaming event** hai jo pipeline ke **mukhtalif stages ya life cycle events** ke baray mein maloomat deta hai. Yeh aapko pipeline ke andar hone wali aham tabdeeliyon ya marhalon se aagah karta hai.

- **event**: `Literal["turn_started", "turn_ended", "session_ended"]`
    - Yeh woh event batata hai jo hua hai. Iski teen possible values hain:
        - `"turn_started"`: Jab user ki baat ka naya "turn" shuru hota hai.
        - `"turn_ended"`: Jab user ki baat ka "turn" khatam hota hai, aur agent jawab dena shuru kar sakta hai.
        - `"session_ended"`: Jab poora voice session khatam ho jata hai.
- **type**: `Literal["voice_stream_event_lifecycle"] = "voice_stream_event_lifecycle"`
    - Event ki type ko identify karta hai, jo "voice_stream_event_lifecycle" hogi.

`VoiceStreamEventError` ()

Yeh `VoicePipeline` se aane wala **streaming event** hai jo **pipeline ke dauran hone wale errors** ko inform karta hai.

- **error**: `Exception`
    - Yeh woh **error (exception)** hai jo pipeline mein hua hai. Isse aapko error ki wajah aur details ka pata chalta hai, jo debugging ke liye mufeed hai.
- **type**: `Literal["voice_stream_event_error"] = "voice_stream_event_error"`
    - Event ki type ko identify karta hai, jo "voice_stream_event_error" hogi.

Khulasa (Summary)

`VoiceStreamEvent` ek umbrella type hai jo `VoicePipeline` se aane wale tamam mukhtalif qism ke outputs ko cover karti hai. Chahe woh asal audio data ho, pipeline ke status ke bare mein updates hon, ya koi error ho, yeh events aapko real-time mein voice interaction ko manage aur monitor karne ki sahoolat dete hain. Aap in events ko `StreamedAudioResult.stream()` method se receive kar sakte hain aur har event ki `type` property check karke usay process kar sakte hain.

Exceptions

STTWebsocketConnectionError ()

`STTWebsocketConnectionError` (Speech-to-Text Websocket Connection Error) ek khaas tarah ka **exception** (error) hai jo Agents SDK mein us waqt paida hota hai jab **STT (Speech-to-Text) websocket connection mein koi masla aata hai ya woh fail ho jata hai.**

Iska Matlab Kya Hai?

Jab aap `VoicePipeline` istemal karte hain, to audio input ko text mein badalne ke liye aksar ek STT service se websocket connection banaya jata hai. Websocket connection real-time, do-tarfah communication ke liye istemal hota hai.

`STTWebsocketConnectionError` is baat ki nishandahi karta hai ke:

- **Connection Nahi Ban Saka:** Pipeline STT service se connection qaim nahi kar saki.
- **Connection Toot Gaya:** Connection banne ke baad, woh kisi wajah se toot gaya (disconnect ho gaya).
- **Data Transfer Problem:** Websocket par data bhejne ya receive karne mein koi masla hua.

Yeh Kyun Hota Hai?

Is error ki kayi wajah ho sakti hain, masalan:

- **Network Issues:** Internet connection ka na hona ya unstable hona.
- **STT Service Downtime:** Jis STT service ko aap istemal kar rahe hain, woh band ho ya usmein koi technical masla ho.
- **Incorrect Configuration:** STT service ke endpoint ya credentials mein ghalati (jaisa ke API key ka ghalat hona).
- **Firewall/Proxy Issues:** Network ki pabandiyan jo websocket connections ko block kar rahi hon.

Khulasa

`STTWebsocketConnectionError` ek **specific error type** hai jo `VoicePipeline` ke STT hisse mein connection se mutalliq masail ko zahir karta hai. Jab aapko yeh exception milta hai, to aap samajh sakte hain ke masla audio transcription service se rabte mein hai. Isse aapko debugging karne aur masle ko hal karne mein madad milti hai, kyunki aap jante hain ke kis area par tawajjah deni hai.

Model

Voice Models aur Unki Settings ()

Agents SDK ka voice module speech-to-text (STT) aur text-to-speech (TTS) functionalities ke liye mukhtalif models aur unki settings ko define karta hai. Yeh classes is baat ko yakeeni banati hain ke audio aur text ki tabdeeli aik organized aur configurable tareeqay se ho.

TTS (Text-to-Speech) Models

TTS models woh hain jo likhe hue text ko boli jaane wali awaaz mein badalte hain.

`TTSVoice` ()

Python

```python
TTSVoice = Literal["alloy", "ash", "coral", "echo", "fable", "onyx", "nova", "sage", "shimmer"]
```

- Yeh ek **type alias** hai jo `TTSModelSettings` ke `voice` enum (enumerated type) ke liye exportable types ko define karta hai.
- **Maqsad:** Yeh specific, pre-defined voices ki list provide karta hai jo TTS model generate kar sakta hai. Aap inmein se koi aik voice chun sakte hain taake aapke agent ki awaaz ko customize kiya ja sake.

`TTSModelSettings` () - dataclass

Yeh TTS model ko chalane ke liye mukhtalif settings ko specify karta hai.

- **`voice`**: `TTSVoice | None = None`
  - TTS model ke liye istemal hone wali **awaaz**. Agar provide nahi ki jaye, to mutaliqa model ki default voice istemal hogi.
- **`buffer_size`**: `int = 120`
  - Audio data ke chunks ka **kam se kam size** jo stream kiya ja raha hai. Yeh streaming efficiency ko control karta hai.
- **`dtype`**: `DTypeLike = int16`
  - Wapas kiye jaane wale audio data ki **data type**. Default 16-bit integer (int16) hai.
- **`transform_data`**: `Callable[[NDArray[int16 | float32]], NDArray[int16 | float32]] | None = None`
  - TTS model se hasil shuda data ko **transform karne ke liye ek function**. Yeh us waqt mufeed hai jab aap chahte hain ke anjami audio stream mein data kisi khaas shape mein ho.
- **`instructions`**: `str = "You will receive partial sentences. Do not complete the sentence just read out the text."`
  - TTS model ke liye **instructions**. Yeh audio output ke tone ya bolne ke tareeqay ko control karne ke liye istemal hota hai. Misal ke taur par, is default instruction ka maqsad hai ke model mukammal jumle ka intezar na kare aur partial sentences ko bhi read out kare.
- **`text_splitter`**: `Callable[[str], tuple[str, str]] = get_sentence_based_splitter()`

    - Text ko **chunks mein split karne ke liye ek function**. Yeh us waqt mufeed hai jab aap poore text ke process hone ka intezar karne ke bajaye, TTS model ko bhejne se pehle text ko chunks mein split karna chahte hain. Is se latency (der) kam hoti hai.
- **`speed`**: `float | None = None`
    - TTS model jis **raftar** (speed) se text ko read karega. Iski range 0.25 se 4.0 tak hai.

`TTSModel` () - ABC

Yeh ek **abstract base class** hai jo aik text-to-speech model ke liye interface define karti hai.

- **`model_name`**: `abstractmethod property -> str`
    - TTS model ka **naam**.
- **`run(text: str, settings: TTSModelSettings) -> AsyncIterator[bytes]`**: `abstractmethod`
    - Diye gaye text string se **audio bytes ka stream** paida karta hai, jo PCM format mein hota hai.
    - **Input:** `text` (text jo audio mein convert karna hai), `settings` (TTSModelSettings).
    - **Output:** `AsyncIterator[bytes]` (PCM format mein audio bytes ka asynchronous iterator).

STT (Speech-to-Text) Models

STT models woh hain jo boli jaane wali awaaz ko likhe hue text mein badalte hain.

`STTModelSettings` () - dataclass

Yeh speech-to-text model ke liye settings ko specify karta hai.

- **`prompt`**: `str | None = None`
    - Model ko follow karne ke liye **instructions**. Yeh transcription ki quality ko behtar banane ke liye context ya guide provide karta hai.
- **`language`**: `str | None = None`
    - Audio input ki **language**. Model ko sahi language mein recognize karne mein madad karta hai.
- **`temperature`**: `float | None = None`
    - Model ka **temperature**. Yeh transcription ki randomness ko control karta hai. Higher temperature more creative/less certain results de sakta hai.
- **`turn_detection`**: `dict[str, Any] | None = None`
    - **Turn detection settings** jo streamed audio input istemal karte waqt model ke liye hain. Yeh batata hai ke system kab samjhe ke user ki baat ka 'turn' khatam ho gaya hai.

`STTModel` () - ABC

Yeh ek **abstract base class** hai jo aik speech-to-text model ke liye interface define karti hai.

- **`model_name`**: `abstractmethod property -> str`
    - STT model ka **naam**.
- **`transcribe(input: AudioInput, settings: STTModelSettings, trace_include_sensitive_data: bool, trace_include_sensitive_audio_data: bool) -> str`**: `abstractmethod async`
    - Diye gaye audio input se **text transcription** paida karta hai.
    - **Inputs:** `input` (AudioInput), `settings` (STTModelSettings), `trace_include_sensitive_data`, `trace_include_sensitive_audio_data`.
    - **Output:** `str` (audio input ki text transcription).

- **`create_session(input: StreamedAudioInput, settings: STTModelSettings, trace_include_sensitive_data: bool, trace_include_sensitive_audio_data: bool) -> StreamedTranscriptionSession`**: `abstractmethod async`
  - Aik naya **transcription session** banata hai, jismein aap audio push kar sakte hain aur text transcriptions ka stream receive kar sakte hain. Yeh real-time transcription ke liye hai.

`StreamedTranscriptionSession` () - ABC

Yeh audio input ki **streamed transcription** ko represent karta hai.

- **`transcribe_turns() -> AsyncIterator[str]`**: `abstractmethod`
  - Text transcriptions ka stream yield karta hai. Har transcription conversation mein aik turn hota hai. Yeh method `close()` call hone ke baad hi return hone ki umeed rakhi jati hai.
- **`close() -> None`**: `abstractmethod async`
  - Session ko **band karta hai**.

`VoiceModelProvider` () - ABC

Yeh **voice model provider ke liye base interface** hai.

- **Maqsad:** Aik model provider ka kaam hai ke woh diye gaye naam se speech-to-text aur text-to-speech models create kare.
- **`get_stt_model(model_name: str | None) -> STTModel`**: `abstractmethod`
  - Naam se speech-to-text model hasil karta hai.
- **`get_tts_model(model_name: str | None) -> TTSModel`**: `abstractmethod`
  - Naam se text-to-speech model hasil karta hai.

Khulasa: Yeh tamam classes aur interfaces mil kar ek robust aur flexible system banate hain jo voice agents ko bana kar chalane mein madad karta hai. Yeh aapko mukhtalif models, unki settings, aur real-time data streaming par mukammal control dete hain.

Utils

. get_sentence_based_splitter ()

`get_sentence_based_splitter` ek utility function hai jo aapko ek aisa **function return karta hai jo text ko jumloun (sentences) ki bunyad par chote-chote hisson (chunks) mein taqseem karta hai.** Iska bunyadi maqsad yeh hai ke Text-to-Speech (TTS) models ko poora text aik sath dene ke bajaye, use chhote, meaningful (matlab wale) chunks mein diya jaye. Is se TTS ki latency (der) kam hoti hai aur output zyada natural lagta hai kyun ke model ko poora jumla complete karne ka signal mil jata hai.

Function Signature ()

Python

```
get_sentence_based_splitter(
    min_sentence_length: int = 20,
) -> Callable[[str], tuple[str, str]]
```

Parameters ()

- **`min_sentence_length`**: `int = 20`
  - **Description:** Yeh **kam se kam jumle ki lambai** hai jo aik chunk mein shamil ki jayegi. Yaani, splitter chhote jumlon ko tab tak jama karta rahega jab tak unki kul lambai is `min_sentence_length` tak na pahunch jaye.
  - **Default:** `20` characters.
  - **Maqsad:** Is se yeh yakeeni banaya jata hai ke TTS model ko itna text zaroor mile ke woh aik reasonable (munasib) awaaz ka hissa bana sake, aur bohat zyada chhote ya fragmented (tukron mein bante hue) audio clips na banain.

Returns ()

- **Type:** `Callable[[str], tuple[str, str]]`
  - **Description:** Yeh function khud ek aur function wapas karta hai. Yeh wapas kiya gaya function woh hai jo asal splitting ka kaam karta hai.
  - **Wapas kiya gaya function ka signature:** `(text: str) -> tuple[str, str]`
    - Yeh function **ek `str` (poora text)** input ke taur par leta hai.
    - Aur **do `str` (strings) ka aik `tuple`** wapas karta hai:
      - Pehla string: Woh **chunk jo TTS model ko bheja jayega** (yani woh hissa jo bola jayega).
      - Doosra string: Woh **baqi bacha hua text** jise agle process ke liye rakha jayega.

Kaam Karne Ka Tareeqa (How It Works)

`get_sentence_based_splitter` function jab call hota hai to woh ek `splitter` function banata hai aur use return karta hai. Jab aap us `splitter` function ko apne bade text par istemal karte hain:

1. Woh **text ko jumloun mein taqseem** karta hai.
2. Woh in **jumlon ko jama karta hai** (yani jorta hai), is baat ka khayal rakhte hue ke har chunk ki lambai `min_sentence_length` se kam na ho (agar mumkin ho).
3. Jab use aik aisa point mil jata hai jahan par woh aik **mukammal jumla ya jumlon ka majmooa** bana sakta hai jo minimum length ki shart ko poora karta ho, to woh us hisse ko **"chunk to speak"** ke taur par wapas karta hai.
4. **Baaki bache hue text** ko woh doosre hisse ke taur par wapas karta hai, taake agle chunking process ke liye istemal kiya ja sake.

Istemal (Usage)

`get_sentence_based_splitter` ka aam istemal `TTSModelSettings` ke `text_splitter` parameter mein hota hai, jaisa ke aap ne pehle dekha tha. Yeh TTS model ko fori audio generation shuru karne mein madad karta hai, bajaye iske ke woh poore jawab ka intezar kare.

Misal:

Python

```
from agents.voice.utils import get_sentence_based_splitter

# Get a splitter function with default min_sentence_length (20)
my_splitter = get_sentence_based_splitter()

long_text = "Assalam-o-Alaikum. Aaj mausam bohat khushgawar hai. Kya aap ko bahar jana pasand
hai? Ya ghar par hi rehna chahenge?"

# Pehla chunk
chunk1, remaining_text1 = my_splitter(long_text)
print(f"Chunk 1: '{chunk1}'")
print(f"Remaining 1: '{remaining_text1}'")
# Output may be:
# Chunk 1: 'Assalam-o-Alaikum. Aaj mausam bohat khushgawar hai.'
# Remaining 1: 'Kya aap ko bahar jana pasand hai? Ya ghar par hi rehna chahenge?'

# Doosra chunk
chunk2, remaining_text2 = my_splitter(remaining_text1)
print(f"Chunk 2: '{chunk2}'")
print(f"Remaining 2: '{remaining_text2}'")
# Output may be:
# Chunk 2: 'Kya aap ko bahar jana pasand hai? Ya ghar par hi rehna chahenge?'
# Remaining 2: ''
```

Khulasa (Summary)

`get_sentence_based_splitter` ek aham tool hai jo voice applications mein **real-time responsiveness** ko behtar banata hai. Yeh lambe text ko chhote, bolne ke qabil hisson mein taqseem karke Text-to-Speech system ko foran audio generate karna shuru karne ki ijazat deta hai, jis se user ka tajurba (experience) zyada seamless (be-rukawat) aur natural hota hai.

OpenAIVoiceModelProvider

OpenAIVoiceModelProvider ()

`OpenAIVoiceModelProvider` ek **`VoiceModelProvider`** ki concrete implementation hai jo **OpenAI ke models** ko Speech-to-Text (STT) aur Text-to-Speech (TTS) functionalities ke liye istemal karta hai. Yeh aapke voice pipeline ko OpenAI ki powerful AI capabilities se jorta hai.

`__init__` (Constructor)

`OpenAIVoiceModelProvider` ka constructor aapko OpenAI API ke sath rabta qaim karne ke liye zaroori credentials aur client settings define karne ki ijazat deta hai.

Python

```python
__init__(
    *,
    api_key: str | None = None,
    base_url: str | None = None,
    openai_client: AsyncOpenAI | None = None,
    organization: str | None = None,
    project: str | None = None,
) -> None
```

- **`api_key`** (Optional):
  - **Type:** `str | None`
  - **Tafseel: OpenAI client ke liye API key.** Agar aap yeh provide nahi karte, to system default API key (aam taur par environment variables se) istemal karega.
- **`base_url`** (Optional):
  - **Type:** `str | None`
  - **Tafseel: OpenAI client ke liye base URL.** Agar nahi diya gaya, to default base URL istemal hoga. Yeh custom endpoints ya proxy setup ke liye mufeed hai.
- **`openai_client`** (Optional):
  - **Type:** `AsyncOpenAI | None`
  - **Tafseel:** Ek **optional `AsyncOpenAI` client** instance jo istemal kiya jayega. Agar aap yeh provide nahi karte, to provider khud `api_key` aur `base_url` ka istemal karte hue ek naya `AsyncOpenAI` client banayega. Yeh flexibility deta hai agar aapke paas pehle se configured client hai.
- **`organization`** (Optional):
  - **Type:** `str | None`
  - **Tafseel: OpenAI client ke liye organization ID.** Yeh OpenAI ke users ke liye apni requests ko mukhtalif organizations ke tehat group karne ke liye hota hai.
- **`project`** (Optional):
  - **Type:** `str | None`
  - **Tafseel: OpenAI client ke liye project ID.** Agar aapke paas OpenAI mein projects set hain, to aap yahan specific project ID specify kar sakte hain.

`get_stt_model` ()

Python

```
get_stt_model(model_name: str | None) -> STTModel
```

- **Kaam:** Yeh method diye gaye **naam se Speech-to-Text (STT) model ko hasil karta hai.**
- **Parameters:**
  - **`model_name`** (Optional): Woh model ka naam jise aap hasil karna chahte hain. Agar `None` diya gaya, to yeh default STT model (OpenAI ke Whisper model jaisa koi) return karega.
- **Returns:**
  - **Type:** `STTModel`
  - **Tafseel:** Requested STT model ka instance.

`get_tts_model` ()

Python

```
get_tts_model(model_name: str | None) -> TTSModel
```

- **Kaam:** Yeh method diye gaye **naam se Text-to-Speech (TTS) model ko hasil karta hai.**
- **Parameters:**
  - **`model_name`** (Optional): Woh model ka naam jise aap hasil karna chahte hain. Agar `None` diya gaya, to yeh default TTS model (OpenAI ke TTS models jaisa koi) return karega.
- **Returns:**
  - **Type:** `TTSModel`
  - **Tafseel:** Requested TTS model ka instance.

Khulasa (Summary)

`OpenAIVoiceModelProvider` ek zaroori component hai jo aapke voice agent ko **OpenAI ke state-of-the-art speech models** se jorta hai. Yeh aapko asani se STT aur TTS models ko configure aur access karne ki sahoolat deta hai, jo audio input ko process karne aur human-like audio responses generate karne ke liye zaroori hain. Is ki flexibility aapko custom API keys, URLs, aur client instances istemal karne ki ijazat deti hai, jo mukhtalif deployment scenarios ke liye mufeed hai.

OpenAI STT

OpenAI Speech-to-Text (STT) Models ()

Yeh classes OpenAI ke Speech-to-Text (STT) models ko use karne ke liye hain, jo aapki audio input ko text mein badalne ka kaam karte hain. Yeh khas taur par **real-time transcription** aur **static audio transcription** dono ke liye design kiye gaye hain.

`OpenAISTTTranscriptionSession` ()

`OpenAISTTTranscriptionSession` ek **`StreamedTranscriptionSession`** ki concrete implementation hai. Yeh OpenAI ke STT model ke liye aik transcription session ko represent karta hai, jismein aap audio data ko stream kar sakte hain aur real-time mein text transcriptions receive kar sakte hain.

- Yeh class un scenarios ke liye zaroori hai jahan aapko continuous audio input ko process karna ho, jaise ke live conversations ya microphone se aane wali awaaz.
- Ismein aise methods honge jo audio chunks ko service par bhejte hain aur phir transcription ke turns ko `AsyncIterator` ke zariye wapas karte hain.

`OpenAISTTModel` ()

`OpenAISTTModel` ek **`STTModel`** ki concrete implementation hai jo khas taur par OpenAI ke Speech-to-Text models (jaise ke Whisper) ke saath kaam karne ke liye banaya gaya hai.

`__init__` (Constructor)

Python

```
__init__(model: str, openai_client: AsyncOpenAI)
```

- **`model`**: `str` (Zaroori)
  - Yeh **OpenAI STT model ka naam** hai jise aap istemal karna chahte hain (masalan, `"whisper-1"`).
- **`openai_client`**: `AsyncOpenAI` (Zaroori)
  - Yeh **OpenAI ka asynchronous client** hai jise model API calls karne ke liye istemal karega. Ise pehle se configure kiya hona chahiye (API key, base URL, waghera ke saath).

`transcribe` () - Async Method

Python

```python
async transcribe(
    input: AudioInput,
    settings: STTModelSettings,
    trace_include_sensitive_data: bool,
    trace_include_sensitive_audio_data: bool,
) -> str
```

- **Kaam:** Yeh method **ek mukammal (static) audio input ko text mein badalta hai**. Yeh un scenarios ke liye hai jahan aapke paas poori audio file pehle se available ho.
- **Parameters:**
    - **`input`**: `AudioInput` (Zaroori)
        - Woh **audio input** jise text mein badalna hai. Yeh ek static audio buffer hoti hai.
    - **`settings`**: `STTModelSettings` (Zaroori)
        - Transcription ke liye **settings**, masalan language ya prompt.
    - **`trace_include_sensitive_data`**: `bool`
        - Kya sensitive data ko traces mein shamil kiya jaye.
    - **`trace_include_sensitive_audio_data`**: `bool`
        - Kya sensitive audio data ko traces mein shamil kiya jaye.
- **Returns:** `str`
    - Transcribed text (audio ka matni roop).

`create_session` () - Async Method

Python

```python
async create_session(
    input: StreamedAudioInput,
    settings: STTModelSettings,
    trace_include_sensitive_data: bool,
    trace_include_sensitive_audio_data: bool,
) -> StreamedTranscriptionSession
```

- **Kaam:** Yeh method **ek naya transcription session banata hai**. Yeh un scenarios ke liye hai jahan audio input stream ho rahi ho (real-time). Aap is session mein audio push kar sakte hain aur text transcriptions ka stream receive kar sakte hain.
- **Parameters:**
    - **`input`**: `StreamedAudioInput` (Zaroori)
        - Woh **audio input stream** jise text mein badalna hai.
    - **`settings`**: `STTModelSettings` (Zaroori)
        - Transcription session ke liye settings.
    - **`trace_include_sensitive_data`**: `bool`
        - Kya sensitive data ko traces mein shamil kiya jaye.
    - **`trace_include_sensitive_audio_data`**: `bool`
        - Kya sensitive audio data ko traces mein shamil kiya jaye.
- **Returns:** `StreamedTranscriptionSession`
    - Aik naya transcription session ka instance.

Khulasa (Summary)

`OpenAISTTTranscriptionSession` aur `OpenAISTTModel` mil kar **OpenAI ki STT capabilities ko Agents SDK ke voice pipeline mein integrate karte hain.**

- `OpenAISTTModel` aapko ya to **mukammal audio files** ko transcribe karne (`transcribe` method) ya **real-time audio streams** ke liye transcription sessions banane (`create_session` method) ki ijazat deta hai.
- `OpenAISTTTranscriptionSession` asli real-time transcription logic ko handle karta hai, jahan audio chunks milte hain aur text turns ke roop mein wapas aate hain.

Yeh set of classes aapke voice agent ko user ki awaaz ko behtareen tarike se samajhne aur usay text mein badalne ke qabil banata hai, chahe input static ho ya streaming.

OpenAI TTS

OpenAITTSModel ()

`OpenAITTSModel` ek **`TTSModel`** ki concrete (asal) implementation hai, jo **OpenAI ke Text-to-Speech (TTS) models** ka istemal karti hai. Iska buniyadi maqsad likhe hue text ko insani awaaz (audio) mein tabdeel karna hai, jise voice agents mein jawab dene ke liye istemal kiya ja sakta hai.

`__init__` (Constructor)

`OpenAITTSModel` ke constructor mein aap batate hain ke konsa OpenAI TTS model istemal karna hai aur kis OpenAI client ke zariye API calls ki jayengi.

Python

```
__init__(model: str, openai_client: AsyncOpenAI)
```

- **`model`**: `str` (Zaroori)
  - Yeh us **OpenAI TTS model ka naam** hai jise aap istemal karna chahte hain. Masalan, "tts-1" ya "tts-1-hd".
- **`openai_client`**: `AsyncOpenAI` (Zaroori)
  - Yeh **OpenAI ka asynchronous client** hai jise model API requests bhejne ke liye istemal karega. Is client ko pehle se sahi `api_key`, `base_url`, waghera ke saath configure kiya gaya hona chahiye.

`run` () - Async Method

Python

```
async run(
    text: str, settings: TTSModelSettings
) -> AsyncIterator[bytes]
```

Yeh `async` method asal kaam karta hai: **text ko awaaz mein badalta hai.** Yeh khaas taur par streaming ke liye design kiya gaya hai, jahan audio data chunks mein generate hota hai aur foran bheja ja sakta hai.

- **Parameters:**
  - **`text`**: `str` (Zaroori)
    - Woh **text** jise aap awaaz mein convert karna chahte hain.
  - **`settings`**: `TTSModelSettings` (Zaroori)
    - TTS model ke liye istemal hone wali **settings**. Is mein awaaz (voice), bolne ki raftaar (speed), ya text splitter jaisi tarteebat shamil ho sakti hain, jo audio output ko customize karti hain.
- **Returns:** `AsyncIterator[bytes]`
  - Yeh **audio chunks ka asynchronous iterator** wapas karta hai. Iska matlab hai ke aapko poori audio file ka intezar nahi karna parta. Jese jese TTS model audio generate karta hai, woh chote-chote `bytes` ke chunks ko stream karta rehta hai. Aap in chunks ko foran play kar sakte hain, jis se real-time responsiveness behtar hoti hai.

Khulasa (Summary)

`OpenAITTSModel` OpenAI ki muta'asir kun TTS capabilities ko Agents SDK mein integrate karta hai. Yeh aapko text input ko natural-sounding (qudrati lagne wali) audio mein badalne ki ijazat deta hai, aur `run` method ki asynchronous streaming nature ki wajah se yeh real-time voice applications ke liye behtareen hai. Aap is model ko mukhtalif `TTSModelSettings` ke zariye apni marzi ke mutabiq customize bhi kar sakte hain.

EXTENSION

Handoff filters

remove_all_tools ()

remove_all_tools aik function hai jo **HandoffInputData** object se tamam qism ke **tool-related items ko filter out (hatata) karta hai**. Jab aik agent apna control ya task kisi doosre agent ko "handoff" karta hai, to is data (HandoffInputData) mein pichle agent ke istemal shuda tools ki maloomat bhi shamil ho sakti hain. Yeh function is maloomat ko saaf karne ke liye istemal hota hai.

Function Signature ()

Python

```
remove_all_tools(
    handoff_input_data: HandoffInputData,
) -> HandoffInputData
```

Parameters ()

- **handoff_input_data**: HandoffInputData
  - **Tafseel:** Woh HandoffInputData object jise process karna hai. Is object mein pichle agent ki taraf se generate ki gayi mukhtalif qism ki input data shamil hoti hai.

Returns ()

- **Type:** HandoffInputData
  - **Tafseel:** Aik **naya HandoffInputData object** wapas kiya jata hai jismein se **tamaam tool items filter out kar diye gaye hain**. Asal handoff_input_data object mein koi tabdeeli nahi hoti.

Kaam Kya Karta Hai? (What It Does)

Yeh function khaas taur par HandoffInputData object ke andar se darj zel tool-related maloomat ko hatata hai:

- **File Search:** Agar pichle agent ne file search tools istemal kiye the.
- **Web Search:** Agar pichle agent ne web search tools istemal kiye the.
- **Function Calls + Output:** Koi bhi function calls jo pichle agent ne ki thin, aur unke outputs.

Yeh Kyun Zaroori Hai? (Why Is It Important?)

Handoff scenarios mein, kabhi-kabhi aap nahi chahte ke woh tamam tafseelat (joh pichle agent ke tools se mutalliq ho) agle agent tak pahunchein. Iski kayi wajah ho sakti hain:

- **Irrelevance (Be-talluqi):** Agle agent ke liye pichle tool calls ki tafseelat be-maqsad ho sakti hain.
- **Security/Privacy:** Kuch tool outputs mein sensitive maloomat ho sakti hain jo agle agent ko nahi dikhani chahiyen.
- **Simplification:** Handoff data ko saaf suthra aur sirf zaroori maloomat tak mehdood rakhna agle agent ke liye process karna asan banata hai.
- **Performance:** Ghair zaroori data ko hatane se processing load kam ho sakta hai.

Misal ke taur par, agar aik customer support bot ne product database mein search kiya, aur ab woh conversation ko aik human agent ko handoff kar raha hai, to aap shayad nahi chahenge ke human agent ko database search ke raw logs dikhen. Iske bajaye, sirf asal masle ki tafseel dikhana behtar hoga.

Khulasa (Summary)

`remove_all_tools` function `HandoffInputData` object mein se **file search, web search, aur function call/output ki maloomat ko filter karta hai**. Yeh data ko saaf suthra aur relevant rakhta hai jab aik agent apna control kisi doosre agent ko deta hai, security, privacy, aur processing efficiency ko behtar banate hue.

```
def remove_all_tools(handoff_input_data: HandoffInputData) -> HandoffInputData:
    """Filters out all tool items: file search, web search and function calls+output."""

    history = handoff_input_data.input_history
    new_items = handoff_input_data.new_items

    filtered_history = (
        _remove_tool_types_from_input(history) if isinstance(history, tuple) else history
    )
    filtered_pre_handoff_items = _remove_tools_from_items(handoff_input_data.pre_handoff_items)
    filtered_new_items = _remove_tools_from_items(new_items)

    return HandoffInputData(
        input_history=filtered_history,
        pre_handoff_items=filtered_pre_handoff_items,
        new_items=filtered_new_items,

    )
```

Handoff prompt

Handoff Prompts aur Unka Maqsad (Handoff Prompts and Their Purpose)

`RECOMMENDED_PROMPT_PREFIX` aur `prompt_with_handoff_instructions` Agents SDK mein ek ahem role ada karte hain, khaas taur par jab aap **multi-agent system** bana rahe hon jahan mukhtalif agents aik doosre ko baat-cheet ya task "handoff" karte hain. Inka buniyadi maqsad agents ko sahi tareeqay se behave karne ki hidayat dena hai, khaas taur par jab woh **handoffs** istemal kar rahe hon.

RECOMMENDED_PROMPT_PREFIX ()

Python

```
RECOMMENDED_PROMPT_PREFIX = "# System context\nYou are part of a multi-agent system called the Agents SDK, designed to make agent coordination and execution easy. Agents uses two primary abstraction:

**Agents** and **Handoffs**. An agent encompasses instructions and tools and can hand off a conversation to another agent when appropriate. Handoffs are achieved by calling a handoff function, generally named `transfer_to_<agent_name>`. Transfers between agents are handled seamlessly in the background; do not mention or draw attention to these transfers in your conversation with the user.\n"
```

Yeh ek **string constant** hai jo ek taraf se **system context** provide karta hai. Jab aap kisi agent ko tayar karte hain, to is prefix ko uske prompt (hidayat) ke shuru mein shamil kiya jata hai. Iska maqsad agent ko darj zel baatein samjhana hai:

- **Multi-Agent System Ka Hissa:** Agent ko pata ho ke woh "Agents SDK" naam ke multi-agent system ka hissa hai.
- **Aham Abstractions:** Agent ko maloom ho ke system "Agents" aur "Handoffs" ke do bunyadi concepts par mabni hai.
- **Agent Ka Role:** Agent ko samajh aa jaye ke uska kaam instructions aur tools ko istemal karna hai, aur agar zaroorat pade to conversation ko doosre agent ko **handoff** karna hai.
- **Handoff Ka Tareeqa:** Handoff kaise kiya jata hai (`transfer_to_<agent_name>` function ko call karke).
- **User Se Chhupana:** Sabse ahem hidayat yeh hai ke agent ko **user ke saath baat-cheet mein in handoffs ka zikr nahi karna** chahiye aur na hi unki taraf tawajjah dilani chahiye. "Transfers between agents are handled seamlessly in the background." (Agents ke darmiyan transfers pas-e-pardah be-rukawat tareeqay se hote hain.)

Maqsad: Yeh prefix agent ko "behind the scenes" (pas-e-pardah) hone wale handoffs ke bare mein batata hai, take woh user ke liye ek smooth aur unbroken (be-toot) conversation ka tajurba faraham kar sake. User ko yeh mehsoos nahi hona chahiye ke uski baat kisi aur agent ko di ja rahi hai; use lage ke woh ek hi intelligent entity se baat kar raha hai.

`prompt_with_handoff_instructions` ()

Python

```
prompt_with_handoff_instructions(prompt: str) -> str
```

Yeh ek **function** hai jo kisi bhi diye gaye **prompt** mein `RECOMMENDED_PROMPT_PREFIX` ko shamil karke usay wapas karta hai.

Parameters ()

- **`prompt`**: `str`
    - **Tafseel:** Woh asal prompt (hidayat) jo aapne apne agent ke liye likha hai.

Returns ()

- **Type:** `str`
    - **Tafseel:** Aik naya string jo `RECOMMENDED_PROMPT_PREFIX` aur aapke diye gaye `prompt` ko mila kar bana hai.

Kaam (Functionality)

Jab aap is function ko call karte hain, to yeh aapke agent ke liye final prompt banata hai:

```
# System context
You are part of a multi-agent system called the Agents SDK, designed to make agent coordination
and execution easy. Agents uses two primary abstraction: **Agents** and **Handoffs**. An agent
encompasses instructions and tools and can hand off a conversation to another agent when
appropriate. Handoffs are achieved by calling a handoff function, generally named
`transfer_to_<agent_name>`. Transfers between agents are handled seamlessly in the background;
do not mention or draw attention to these transfers in your conversation with the user.
[Aapka asal prompt yahan aaega]
```

Maqsad: Yeh function aapko har baar manually `RECOMMENDED_PROMPT_PREFIX` copy paste karne ki zaroorat se bachata hai aur yakeeni banata hai ke har woh agent jo handoffs istemal kare, usay zaruri system-level hidayat mil jayen.

Khulasa (Summary)

`RECOMMENDED_PROMPT_PREFIX` aur `prompt_with_handoff_instructions` mil kar **multi-agent systems mein user experience ko behtar banate hain**. Yeh agents ko sikhate hain ke kis tarah seamlessly (be-rukawat) tareeqay se control ko aik doosre ko handoff karna hai, aur sabse ahem, yeh handoffs **user se chhupaye rakhte hain**, jis se user ko lagta hai ke woh aik hi, mukammal aur samajhdar AI entity se baat kar raha hai.

```
def prompt_with_handoff_instructions(prompt: str) -> str:
    """
    Add recommended instructions to the prompt for agents that use handoffs.
    """

    return f"{RECOMMENDED_PROMPT_PREFIX}\n\n{prompt}"
```

LiteLLM Models

LiteLLMModel ()

`LiteLLMModel` Agents SDK ke andar aik aisi class hai jo aapko **LiteLLM library ke zariye mukhtalif AI models** ko istemal karne ki ijazat deti hai. LiteLLM aik wrapper (ghilaaf) hai jo mukhtalif Large Language Models (LLMs) providers jaise ke OpenAI, Anthropic, Google Gemini, Mistral, aur kayi doosre models ko aik hi interface ke tehat dastiyab karta hai.

Iska buniyadi maqsad: Developers ke liye mukhtalif AI model APIs ke sath interaction ko **bohat aasan banana** hai. Aapko har provider ke liye alag-alag code likhne ki zaroorat nahi parti; `LiteLLMModel` ke zariye aap sab ko aik hi tarah se handle kar sakte hain.

`LiteLLMModel` Ka Istemal Kyun? (Why Use LiteLLMModel?)

- **Universal Access:** Aap ek hi code base se mukhtalif providers (OpenAI, Anthropic, Gemini, Mistral, waghera) ke models ko access kar sakte hain.
- **Abstraction Layer:** Yeh har model provider ke complex APIs ke upar aik simple layer provide karta hai.
- **Flexibility:** Aap asani se ek model se doosre model par switch kar sakte hain, bina apni application ki core logic ko tabdeel kiye.
- **Cost Management & Fallbacks:** LiteLLM mein built-in features hain jaise ke cost tracking, retries, aur fallbacks (yani agar ek model kaam na kare to doosre par switch karna).

Supported Models ()

LiteLLM bohat se models ko support karta hai. Aap un supported models ki mukammal list is link par dekh sakte hain: litellm models.

Khulasa (Summary)

`LiteLLMModel` aik powerful integration hai jo Agents SDK ko **LLMs ki aik wasee range (wide range)** se jorta hai. Yeh aapke agents ko mukhtalif AI models ki capabilities istemal karne ki sahoolat deta hai, jis se aapki application zyada versatile aur future-proof ban jati hai. Agar aapko apne agent mein mukhtalif AI providers ke models ko seamlessly (be-rukawat) tareeqay se istemal karna hai, to `LiteLLMModel` aik behtareen intekhab hai.

Source code in src/agents/extensions/models/litellm_model.py

```
class LitellmModel(Model):
    """This class enables using any model via LiteLLM. LiteLLM allows you to acess OpenAPI,
    Anthropic, Gemini, Mistral, and many other models.
    See supported models here: [litellm models](https://docs.litellm.ai/docs/providers).
    """

    def __init__(
        self,
        model: str,
        base_url: str | None = None,
        api_key: str | None = None,
    ):
        self.model = model
        self.base_url = base_url
        self.api_key = api_key

    async def get_response(
        self,
        system_instructions: str | None,
        input: str | list[TResponseInputItem],
        model_settings: ModelSettings,
        tools: list[Tool],
        output_schema: AgentOutputSchemaBase | None,
        handoffs: list[Handoff],
        tracing: ModelTracing,
        previous_response_id: str | None,
        prompt: Any | None = None,
    ) -> ModelResponse:
        with generation_span(
            model=str(self.model),
            model_config=model_settings.to_json_dict()
            | {"base_url": str(self.base_url or ""), "model_impl": "litellm"},
            disabled=tracing.is_disabled(),
        ) as span_generation:
            response = await self._fetch_response(
                system_instructions,
                input,
                model_settings,
                tools,
                output_schema,
                handoffs,
                span_generation,
                tracing,
                stream=False,
                prompt=prompt,
            )

            assert isinstance(response.choices[0], litellm.types.utils.Choices)

            if _debug.DONT_LOG_MODEL_DATA:
                logger.debug("Received model response")
            else:
                logger.debug(
                    f"LLM resp:\n{json.dumps(response.choices[0].message.model_dump(), indent=2)}\n"
                )

            if hasattr(response, "usage"):
                response_usage = response.usage
                usage = (
                    Usage(
                        requests=1,
                        input_tokens=response_usage.prompt_tokens,
                        output_tokens=response_usage.completion_tokens,
                        total_tokens=response_usage.total_tokens,
                        input_tokens_details=InputTokensDetails(
                            cached_tokens=getattr(
                                response_usage.prompt_tokens_details, "cached_tokens", 0
                            )
                            or 0
                        ),
                        output_tokens_details=OutputTokensDetails(
                            reasoning_tokens=getattr(
                                response_usage.completion_tokens_details, "reasoning_tokens", 0
                            )
                            or 0
                        ),
                    )
                    if response.usage
                    else Usage()
                )
            else:
                usage = Usage()
                logger.warning("No usage information returned from Litellm")

            if tracing.include_data():
                span_generation.span_data.output = [response.choices[0].message.model_dump()]
            span_generation.span_data.usage = {
                "input_tokens": usage.input_tokens,
                "output_tokens": usage.output_tokens,
            }

            items = Converter.message_to_output_items(
                LitellmConverter.convert_message_to_openai(response.choices[0].message)
            )

            return ModelResponse(
                output=items,
                usage=usage,
                response_id=None,
            )

    async def stream_response(
        self,
        system_instructions: str | None,
        input: str | list[TResponseInputItem],
        model_settings: ModelSettings,
        tools: list[Tool],
        output_schema: AgentOutputSchemaBase | None,
        handoffs: list[Handoff],
        tracing: ModelTracing,
        previous_response_id: str | None,
        prompt: Any | None = None,
    ) -> AsyncIterator[TResponseStreamEvent]:
        with generation_span(
            model=str(self.model),
            model_config=model_settings.to_json_dict()
            | {"base_url": str(self.base_url or ""), "model_impl": "litellm"},
            disabled=tracing.is_disabled(),
        ) as span_generation:
            response, stream = await self._fetch_response(
                system_instructions,
                input,
                model_settings,
                tools,
                output_schema,
                handoffs,
                span_generation,
                tracing,
                stream=True,
                prompt=prompt,
            )

            final_response: Response | None = None
            async for chunk in ChatCmplStreamHandler.handle_stream(response, stream):
                yield chunk
                if chunk.type == "response.completed":
                    final_response = chunk.response

            if tracing.include_data() and final_response:
                span_generation.span_data.output = [final_response.model_dump()]

            if final_response and final_response.usage:
                span_generation.span_data.usage = {
                    "input_tokens": final_response.usage.input_tokens,
                    "output_tokens": final_response.usage.output_tokens,
                }

    @overload
    async def _fetch_response(
        self,
        system_instructions: str | None,
        input: str | list[TResponseInputItem],
        model_settings: ModelSettings,
        tools: list[Tool],
        output_schema: AgentOutputSchemaBase | None,
        handoffs: list[Handoff],
        span: Span[GenerationSpanData],
        tracing: ModelTracing,
        stream: Literal[True],
        prompt: Any | None = None,
    ) -> tuple[Response, AsyncStream[ChatCompletionChunk]]: ...

    @overload
    async def _fetch_response(
        self,
        system_instructions: str | None,
        input: str | list[TResponseInputItem],
        model_settings: ModelSettings,
        tools: list[Tool],
        output_schema: AgentOutputSchemaBase | None,
        handoffs: list[Handoff],
        span: Span[GenerationSpanData],
        tracing: ModelTracing,
        stream: Literal[False],
        prompt: Any | None = None,
    ) -> litellm.types.utils.ModelResponse: ...

    async def _fetch_response(
        self,
        system_instructions: str | None,
        input: str | list[TResponseInputItem],
        model_settings: ModelSettings,
        tools: list[Tool],
        output_schema: AgentOutputSchemaBase | None,
        handoffs: list[Handoff],
        span: Span[GenerationSpanData],
        tracing: ModelTracing,
        stream: bool = False,
        prompt: Any | None = None,
    ) -> litellm.types.utils.ModelResponse | tuple[Response, AsyncStream[ChatCompletionChunk]]:
        converted_messages = Converter.items_to_messages(input)

        if system_instructions:
            converted_messages.insert(
                0,
                {
                    "content": system_instructions,
                    "role": "system",
                },
            )
        if tracing.include_data():
            span.span_data.input = converted_messages
        parallel_tool_calls = (
            True
            if model_settings.parallel_tool_calls and tools and len(tools) > 0
            else False
            if model_settings.parallel_tool_calls is False
            else None
        )
        tool_choice = Converter.convert_tool_choice(model_settings.tool_choice)
        response_format = Converter.convert_response_format(output_schema)

        converted_tools = [Converter.tool_to_openai(tool) for tool in tools] if tools else []

        for handoff in handoffs:
            converted_tools.append(Converter.convert_handoff_tool(handoff))

        if _debug.DONT_LOG_MODEL_DATA:
            logger.debug("Calling LLM")
        else:
            logger.debug(
                f"Calling Litellm model: {self.model}\n"
                f"{json.dumps(converted_messages, indent=2)}\n"
                f"Tools:\n{json.dumps(converted_tools, indent=2)}\n"
                f"Stream: {stream}\n"
                f"Tool choice: {tool_choice}\n"
                f"Response format: {response_format}\n"
            )

        reasoning_effort = model_settings.reasoning.effort if model_settings.reasoning else None

        stream_options = None
        if stream and model_settings.include_usage is not None:
            stream_options = {"include_usage": model_settings.include_usage}

        extra_kwargs = {}
        if model_settings.extra_query:
            extra_kwargs["extra_query"] = model_settings.extra_query
        if model_settings.metadata:
            extra_kwargs["metadata"] = model_settings.metadata
        if model_settings.extra_body and isinstance(model_settings.extra_body, dict):
            extra_kwargs.update(model_settings.extra_body)

        # Add kwargs from model_settings.extra_args, filtering out None values
        if model_settings.extra_args:
            extra_kwargs.update(model_settings.extra_args)

        ret = await litellm.acompletion(
            model=self.model,
            messages=converted_messages,
            tools=converted_tools or None,
            temperature=model_settings.temperature,
            top_p=model_settings.top_p,
            frequency_penalty=model_settings.frequency_penalty,
            presence_penalty=model_settings.presence_penalty,
            max_tokens=model_settings.max_tokens,
            tool_choice=self._remove_not_given(tool_choice),
            response_format=self._remove_not_given(response_format),
            parallel_tool_calls=parallel_tool_calls,
            stream=stream,
            stream_options=stream_options,
            reasoning_effort=reasoning_effort,
            extra_headers={**HEADERS, **(model_settings.extra_headers or {})},
            api_key=self.api_key,
            base_url=self.base_url,
            **extra_kwargs,
        )

        if isinstance(ret, litellm.types.utils.ModelResponse):
            return ret
        response = Response(
            id=FAKE_RESPONSES_ID,
            created_at=time.time(),
            model=self.model,
            object="response",
            output=[],
            tool_choice=cast(Literal["auto", "required", "none"], tool_choice)
            if tool_choice != NOT_GIVEN
            else "auto",
            top_p=model_settings.top_p,
            temperature=model_settings.temperature,
            tools=[],
            parallel_tool_calls=parallel_tool_calls or False,
            reasoning=model_settings.reasoning,
        )
        return response, ret

    def _remove_not_given(self, value: Any) -> Any:
        if isinstance(value, NotGiven):
            return None
        return value
```
