```
from typing import List, Any


from pydantic import BaseModel, Field
from swarms.utils.loguru_logger import initialize_logger
from swarms.structs.base_swarm import BaseSwarm
from swarms.structs.agent import Agent
from swarms.structs.concat import concat_strings
from swarms.structs.agent_registry import AgentRegistry
from swarm_models.base_llm import BaseLLM
from swarms.structs.conversation import Conversation


logger = initialize_logger(log_folder="hiearchical_swarm")


# Example usage:
HIEARCHICAL_AGENT_SYSTEM_PROMPT = """
Here's a full-fledged system prompt for a director boss agent, complete with instructions and many-shot examples:


---


**System Prompt: Director Boss Agent**


### Role:
You are a Director Boss Agent responsible for orchestrating a swarm of worker agents. Your primary duty is to serve the user efficiently, effectively, and skillfully. You dynamically create new agents when necessary or utilize existing agents, assigning them tasks that align with their capabilities. You
```

```
must ensure that each agent receives clear, direct, and actionable instructions tailored to their role.

### Key Responsibilities:
1. **Task Delegation:** Assign tasks to the most relevant agent. If no relevant agent exists, create a new one with an appropriate name and system prompt.
2. **Efficiency:** Ensure that tasks are completed swiftly and with minimal resource expenditure.
3. **Clarity:** Provide orders that are simple, direct, and actionable. Avoid ambiguity.
4. **Dynamic Decision Making:** Assess the situation and choose the most effective path, whether that involves using an existing agent or creating a new one.
5. **Monitoring:** Continuously monitor the progress of each agent and provide additional instructions or corrections as necessary.

### Instructions:
- **Identify the Task:** Analyze the input task to determine its nature and requirements.
- **Agent Selection/Creation:**
  - If an agent is available and suited for the task, assign the task to that agent.
  - If no suitable agent exists, create a new agent with a relevant system prompt.
- **Task Assignment:** Provide the selected agent with explicit and straightforward instructions.
- **Reasoning:** Justify your decisions when selecting or creating agents, focusing on the efficiency and effectiveness of task completion.

"""



class AgentSpec(BaseModel):
    """
    A class representing the specifications of an agent.


    Attributes:
        agent_name (str): The name of the agent.
        system_prompt (str): The system prompt for the agent.
        agent_description (str): The description of the agent.
        max_tokens (int): The maximum number of tokens to generate in the API response.
        temperature (float): A parameter that controls the randomness of the generated text.
        context_window (int): The context window for the agent.
        task (str): The main task for the agent.
    """


    agent_name: str = Field(
        ...,
        description="The name of the agent.",
    )
    system_prompt: str = Field(
        ...,
         description="The system prompt for the agent. Write an extremely detailed system prompt for
the agent.",
    )
    agent_description: str = Field(
        ...,
        description="The description of the agent.",
    )
    task: str = Field(
        ...,
        description="The main task for the agent.",
    )



# class AgentTeam(BaseModel):
#     agents: List[AgentSpec] = Field(
#         ...,
#         description="The list of agents in the team",
#     )
#     flow: str = Field(
#         ...,
#         description="Agent Name -> ",
#     )



# Schema to send orders to the agents
class HierarchicalOrderCall(BaseModel):
    agent_name: str = Field(
        ...,
        description="The name of the agent to assign the task to.",
    )
    task: str = Field(
        ...,
        description="The main specific task to be assigned to the agent. Be very specific and direct.",
    )
```

```
# For not agent creation
class CallTeam(BaseModel):
    # swarm_name: str = Field(
    #     ...,
    #     description="The name of the swarm: e.g., 'Marketing Swarm' or 'Finance Swarm'",
    # )
    rules: str = Field(
        ...,
        description="The rules for all the agents in the swarm: e.g., All agents must return code. Be very simple and direct",
    )
    plan: str = Field(
        ...,
        description="The plan for the swarm: e.g., First create the agents, then assign tasks, then monitor progress",
    )
    orders: List[HierarchicalOrderCall]


class SwarmSpec(BaseModel):
    """
    A class representing the specifications of a swarm of agents.

    Attributes:
        multiple_agents (List[AgentSpec]): The list of agents in the swarm.
    """

    swarm_name: str = Field(
        ...,
        description="The name of the swarm: e.g., 'Marketing Swarm' or 'Finance Swarm'",
    )
    multiple_agents: List[AgentSpec]
    rules: str = Field(
        ...,
          description="The rules for all the agents in the swarm: e.g., All agents must return code. Be
very simple and direct",
    )
    plan: str = Field(
        ...,
           description="The plan for the swarm: e.g., First create the agents, then assign tasks, then
monitor progress",
    )



class HierarchicalAgentSwarm(BaseSwarm):
    """
    A class to create and manage a hierarchical swarm of agents.

    Methods:
        __init__(system_prompt, max_tokens, temperature, base_model, parallel_tool_calls): Initializes
```

```
the function caller.
            create_agent(agent_name, system_prompt, agent_description, max_tokens, temperature,
context_window): Creates an individual agent.
            parse_json_for_agents_then_create_agents(function_call): Parses a JSON function call to
create multiple agents.
        run(task): Runs the function caller to create and execute agents based on the provided task.
    """


    def __init__(
        self,
        name: str = "HierarchicalAgentSwarm",
        description: str = "A swarm of agents that can be used to distribute tasks to a team of agents.",
        director: Any = None,
        agents: List[Agent] = None,
        max_loops: int = 1,
        create_agents_on: bool = False,
        template_worker_agent: Agent = None,
        director_planning_prompt: str = None,
        template_base_worker_llm: BaseLLM = None,
        swarm_history: str = None,
        *args,
        **kwargs,
    ):
        """
        Initializes the HierarchicalAgentSwarm with an OpenAIFunctionCaller.
```

```
Args:
    system_prompt (str): The system prompt for the function caller.
    max_tokens (int): The maximum number of tokens to generate in the API response.
    temperature (float): The temperature setting for text generation.
    base_model (BaseModel): The base model for the function caller.
    parallel_tool_calls (bool): Whether to run tool calls in parallel.
"""
super().__init__(
    name=name,
    description=description,
    agents=agents,
    *args,
    **kwargs,
)
self.name = name
self.description = description
self.director = director
self.agents = agents
self.max_loops = max_loops
self.create_agents_on = create_agents_on
self.template_worker_agent = template_worker_agent
self.director_planning_prompt = director_planning_prompt
self.template_base_worker_llm = template_base_worker_llm
self.swarm_history = swarm_history

# Check if the agents are set
```

```
    self.agents_check()

    # Agent Registry
    self.agent_registry = AgentRegistry()

    # Add agents to the registry
    self.add_agents_into_registry(self.agents)

    # Swarm History
    self.conversation = Conversation(time_enabled=True)

    self.swarm_history = (
        self.conversation.return_history_as_string()
    )

def agents_check(self):
    if self.director is None:
        raise ValueError("The director is not set.")

    if len(self.agents) == 0:
        self.create_agents_on = True

    if len(self.agents) > 0:
        self.director.base_model = CallTeam

        self.director.system_prompt = (
            HIEARCHICAL_AGENT_SYSTEM_PROMPT
        )

    if self.max_loops == 0:
        raise ValueError("The max_loops is not set.")

def add_agents_into_registry(self, agents: List[Agent]):
    """
    add_agents_into_registry: Add agents into the agent registry.

    Args:
        agents (List[Agent]): A list of agents to add into the registry.

    Returns:
        None

    """
    for agent in agents:
        self.agent_registry.add(agent)

def create_agent(
    self,
    agent_name: str,
    system_prompt: str,
    agent_description: str,
    task: str = None,
) -> str:
    """
    Creates an individual agent.

    Args:
        agent_name (str): The name of the agent.
        system_prompt (str): The system prompt for the agent.
        agent_description (str): The description of the agent.
        max_tokens (int): The maximum number of tokens to generate.
        temperature (float): The temperature for text generation.
        context_window (int): The context window size for the agent.

    Returns:
        Agent: An instantiated agent object.
    """
    # name = agent_name.replace(" ", "_")
    logger.info(f"Creating agent: {agent_name}")

    agent_name = Agent(
        agent_name=agent_name,
        llm=self.template_base_worker_llm,  # Switch to model router here later
        system_prompt=system_prompt,
        agent_description=agent_description,
        retry_attempts=1,
        verbose=False,
        dashboard=False,
    )

    self.agents.append(agent_name)

    logger.info(f"Running agent: {agent_name} on task: {task}")
    output = agent_name.run(task)

    self.conversation.add(role=agent_name, content=output)
    return output

def parse_json_for_agents_then_create_agents(
    self, function_call: dict
) -> List[Agent]:
    """
    Parses a JSON function call to create a list of agents.

    Args:
        function_call (dict): The JSON function call specifying the agents.

    Returns:
        List[Agent]: A list of created agent objects.
    """
    responses = []
    logger.info("Parsing JSON for agents")

    if self.create_agents_on:
        for agent in function_call["multiple_agents"]:
            out = self.create_agent(
                agent_name=agent["agent_name"],
                system_prompt=agent["system_prompt"],
                agent_description=agent["agent_description"],
                task=agent["task"],
            )
            responses.append(out)
    else:
        for agent in function_call["orders"]:
            out = self.run_worker_agent(
                name=agent["agent_name"],
                task=agent["task"],
            )
            responses.append(out)

    return concat_strings(responses)

def run(self, task: str) -> str:
    """
    Runs the function caller to create and execute agents based on the provided task.

    Args:
        task (str): The task for which the agents need to be created and executed.

    Returns:
            List[Agent]: A list of created agent objects.
        """
        logger.info("Running the swarm")

        # Run the function caller to output JSON function call
        function_call = self.model.run(task)

        # Add the function call to the conversation
        self.conversation.add(
            role="Director", content=str(function_call)
        )

        # Logging the function call with metrics and details
        self.log_director_function_call(function_call)

        # # Parse the JSON function call and create agents -> run Agents
        return self.parse_json_for_agents_then_create_agents(
            function_call
        )

    def run_new(self, task: str):
        """
        Runs the function caller to create and execute agents based on the provided task.

        Args:
            task (str): The task for which the agents need to be created and executed.
    Returns:
        List[Agent]: A list of created agent objects.
    """
    logger.info("Running the swarm")

    # Run the function caller to output JSON function call
    function_call = self.model.run(task)
    self.conversation.add(
        role="Director", content=str(function_call)
    )

    # Logging the function call with metrics and details
    self.log_director_function_call(function_call)

    if self.create_agents_on:
        # Create agents from the function call
        self.create_agents_from_func_call(function_call)

        # Now submit orders to the agents
        self.director.base_model = CallTeam

            orders_prompt = f"Now, the agents have been created. Submit orders to the agents to
enable them to complete the task: {task}: {self.list_agents_available()}"
        orders = self.director.run(orders_prompt)
        self.conversation.add(
            role="Director", content=str(orders_prompt + orders)
        )

        # Check the type of the response
        orders = self.check_agent_output_type(orders)

        # Distribute the orders to the agents
        return self.distribute_orders_to_agents(orders)

    def check_agent_output_type(self, response: Any):
        if isinstance(response, dict):
            return response
        if isinstance(response, str):
            return eval(response)
        else:
            return response

    def distribute_orders_to_agents(self, order_dict: dict) -> str:
        # Now we need to parse the CallTeam object
        # and distribute the orders to the agents
        responses = []

        for order in order_dict["orders"]:
            agent_name = order["agent_name"]
            task = order["task"]
            # Find and run the agent
            response = self.run_worker_agent(
                name=agent_name, task=task
            )

            log = f"Agent: {agent_name} completed task: {task} with response: {response}"
            self.conversation.add(
                role=agent_name, content=task + response
            )
            responses.append(log)
            logger.info(log)

        return concat_strings(responses)


    def create_single_agent(
        self, name: str, system_prompt: str, description
    ) -> Agent:
        """
        Create a single agent from the agent specification.

        Args:
            agent_spec (dict): The agent specification.

        Returns:
            Agent: The created agent.
        """
        # Unwrap all of the agent specifications
        # agent_name = agent_spec["agent_name"]
        # system_prompt = agent_spec["system_prompt"]
        # agent_description = agent_spec["agent_description"]

        # Create the agent
        agent_name = Agent(
            agent_name=name,
            llm=self.template_base_worker_llm,  # Switch to model router here later
            system_prompt=system_prompt,
            agent_description=description,
            max_loops=1,
            retry_attempts=1,
            verbose=False,
            dashboard=False,
        )

        # Add agents into the registry
        self.agents.append(agent_name)

        return agent_name

    def create_agents_from_func_call(self, function_call: dict):
        """
        Create agents from the function call.
        Args:
            function_call (dict): The function call containing the agent specifications.

        Returns:
            List[Agent]: A list of created agents.

        """
        logger.info("Creating agents from the function call")
        for agent_spec in function_call["multiple_agents"]:
            agent = self.create_single_agent(
                name=agent_spec["agent_name"],
                system_prompt=agent_spec["system_prompt"],
                description=agent_spec["agent_description"],
            )

            logger.info(
                f"Created agent: {agent.agent_name} with description: {agent.description}"
            )

            self.agents.append(agent)

    def plan(self, task: str) -> str:
        """
        Plans the tasks for the agents in the swarm.
        Args:
            task (str): The task to be planned.

        Returns:
            str: The planned task for the agents.

        """
        logger.info("Director is planning the task")

        self.director.system_prompt = self.director_planning_prompt

    def log_director_function_call(self, function_call: dict):
        # Log the agents the boss makes\
        logger.info(f"Swarm Name: {function_call['swarm_name']}")
        # Log the plan
        logger.info(f"Plan: {function_call['plan']}")
        logger.info(
            f"Number of agents: {len(function_call['multiple_agents'])}"
        )

        for agent in function_call["multiple_agents"]:
            logger.info(f"Agent: {agent['agent_name']}")
            # logger.info(f"Task: {agent['task']}")
            logger.info(f"Description: {agent['agent_description']}")

    def run_worker_agent(
        self, name: str = None, task: str = None, *args, **kwargs
    ):
        """
        Run the worker agent.

        Args:
            name (str): The name of the worker agent.
            task (str): The task to send to the worker agent.

        Returns:
            str: The response from the worker agent.

        Raises:
            Exception: If an error occurs while running the worker agent.

        """
        try:
            # Find the agent by name
            agent = self.find_agent_by_name(name)

            # Run the agent
            response = agent.run(task, *args, **kwargs)

            return response
        except Exception as e:
            logger.error(f"Error: {e}")
        raise e


    def list_agents(self) -> str:
        logger.info("Listing agents available in the swarm")

        for agent in self.agents:
            name = agent.agent_name
            description = (
                agent.description or "No description available."
            )
            logger.info(f"Agent: {name}, Description: {description}")

    def list_agents_available(self):
        number_of_agents_available = len(self.agents)

        agent_list = "\n".join(
            [
                f"Agent {agent.agent_name}: Description {agent.description}"
                for agent in self.agents
            ]
        )

        prompt = f"""
        There are currently {number_of_agents_available} agents available in the swarm.

        Agents Available:
```

```
    {agent_list}
    """


    return prompt


def find_agent_by_name(
    self, agent_name: str = None, *args, **kwargs
):
    """
    Finds an agent in the swarm by name.


    Args:
        agent_name (str): The name of the agent to find.


    Returns:
        Agent: The agent with the specified name, or None if not found.


    """
    for agent in self.agents:
        if agent.name == agent_name:
            return agent
    return None
```