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

from pydantic import BaseModel, Field

from swarms.structs.agent import Agent
from swarms.telemetry.capture_sys_data import log_agent_data
from swarms.schemas.agent_step_schemas import ManySteps
from swarms.prompts.ag_prompt import aggregator_system_prompt
from swarms.utils.loguru_logger import initialize_logger

logger = initialize_logger(log_folder="mixture_of_agents")

time_stamp = time.strftime("%Y-%m-%d %H:%M:%S")


class MixtureOfAgentsInput(BaseModel):
    name: str = "MixtureOfAgents"
    description: str = (
        "A class to run a mixture of agents and aggregate their responses."
    )
    agents: List[Dict[str, Any]]
    aggregator_agent: Any = Field(
        ...,
        description="An aggregator agent to be used in the mixture.",
```

```
    )
    aggregator_system_prompt: str = Field(
        default=aggregator_system_prompt.get_prompt(),
        description=aggregator_system_prompt.description,
    )
    layers: int = 3
    time_created: str = Field(
        time_stamp,
        description="The time the mixture of agents was created.",
    )


class MixtureOfAgentsOutput(BaseModel):
    id: str = Field(
        ..., description="The ID of the mixture of agents."
    )
    task: str = Field(..., description="None")
    InputConfig: MixtureOfAgentsInput
    # output: List[ManySteps]
    normal_agent_outputs: List[ManySteps]
    aggregator_agent_summary: str
    time_completed: str = Field(
        time_stamp,
        description="The time the mixture of agents was completed.",
    )
```

```
class MixtureOfAgents:
    """
    A class to manage and run a mixture of agents, aggregating their responses.
    """

    def __init__(
        self,
        name: str = "MixtureOfAgents",
        description: str = "A class to run a mixture of agents and aggregate their responses.",
        agents: List[Agent] = [],
        aggregator_agent: Agent = None,
        aggregator_system_prompt: str = "",
        layers: int = 3,
    ) -> None:
        """
        Initialize the Mixture of Agents class with agents and configuration.

        Args:
            name (str, optional): The name of the mixture of agents. Defaults to "MixtureOfAgents".
            description (str, optional): A description of the mixture of agents. Defaults to "A class to run a mixture of agents and aggregate their responses.".
            agents (List[Agent], optional): A list of reference agents to be used in the mixture. Defaults to [].
                aggregator_agent (Agent, optional): The aggregator agent to be used in the mixture. Defaults to None.
```

```
        aggregator_system_prompt (str, optional): The system prompt for the aggregator agent. Defaults to "".
        layers (int, optional): The number of layers to process in the mixture. Defaults to 3.
    """
    self.name = name
    self.description = description
    self.agents: List[Agent] = agents
    self.aggregator_agent: Agent = aggregator_agent
    self.aggregator_system_prompt: str = aggregator_system_prompt
    self.layers: int = layers

    self.input_schema = MixtureOfAgentsInput(
        name=name,
        description=description,
        agents=[agent.to_dict() for agent in self.agents],
        aggregator_agent=aggregator_agent.to_dict(),
        aggregator_system_prompt=self.aggregator_system_prompt,
        layers=self.layers,
        time_created=time_stamp,
    )

    self.output_schema = MixtureOfAgentsOutput(
        id="MixtureOfAgents",
        InputConfig=self.input_schema.model_dump(),
        normal_agent_outputs=[],
        aggregator_agent_summary="",
```

```
        task="",
    )

    self.reliability_check()

def reliability_check(self) -> None:
    """
    Performs a reliability check on the Mixture of Agents class.
    """
    logger.info(
        "Checking the reliability of the Mixture of Agents class."
    )

    if not self.agents:
        raise ValueError("No reference agents provided.")

    if not self.aggregator_agent:
        raise ValueError("No aggregator agent provided.")

    if not self.aggregator_system_prompt:
        raise ValueError("No aggregator system prompt provided.")

    if not self.layers:
        raise ValueError("No layers provided.")

    if self.layers < 1:
```

```
            raise ValueError("Layers must be greater than 0.")

        logger.info("Reliability check passed.")
        logger.info("Mixture of Agents class is ready for use.")

    def _get_final_system_prompt(
        self, system_prompt: str, results: List[str]
    ) -> str:
        """
        Constructs a system prompt for subsequent layers that includes previous responses.

        Args:
            system_prompt (str): The initial system prompt.
            results (List[str]): A list of previous responses.

        Returns:
            str: The final system prompt including previous responses.
        """
        return (
            system_prompt
            + "\n"
            + "\n".join(
                [
                    f"{i+1}. {str(element)}"
                    for i, element in enumerate(results)
                ]
```

```
        )
    )

async def _run_agent_async(
    self,
    agent: Agent,
    task: str,
    prev_responses: Optional[List[str]] = None,
) -> str:
    """
    Asynchronous method to run a single agent.

    Args:
        agent (Agent): The agent to be run.
        task (str): The task for the agent.
        prev_responses (Optional[List[str]], optional): A list of previous responses. Defaults to None.

    Returns:
        str: The response from the agent.
    """
    # Update the task in the output schema
    self.output_schema.task = task

    # If there are previous responses, update the agent's system prompt
    if prev_responses:
        system_prompt_with_responses = (
```

```
            self._get_final_system_prompt(
                self.aggregator_system_prompt, prev_responses
            )
        )
        agent.system_prompt = system_prompt_with_responses

    # Run the agent asynchronously
    response = await asyncio.to_thread(agent.run, task)
    self.output_schema.normal_agent_outputs.append(
        agent.agent_output
    )

    # Log the agent's response
    print(f"Agent {agent.agent_name} response: {response}")
    return response

async def _run_async(self, task: str) -> None:
    """
    Asynchronous method to run the Mixture of Agents process.

    Args:
        task (str): The task for the mixture of agents.
    """
    # Gather initial responses from reference agents
    results: List[str] = await asyncio.gather(
        *[
```

```
            self._run_agent_async(agent, task)
            for agent in self.agents
        ]
    )

    # Process additional layers, if applicable
    for _ in range(1, self.layers - 1):
        results = await asyncio.gather(
            *[
                self._run_agent_async(
                    agent, task, prev_responses=results
                )
                for agent in self.agents
            ]
        )

    # Perform final aggregation using the aggregator agent
    final_result = await self._run_agent_async(
        self.aggregator_agent, task, prev_responses=results
    )
    self.output_schema.aggregator_agent_summary = final_result

    print(f"Final Aggregated Response: {final_result}")

def run(self, task: str) -> None:
    """
```

Synchronous wrapper to run the async process.

Args:

    task (str): The task for the mixture of agents.

"""

asyncio.run(self._run_async(task))

self.output_schema.task = task

log_agent_data(self.output_schema.model_dump())

return self.output_schema.model_dump_json(indent=4)