```
import asyncio
import json
import uuid
from concurrent.futures import ThreadPoolExecutor
from datetime import datetime
from typing import Any, Callable, Dict, List, Optional

from pydantic import BaseModel, Field

from swarms.schemas.agent_step_schemas import ManySteps
from swarms.structs.agent import Agent
from swarms.structs.agents_available import showcase_available_agents
from swarms.structs.base_swarm import BaseSwarm
from swarms.structs.output_types import OutputType
from swarms.utils.loguru_logger import initialize_logger
from swarms.utils.wrapper_clusterop import (
    exec_callable_with_clusterops,
)
from swarms.telemetry.capture_sys_data import log_agent_data

logger = initialize_logger(log_folder="rearrange")


def swarm_id():
    return uuid.uuid4().hex
```

```
class AgentRearrangeInput(BaseModel):
    swarm_id: Optional[str] = None
    name: Optional[str] = None
    description: Optional[str] = None
    flow: Optional[str] = None
    max_loops: Optional[int] = None
    time: str = Field(
        default_factory=lambda: datetime.now().strftime(
            "%Y-%m-%d %H:%M:%S"
        ),
        description="The time the agent was created.",
    )
    output_type: OutputType = Field(default="final")


class AgentRearrangeOutput(BaseModel):
    output_id: str = Field(
        default=swarm_id(), description="Output-UUID"
    )
    input: Optional[AgentRearrangeInput] = None
    outputs: Optional[List[ManySteps]] = None
    time: str = Field(
        default_factory=lambda: datetime.now().strftime(
            "%Y-%m-%d %H:%M:%S"
        ),
```

```
        description="The time the agent was created.",
    )



class AgentRearrange(BaseSwarm):
    """
    A class representing a swarm of agents for rearranging tasks.

    Attributes:
        id (str): Unique identifier for the swarm
        name (str): Name of the swarm
        description (str): Description of the swarm's purpose
        agents (callable): Dictionary mapping agent names to Agent objects
        flow (str): The flow pattern defining task execution order
        max_loops (int): Maximum number of execution loops
        verbose (bool): Whether to enable verbose logging
        memory_system (BaseVectorDatabase): Memory system for storing agent interactions
        human_in_the_loop (bool): Whether human intervention is enabled
        custom_human_in_the_loop (Callable): Custom function for human intervention
        return_json (bool): Whether to return output in JSON format
        output_type (OutputType): Format of output ("all", "final", "list", or "dict")
        swarm_history (dict): History of agent interactions
        input_config (AgentRearrangeInput): Input configuration schema
        output_schema (AgentRearrangeOutput): Output schema

    Methods:
```

```
    __init__(): Initializes the AgentRearrange object
    reliability_checks(): Validates swarm configuration
    set_custom_flow(): Sets a custom flow pattern
    add_agent(): Adds an agent to the swarm
    track_history(): Records agent interaction history
    remove_agent(): Removes an agent from the swarm
    add_agents(): Adds multiple agents to the swarm
    validate_flow(): Validates the flow pattern
    run(): Executes the swarm's task processing
    astream(): Runs the swarm with streaming output
    batch_run(): Processes multiple tasks in batches
    abatch_run(): Asynchronously processes multiple tasks in batches
    concurrent_run(): Processes multiple tasks concurrently

"""

def __init__(
    self,
    id: str = swarm_id(),
    name: str = "AgentRearrange",
    description: str = "A swarm of agents for rearranging tasks.",
    agents: List[Agent] = None,
    flow: str = None,
    max_loops: int = 1,
    verbose: bool = True,
    memory_system: Any = None,
```

```
    human_in_the_loop: bool = False,
    custom_human_in_the_loop: Optional[
        Callable[[str], str]
    ] = None,
    return_json: bool = False,
    output_type: OutputType = "all",
    docs: List[str] = None,
    doc_folder: str = None,
    device: str = "cpu",
    device_id: int = 0,
    all_cores: bool = False,
    all_gpus: bool = True,
    no_use_clusterops: bool = True,
    autosave: bool = True,
    *args,
    **kwargs,
):
    super(AgentRearrange, self).__init__(
        name=name,
        description=description,
        agents=agents if agents else [],
        *args,
        **kwargs,
    )
    self.id = id
    self.agents = {agent.agent_name: agent for agent in agents}
```

```
self.flow = flow if flow is not None else ""
self.verbose = verbose
self.max_loops = max_loops if max_loops > 0 else 1
self.memory_system = memory_system
self.human_in_the_loop = human_in_the_loop
self.custom_human_in_the_loop = custom_human_in_the_loop
self.return_json = return_json
self.output_type = output_type
self.docs = docs
self.doc_folder = doc_folder
self.device = device
self.device_id = device_id
self.all_cores = all_cores
self.all_gpus = all_gpus
self.no_use_clusterops = no_use_clusterops
self.autosave = autosave

self.output_schema = AgentRearrangeOutput(
    input=AgentRearrangeInput(
        swarm_id=id,
        name=name,
        description=description,
        flow=flow,
        max_loops=max_loops,
    ),
    outputs=[],
```

```
    )


def showcase_agents(self):
    # Get formatted agent info once
    agents_available = showcase_available_agents(
        name=self.name,
        description=self.description,
        agents=self.agents,
        format="Table",
    )

    return agents_available


def rearrange_prompt_prep(self) -> str:
    """Prepares a formatted prompt describing the swarm configuration.

    Returns:
        str: A formatted string containing the swarm's name, description,
            flow pattern, and participating agents.
    """
    agents_available = self.showcase_agents()
    prompt = f"""
    ===== Swarm Configuration =====

    Name: {self.name}
    Description: {self.description}
```

```
    ===== Execution Flow =====
    {self.flow}

    ===== Participating Agents =====
    {agents_available}

    ============================
    """
    return prompt

def set_custom_flow(self, flow: str):
    self.flow = flow
    logger.info(f"Custom flow set: {flow}")

def add_agent(self, agent: Agent):
    """
    Adds an agent to the swarm.

    Args:
        agent (Agent): The agent to be added.
    """
    logger.info(f"Adding agent {agent.agent_name} to the swarm.")
    self.agents[agent.agent_name] = agent

def track_history(
```

```
    self,
    agent_name: str,
    result: str,
):
    self.swarm_history[agent_name].append(result)


def remove_agent(self, agent_name: str):
    """
    Removes an agent from the swarm.

    Args:
        agent_name (str): The name of the agent to be removed.
    """
    del self.agents[agent_name]


def add_agents(self, agents: List[Agent]):
    """
    Adds multiple agents to the swarm.

    Args:
        agents (List[Agent]): A list of Agent objects.
    """
    for agent in agents:
        self.agents[agent.agent_name] = agent


def validate_flow(self):
```

```
"""
Validates the flow pattern.

Raises:
    ValueError: If the flow pattern is incorrectly formatted or contains duplicate agent names.

Returns:
    bool: True if the flow pattern is valid.
"""
if "->" not in self.flow:
    raise ValueError(
        "Flow must include '->' to denote the direction of the task."
    )

agents_in_flow = []

# Arrow
tasks = self.flow.split("->")

# For the task in tasks
for task in tasks:
    agent_names = [name.strip() for name in task.split(",")]

    # Loop over the agent names
    for agent_name in agent_names:
        if (
```

```
                agent_name not in self.agents
                and agent_name != "H"
            ):
                raise ValueError(
                    f"Agent '{agent_name}' is not registered."
                )
            agents_in_flow.append(agent_name)

    # If the length of the agents does not equal the length of the agents in flow
    if len(set(agents_in_flow)) != len(agents_in_flow):
        raise ValueError(
            "Duplicate agent names in the flow are not allowed."
        )

    logger.info(f"Flow: {self.flow} is valid.")
    return True

def _run(
    self,
    task: str = None,
    img: str = None,
    custom_tasks: Dict[str, str] = None,
    *args,
    **kwargs,
):
    """
```

```
Runs the swarm to rearrange the tasks.

Args:
    task (str, optional): The initial task to be processed. Defaults to None.
    img (str, optional): Image input for agents that support it. Defaults to None.
    custom_tasks (Dict[str, str], optional): Custom tasks for specific agents. Defaults to None.
    output_type (str, optional): Format of the output. Can be:
        - "all": String containing all agent responses concatenated
        - "final": Only the final agent's response
        - "list": List of all agent responses
        - "dict": Dict mapping agent names to their responses
        Defaults to "final".
    *args: Additional positional arguments
    **kwargs: Additional keyword arguments

Returns:
    Union[str, List[str], Dict[str, str]]: The processed output in the specified format

Raises:
    ValueError: If flow validation fails
    Exception: For any other errors during execution
"""
try:
    if not self.validate_flow():
        logger.error("Flow validation failed")
        return "Invalid flow configuration."
```

```
tasks = self.flow.split("->")
current_task = task
all_responses = []
response_dict = {}
previous_agent = None

logger.info(
    f"Starting task execution with {len(tasks)} steps"
)

# Handle custom tasks
if custom_tasks is not None:
    logger.info("Processing custom tasks")
    c_agent_name, c_task = next(
        iter(custom_tasks.items())
    )
    position = tasks.index(c_agent_name)

    if position > 0:
        tasks[position - 1] += "->" + c_task
    else:
        tasks.insert(position, c_task)

loop_count = 0
while loop_count < self.max_loops:
```

```
logger.info(
    f"Starting loop {loop_count + 1}/{self.max_loops}"
)

for task_idx, task in enumerate(tasks):
    is_last = task == tasks[-1]
    agent_names = [
        name.strip() for name in task.split(",")
    ]

    # Prepare prompt with previous agent info
    prompt_prefix = ""
    if previous_agent and task_idx > 0:
        prompt_prefix = f"Previous agent {previous_agent} output: {current_task}\n"
    elif task_idx == 0:
        prompt_prefix = "Initial task: "

    if len(agent_names) > 1:
        # Parallel processing
        logger.info(
            f"Running agents in parallel: {agent_names}"
        )
        results = []

        for agent_name in agent_names:
            if agent_name == "H":
```

```
    if (
        self.human_in_the_loop
        and self.custom_human_in_the_loop
    ):
        current_task = (
            self.custom_human_in_the_loop(
                prompt_prefix
                + str(current_task)
            )
        )
    else:
        current_task = input(
            prompt_prefix
            + "Enter your response: "
        )
    results.append(current_task)
    response_dict[agent_name] = (
        current_task
    )
else:
    agent = self.agents[agent_name]
    task_with_context = (
        prompt_prefix + str(current_task)
        if current_task
        else prompt_prefix
    )
```

```
                result = agent.run(
                    task=task_with_context,
                    img=img,
                    is_last=is_last,
                    *args,
                    **kwargs,
                )
                result = str(result)
                results.append(result)
                response_dict[agent_name] = result
                self.output_schema.outputs.append(
                    agent.agent_output
                )
                logger.debug(
                    f"Agent {agent_name} output: {result}"
                )

            current_task = "; ".join(results)
            all_responses.extend(results)
            previous_agent = ",".join(agent_names)

        else:
            # Sequential processing
            logger.info(
                f"Running agent sequentially: {agent_names[0]}"
            )
```

```
agent_name = agent_names[0]


if agent_name == "H":
    if (
        self.human_in_the_loop
        and self.custom_human_in_the_loop
    ):
        current_task = (
            self.custom_human_in_the_loop(
                prompt_prefix
                + str(current_task)
            )
        )
    else:
        current_task = input(
            prompt_prefix
            + "Enter the next task: "
        )
    response_dict[agent_name] = current_task
else:
    agent = self.agents[agent_name]
    task_with_context = (
        prompt_prefix + str(current_task)
        if current_task
        else prompt_prefix
    )
```

```
                current_task = agent.run(
                    task=task_with_context,
                    img=img,
                    is_last=is_last,
                    *args,
                    **kwargs,
                )
                current_task = str(current_task)
                response_dict[agent_name] = current_task
                self.output_schema.outputs.append(
                    agent.agent_output
                )
                logger.debug(
                    f"Agent {agent_name} output: {current_task}"
                )

            all_responses.append(
                f"Agent Name: {agent.agent_name} \n Output: {current_task} "
            )
            previous_agent = agent_name

    loop_count += 1

logger.info("Task execution completed")

if self.return_json:
```

```
                return self.output_schema.model_dump_json(indent=4)

            # Handle different output types
            if self.output_type == "all":
                output = " ".join(all_responses)
            elif self.output_type == "list":
                output = all_responses
            elif self.output_type == "dict":
                output = response_dict
            else:  # "final"
                output = current_task

            return output

        except Exception as e:
            self._catch_error(e)

    def _catch_error(self, e: Exception):
        if self.autosave is True:
            log_agent_data(self.to_dict())

                logger.error(f"An error occurred with your swarm {self.name}: Error: {e} Traceback: {e.__traceback__}")

        return e
```

```
def run(
    self,
    task: str = None,
    img: str = None,
    device: str = "cpu",
    device_id: int = 2,
    all_cores: bool = True,
    all_gpus: bool = False,
    no_use_clusterops: bool = True,
    *args,
    **kwargs,
):
    """
    Execute the agent rearrangement task with specified compute resources.

    Args:
        task (str, optional): The task to execute. Defaults to None.
        img (str, optional): Path to input image if required. Defaults to None.
        device (str, optional): Computing device to use ('cpu' or 'gpu'). Defaults to "cpu".
        device_id (int, optional): ID of specific device to use. Defaults to 1.
        all_cores (bool, optional): Whether to use all CPU cores. Defaults to True.
        all_gpus (bool, optional): Whether to use all available GPUs. Defaults to False.
        no_use_clusterops (bool, optional): Whether to use clusterops. Defaults to False.
        *args: Additional positional arguments passed to _run().
        **kwargs: Additional keyword arguments passed to _run().
```

```
Returns:
    The result from executing the task through the cluster operations wrapper.
"""
try:
    no_use_clusterops = (
        no_use_clusterops or self.no_use_clusterops
    )

    if no_use_clusterops is True:
        return self._run(
            task=task,
            img=img,
            *args,
            **kwargs,
        )
    else:
        return exec_callable_with_clusterops(
            device=device,
            device_id=device_id,
            all_cores=all_cores,
            all_gpus=all_gpus,
            func=self._run,
            task=task,
            img=img,
            *args,
            **kwargs,
```

```
            )
        except Exception as e:
            self._catch_error(e)


    def __call__(self, task: str, *args, **kwargs):
        """
        Make the class callable by executing the run() method.

        Args:
            task (str): The task to execute.
            *args: Additional positional arguments passed to run().
            **kwargs: Additional keyword arguments passed to run().

        Returns:
            The result from executing run().
        """
        try:
            return self.run(task=task, *args, **kwargs)
        except Exception as e:
            logger.error(f"An error occurred: {e}")
            return e


    def batch_run(
        self,
        tasks: List[str],
        img: Optional[List[str]] = None,
```

```
    batch_size: int = 10,
    device: str = "cpu",
    device_id: int = None,
    all_cores: bool = True,
    all_gpus: bool = False,
    *args,
    **kwargs,
) -> List[str]:
    """
    Process multiple tasks in batches.

    Args:
        tasks: List of tasks to process
        img: Optional list of images corresponding to tasks
        batch_size: Number of tasks to process simultaneously
        device: Computing device to use
        device_id: Specific device ID if applicable
        all_cores: Whether to use all CPU cores
        all_gpus: Whether to use all available GPUs

    Returns:
        List of results corresponding to input tasks
    """
    try:
        results = []
        for i in range(0, len(tasks), batch_size):
```

```
            batch_tasks = tasks[i : i + batch_size]
            batch_imgs = (
                img[i : i + batch_size]
                if img
                else [None] * len(batch_tasks)
            )

            # Process batch using concurrent execution
            batch_results = [
                self.run(
                    task=task,
                    img=img_path,
                    device=device,
                    device_id=device_id,
                    all_cores=all_cores,
                    all_gpus=all_gpus,
                    *args,
                    **kwargs,
                )
                for task, img_path in zip(batch_tasks, batch_imgs)
            ]
            results.extend(batch_results)

        return results
    except Exception as e:
        self._catch_error(e)
```

```
async def abatch_run(
    self,
    tasks: List[str],
    img: Optional[List[str]] = None,
    batch_size: int = 10,
    *args,
    **kwargs,
) -> List[str]:
    """
    Asynchronously process multiple tasks in batches.

    Args:
        tasks: List of tasks to process
        img: Optional list of images corresponding to tasks
        batch_size: Number of tasks to process simultaneously

    Returns:
        List of results corresponding to input tasks
    """
    try:
        results = []
        for i in range(0, len(tasks), batch_size):
            batch_tasks = tasks[i : i + batch_size]
            batch_imgs = (
                img[i : i + batch_size]
```

```
                if img
                else [None] * len(batch_tasks)
            )

            # Process batch using asyncio.gather
            batch_coros = [
                self.astream(task=task, img=img_path, *args, **kwargs)
                for task, img_path in zip(batch_tasks, batch_imgs)
            ]
            batch_results = await asyncio.gather(*batch_coros)
            results.extend(batch_results)

        return results
    except Exception as e:
        self._catch_error(e)

def concurrent_run(
    self,
    tasks: List[str],
    img: Optional[List[str]] = None,
    max_workers: Optional[int] = None,
    device: str = "cpu",
    device_id: int = None,
    all_cores: bool = True,
    all_gpus: bool = False,
    *args,
```

```
    **kwargs,
) -> List[str]:
    """
    Process multiple tasks concurrently using ThreadPoolExecutor.

    Args:
        tasks: List of tasks to process
        img: Optional list of images corresponding to tasks
        max_workers: Maximum number of worker threads
        device: Computing device to use
        device_id: Specific device ID if applicable
        all_cores: Whether to use all CPU cores
        all_gpus: Whether to use all available GPUs

    Returns:
        List of results corresponding to input tasks
    """
    try:
        with ThreadPoolExecutor(max_workers=max_workers) as executor:
            imgs = img if img else [None] * len(tasks)
            futures = [
                executor.submit(
                    self.run,
                    task=task,
                    img=img_path,
                    device=device,
```

```
                    device_id=device_id,
                    all_cores=all_cores,
                    all_gpus=all_gpus,
                    *args,
                    **kwargs,
                )
                for task, img_path in zip(tasks, imgs)
            ]
            return [future.result() for future in futures]
    except Exception as e:
        self._catch_error(e)



def _serialize_callable(
    self, attr_value: Callable
) -> Dict[str, Any]:
    """
    Serializes callable attributes by extracting their name and docstring.

    Args:
        attr_value (Callable): The callable to serialize.

    Returns:
        Dict[str, Any]: Dictionary with name and docstring of the callable.
    """
    return {
```

```
        "name": getattr(
            attr_value, "__name__", type(attr_value).__name__
        ),
        "doc": getattr(attr_value, "__doc__", None),
    }

def _serialize_attr(self, attr_name: str, attr_value: Any) -> Any:
    """
    Serializes an individual attribute, handling non-serializable objects.

    Args:
        attr_name (str): The name of the attribute.
        attr_value (Any): The value of the attribute.

    Returns:
        Any: The serialized value of the attribute.
    """
    try:
        if callable(attr_value):
            return self._serialize_callable(attr_value)
        elif hasattr(attr_value, "to_dict"):
            return (
                attr_value.to_dict()
            )  # Recursive serialization for nested objects
        else:
            json.dumps(
```

```
                attr_value
            )  # Attempt to serialize to catch non-serializable objects
            return attr_value
        except (TypeError, ValueError):
            return f"<Non-serializable: {type(attr_value).__name__}>"

    def to_dict(self) -> Dict[str, Any]:
        """
        Converts all attributes of the class, including callables, into a dictionary.
        Handles non-serializable attributes by converting them or skipping them.

        Returns:
            Dict[str, Any]: A dictionary representation of the class attributes.
        """
        return {
            attr_name: self._serialize_attr(attr_name, attr_value)
            for attr_name, attr_value in self.__dict__.items()
        }



def rearrange(
    agents: List[Agent] = None,
    flow: str = None,
    task: str = None,
    img: str = None,
```

```
    *args,
    **kwargs,
):
    """
    Rearranges the given list of agents based on the specified flow.

    Parameters:
        agents (List[Agent]): The list of agents to be rearranged.
        flow (str): The flow used for rearranging the agents.
        task (str, optional): The task to be performed during rearrangement. Defaults to None.
        *args: Additional positional arguments.
        **kwargs: Additional keyword arguments.

    Returns:
        The result of running the agent system with the specified task.

    Example:
        agents = [agent1, agent2, agent3]
        flow = "agent1 -> agent2, agent3"
        task = "Perform a task"
        rearrange(agents, flow, task)
    """
    agent_system = AgentRearrange(
        agents=agents, flow=flow, *args, **kwargs
    )
    return agent_system.run(task, img=img, *args, **kwargs)
```