```
import os
import random
import time
from concurrent.futures import ThreadPoolExecutor
from datetime import datetime, timedelta
from threading import Lock
from typing import Any, Callable, Dict, List, Optional

from loguru import logger
from pydantic import BaseModel, Field

from swarms import Agent, create_file_in_folder
from swarms.schemas.agent_step_schemas import ManySteps


class MultiAgentCollaborationSchema(BaseModel):
    name: str = Field(..., title="Name of the collaboration")
    description: str = Field(
        ..., title="Description of the collaboration"
    )
    agent_outputs: List[ManySteps] = Field(
        ..., title="List of agent outputs"
    )
    timestamp: str = Field(
        default_factory=lambda: time.strftime("%Y-%m-%d %H:%M:%S"),
        title="Timestamp of the collaboration",
    )
    number_of_agents: int = Field(
        ..., title="Number of agents in the collaboration"
    )


class Cache:
    def __init__(self, expiration_time: Optional[timedelta] = None):
        """
        Initializes the cache.

        :param expiration_time: Time after which a cache entry should expire.
        """
        self.cache: Dict[str, Dict[str, Any]] = {}
        self.expiration_time = expiration_time
        self.lock = Lock()

    def set(self, key: str, value: Any):
        """
        Stores a value in the cache with an optional expiration time.

        :param key: Cache key.
        :param value: Value to store in the cache.
        """
        with self.lock:
            expiry = (
                datetime.utcnow() + self.expiration_time
                if self.expiration_time
                else None
            )
            self.cache[key] = {"value": value, "expiry": expiry}
            logger.debug(
                f"Cache set for key '{key}' with expiry {expiry}"
            )

    def get(self, key: str) -> Optional[Any]:
        """
        Retrieves a value from the cache.

        :param key: Cache key.
        :return: Cached value if available and not expired, else None.
        """
        with self.lock:
            if key in self.cache:
                entry = self.cache[key]
                if (
                    entry["expiry"]
                    and entry["expiry"] < datetime.utcnow()
                ):
                    logger.debug(f"Cache expired for key '{key}'")
                    del self.cache[key]
                    return None
            logger.debug(f"Cache hit for key '{key}'")
            return entry["value"]
        logger.debug(f"Cache miss for key '{key}'")
        return None


def random_selector(agents: List[Agent], iteration: int) -> Agent:
    """
    Selects a random agent.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number (unused).
    :return: A randomly selected agent.
    """
    return random.choice(agents)


def first_agent_selector(
    agents: List[Agent], iteration: int
) -> Agent:
    """
    Always selects the first agent in the list.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number (unused).
    :return: The first agent in the list.
    """
    return agents[0]


def last_agent_selector(agents: List[Agent], iteration: int) -> Agent:
    """
    Always selects the last agent in the list.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number (unused).
    :return: The last agent in the list.
    """
    return agents[-1]


def reverse_round_robin_selector(
    agents: List[Agent], iteration: int
) -> Agent:
    """
    Selects agents in reverse round-robin order.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number.
    :return: The agent selected in reverse round-robin order.
    """
    index = -((iteration % len(agents)) + 1)
    return agents[index]


def even_iteration_selector(
    agents: List[Agent], iteration: int
) -> Agent:
    """
    Selects the agent based on even iteration; defaults to the first agent if odd.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number.
    :return: The selected agent based on even iteration.
    """
    return (
        agents[iteration % len(agents)]
        if iteration % 2 == 0
        else agents[0]
    )


def odd_iteration_selector(
    agents: List[Agent], iteration: int
) -> Agent:
    """
    Selects the agent based on odd iteration; defaults to the last agent if even.
    :param agents: List of agents to select from.
    :param iteration: The current iteration number.
    :return: The selected agent based on odd iteration.
    """
    return (
        agents[iteration % len(agents)]
        if iteration % 2 != 0
        else agents[-1]
    )


def weighted_random_selector(
    agents: List[Agent], iteration: int
) -> Agent:
    """
    Selects an agent based on weighted random choice, with the first agent having a higher weight.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number (unused).
    :return: A randomly selected agent with weighted preference.
    """
    weights = [1] * len(agents)
    weights[0] = 2  # Give the first agent higher weight
    return random.choices(agents, weights=weights, k=1)[0]
```

```
def increasing_weight_selector(
    agents: List[Agent], iteration: int
) -> Agent:
    """
    Selects an agent based on increasing weight with iteration (favoring later agents).

    :param agents: List of agents to select from.
    :param iteration: The current iteration number (unused).
    :return: A randomly selected agent with increasing weight.
    """
    weights = [i + 1 for i in range(len(agents))]
    return random.choices(agents, weights=weights, k=1)[0]


def decreasing_weight_selector(
    agents: List[Agent], iteration: int
) -> Agent:
    """
    Selects an agent based on decreasing weight with iteration (favoring earlier agents).

    :param agents: List of agents to select from.
    :param iteration: The current iteration number (unused).
    :return: A randomly selected agent with decreasing weight.
    """
    weights = [len(agents) - i for i in range(len(agents))]
    return random.choices(agents, weights=weights, k=1)[0]
```

```
def round_robin_with_skip_selector(
    agents: List[Agent], iteration: int
) -> Agent:
    """
    Selects agents in a round-robin fashion but skips every third agent.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number.
    :return: The selected agent with a skipping pattern.
    """
    index = (iteration * 2) % len(agents)
    return agents[index]


def priority_selector(
    agents: List[Agent], iteration: int, priority_index: int = 0
) -> Agent:
    """
    Selects an agent based on a priority index, always selecting the agent at the given index.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number (unused).
    :param priority_index: The index of the agent with priority.
    :return: The agent at the priority index.
    """
    return agents[priority_index]


def dynamic_priority_selector(
    agents: List[Agent], iteration: int, priorities: List[int] = None
) -> Agent:
    """
    Selects an agent based on dynamic priorities, which can change over iterations.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number.
    :param priorities: A list of priorities for each agent, determining their selection likelihood.
    :return: The selected agent based on dynamic priorities.
    """
    if priorities is None:
        priorities = [1] * len(agents)
    index = random.choices(
        range(len(agents)), weights=priorities, k=1
    )[0]
    return agents[index]


def alternating_selector(
    agents: List[Agent], iteration: int
) -> Agent:
    """
    Alternates between the first and last agent.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number.
    :return: The first agent if the iteration is even, the last if odd.
    """
    return agents[0] if iteration % 2 == 0 else agents[-1]


def middle_agent_selector(
    agents: List[Agent], iteration: int
) -> Agent:
    """
    Always selects the middle agent.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number (unused).
    :return: The middle agent in the list.
    """
    index = len(agents) // 2
    return agents[index]


def weighted_round_robin_selector(
    agents: List[Agent], iteration: int, weights: List[int] = None
) -> Agent:
    """
    Selects agents in a weighted round-robin fashion.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number.
    :param weights: A list of weights to determine the likelihood of selection.
    :return: The selected agent based on weighted round-robin.
    """
    if weights is None:
        weights = [1] * len(agents)
    index = random.choices(range(len(agents)), weights=weights, k=1)[
        0
    ]
    return agents[index]


def even_odd_priority_selector(
    agents: List[Agent], iteration: int
) -> Agent:
    """
        Gives priority to even-indexed agents on even iterations and odd-indexed agents on odd iterations.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number.
    :return: The selected agent based on even/odd priority.
    """
    even_agents = agents[::2]
    odd_agents = agents[1::2]
    return (
        random.choice(even_agents)
        if iteration % 2 == 0
        else random.choice(odd_agents)
    )


def reverse_selector(agents: List[Agent], iteration: int) -> Agent:
    """
    Selects agents in reverse order starting from the last agent.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number.
    :return: The agent selected in reverse order.
    """
    return agents[-(iteration % len(agents)) - 1]


def frequent_first_selector(
    agents: List[Agent], iteration: int, frequency: int = 3
) -> Agent:
    """
    Frequently selects the first agent every 'n' iterations, otherwise selects in round-robin.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number.
    :param frequency: The frequency of selecting the first agent.
    :return: The selected agent with frequent first preference.
    """
    if iteration % frequency == 0:
        return agents[0]
    return agents[iteration % len(agents)]


def frequent_last_selector(
    agents: List[Agent], iteration: int, frequency: int = 3
) -> Agent:
    """
    Frequently selects the last agent every 'n' iterations, otherwise selects in round-robin.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number.
    :param frequency: The frequency of selecting the last agent.
    :return: The selected agent with frequent last preference.
    """
    if iteration % frequency == 0:
        return agents[-1]
    return agents[iteration % len(agents)]
```

```
def random_skip_selector(
    agents: List[Agent], iteration: int, skip_probability: float = 0.5
) -> Agent:
    """
    Randomly skips agents with a given probability, selecting the next in line.

    :param agents: List of agents to select from.
    :param iteration: The current iteration number.
    :param skip_probability: The probability of skipping an agent.
    :return: The selected agent with random skips.
    """
    while random.random() < skip_probability:
        iteration += 1
    return agents[iteration % len(agents)]


def adaptive_selector(
    agents: List[Agent],
    iteration: int,
    performance_metric: Callable[[Agent], float] = None,
) -> Agent:
    """
    Selects the agent based on a performance metric, favoring better-performing agents.
    :param agents: List of agents to select from.
    :param iteration: The current iteration number.
    :param performance_metric: A function to determine the performance of each agent.
    :return: The selected agent based on adaptive performance.
    """
    if performance_metric is None:

        def performance_metric(agent):
            return (
                random.random()
            )  # Default random performance metric

    performance_scores = [
        performance_metric(agent) for agent in agents
    ]
    best_agent_index = performance_scores.index(
        max(performance_scores)
    )
    return agents[best_agent_index]


class MultiAgentCollaboration:
    """
    Initializes the MultiAgentCollaboration.

    :param agents: List of Agent instances.
```

```
:param speaker_fn: Function to select the agent for each loop.

:param max_loops: Maximum number of iterations.

:param use_cache: Boolean to enable or disable caching.

:param autosave_on: Boolean to enable or disable autosaving the output.

"""


def __init__(

    self,

    name: str = "MultiAgentCollaboration",

    description: str = "A collaboration of multiple agents",

    agents: List[Agent] = [],

    speaker_fn: Callable[[List[Agent], int], Agent] = [],

    max_loops: int = 1,

    use_cache: bool = True,

    autosave_on: bool = True,

):

    self.name = name

    self.description = description

    self.agents = agents

    self.speaker_fn = speaker_fn

    self.max_loops = max_loops

    self.autosave_on = autosave_on

    self.lock = Lock()

    self.max_workers = os.cpu_count()

    self.use_cache = use_cache

    logger.info(
            f"Initialized MultiAgentCollaboration with {len(agents)} agents and max_loops={max_loops}"
        )

        # Cache
        self.cache = Cache(expiration_time=timedelta(minutes=5))

        # Output schema
        self.output_schema = MultiAgentCollaborationSchema(
            name=name,
            description=description,
            agent_outputs=[],
            number_of_agents=len(agents),
        )

    def _execute_agent(self, agent: Agent, task: str, loop: int):
        """
        Executes an agent's run method and records the output.

        :param agent: The Agent instance to execute.
        :param task: The input prompt for the agent.
        :param loop: Current loop iteration.
        """
        logger.debug(
            f"Executing agent '{agent.agent_name}' on loop {loop}"
        )
        output = agent.run(task)


        agent_output = agent.agent_output


        self.output_schema.agent_outputs.append(agent_output)


        return output


    def run(self, task: str, *args, **kwargs):
        """
        Runs the agents in sequence, passing the output of one as the input to the next.


        :param task: The input prompt to pass to each agent.
        :return: The final output of the last agent.
        """
        logger.info("Starting MultiAgentCollaboration run.")
        current_task = task
        with ThreadPoolExecutor(
            max_workers=self.max_workers
        ) as executor:
            for i in range(self.max_loops):
                selected_agent = self.speaker_fn(self.agents, i)
                logger.debug(
                    f"Loop {i}: Selected agent '{selected_agent.agent_name}'"
                )
                future = executor.submit(
                self._execute_agent,
                selected_agent,
                current_task,
                i,
                *args,
                **kwargs,
            )
            try:
                current_task = (
                    future.result()
                )  # The output of this agent becomes the input for the next
            except Exception as exc:
                logger.error(
                    f"Loop {i} generated an exception: {exc}"
                )
                break

    logger.info("Completed MultiAgentCollaboration run.")

    if self.autosave_on is True:
        self.save_file()

    return self.return_output_schema_json()

def save_file(self):
    time = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
```

```
        create_file_in_folder(
            "multi_agent_collab_folder",
            f"{self.name}_time_{time}_output.json",
            self.return_output_schema_json(),
        )

    def get_outputs_dict(self):
        """
        Retrieves all recorded agent outputs as a list of dictionaries.

        :return: List of dictionaries representing AgentOutput instances.
        """
        return self.output_schema.model_dump()

    def return_output_schema_json(self):
        return self.output_schema.model_dump_json(indent=4)


def round_robin_speaker(agents: List[Agent], iteration: int) -> Agent:
    """
    Selects an agent from the given list of agents using round-robin scheduling.

    Args:
        agents (List[Agent]): The list of agents to select from.
        iteration (int): The current iteration number.
    Returns:
        Agent: The selected agent.

    """
    selected = agents[iteration % len(agents)]
    logger.debug(
        f"Round-robin selected agent '{selected.agent_name}' for iteration {iteration}"
    )
    return selected


# Example usage
if __name__ == "__main__":
    from swarm_models import OpenAIChat
    from swarms.prompts.finance_agent_sys_prompt import (
        FINANCIAL_AGENT_SYS_PROMPT,
    )

    # Get the OpenAI API key from the environment variable
    api_key = os.getenv("OPENAI_API_KEY")
    if not api_key:
        logger.error(
            "OpenAI API key not found in environment variables."
        )
        exit(1)
```

```
# Create instances of the OpenAIChat class
model = OpenAIChat(
    api_key=api_key, model_name="gpt-4o-mini", temperature=0.1
)

# Initialize agents
agent1 = Agent(
    agent_name="Financial-Analysis-Agent_1",
    system_prompt=FINANCIAL_AGENT_SYS_PROMPT,
    llm=model,
    max_loops=1,
    dynamic_temperature_enabled=True,
    saved_state_path="finance_agent_1.json",
    user_name="swarms_corp",
    retry_attempts=1,
    context_length=200000,
    return_step_meta=False,
)

agent2 = Agent(
    agent_name="Financial-Analysis-Agent_2",
    system_prompt=FINANCIAL_AGENT_SYS_PROMPT,
    llm=model,
    max_loops=1,
    dynamic_temperature_enabled=True,
    saved_state_path="finance_agent_2.json",
    user_name="swarms_corp",
    retry_attempts=1,
    context_length=200000,
    return_step_meta=False,
)


agent2 = Agent(
    agent_name="Financial-Analysis-Agent_3",
    system_prompt=FINANCIAL_AGENT_SYS_PROMPT,
    llm=model,
    max_loops=1,
    dynamic_temperature_enabled=True,
    saved_state_path="finance_agent_2.json",
    user_name="swarms_corp",
    retry_attempts=1,
    context_length=200000,
    return_step_meta=False,
)


# Initialize the MultiAgentCollaboration with the round-robin speaker function
multi_agent_framework = MultiAgentCollaboration(
    agents=[agent1, agent2],
    speaker_fn=round_robin_speaker,
    max_loops=3,
    use_cache=True,  # Enable caching
    autosave_on=True,
)


# Run the framework with an input prompt
task = "How can I establish a ROTH IRA to buy stocks and get a tax break? What are the criteria"
out = multi_agent_framework.run(task)
print(out)


print(multi_agent_framework.return_output_schema_json())
```