```
import os
import asyncio
import json
import uuid
from swarms.utils.file_processing import create_file_in_folder
from abc import ABC
from concurrent.futures import ThreadPoolExecutor, as_completed
from typing import (
    Any,
    Callable,
    Dict,
    List,
    Optional,
    Sequence,
)

import yaml

from swarms.structs.agent import Agent
from swarms.structs.conversation import Conversation
from swarms.structs.omni_agent_types import AgentType
from pydantic import BaseModel
from swarms.utils.pandas_utils import (
    dict_to_dataframe,
    display_agents_info,
    pydantic_model_to_dataframe,
```

```
)
from swarms.utils.loguru_logger import initialize_logger


logger = initialize_logger(log_folder="base_swarm")



class BaseSwarm(ABC):
    """
    Base Swarm Class for all multi-agent systems

    Attributes:
        agents (List[Agent]): A list of agents
        max_loops (int): The maximum number of loops to run


    Methods:
            communicate: Communicate with the swarm through the orchestrator, protocols, and the universal communication layer
        run: Run the swarm
        step: Step the swarm
        add_agent: Add a agent to the swarm
        remove_agent: Remove a agent from the swarm
        broadcast: Broadcast a message to all agents
        reset: Reset the swarm
        plan: agents must individually plan using a workflow or pipeline
        direct_message: Send a direct message to a agent
```

autoscaler: Autoscaler that acts like kubernetes for autonomous agents

get_agent_by_id: Locate a agent by id

get_agent_by_name: Locate a agent by name

assign_task: Assign a task to a agent

get_all_tasks: Get all tasks

get_finished_tasks: Get all finished tasks

get_pending_tasks: Get all penPding tasks

pause_agent: Pause a agent

resume_agent: Resume a agent

stop_agent: Stop a agent

restart_agent: Restart agent

scale_up: Scale up the number of agents

scale_down: Scale down the number of agents

scale_to: Scale to a specific number of agents

get_all_agents: Get all agents

get_swarm_size: Get the size of the swarm

get_swarm_status: Get the status of the swarm

save_swarm_state: Save the swarm state

loop: Loop through the swarm

run_async: Run the swarm asynchronously

run_batch_async: Run the swarm asynchronously

run_batch: Run the swarm asynchronously

batched_run: Run the swarm asynchronously

abatch_run: Asynchronous batch run with language model

arun: Asynchronous run

"""

```
def __init__(
    self,
    name: Optional[str] = None,
    description: Optional[str] = None,
    agents: Optional[List[Agent]] = None,
    models: Optional[List[Any]] = None,
    max_loops: Optional[int] = 200,
    callbacks: Optional[Sequence[callable]] = None,
    autosave: Optional[bool] = False,
    logging: Optional[bool] = False,
    return_metadata: Optional[bool] = False,
    metadata_filename: Optional[
        str
    ] = "multiagent_structure_metadata.json",
    stopping_function: Optional[Callable] = None,
    stopping_condition: Optional[str] = "stop",
    stopping_condition_args: Optional[Dict] = None,
    agentops_on: Optional[bool] = False,
    speaker_selection_func: Optional[Callable] = None,
    rules: Optional[str] = None,
    collective_memory_system: Optional[Any] = False,
    agent_ops_on: bool = False,
    output_schema: Optional[BaseModel] = None,
    *args,
```

```
    **kwargs,
):
    """Initialize the swarm with agents"""
    self.name = name
    self.description = description
    self.agents = agents
    self.models = models
    self.max_loops = max_loops
    self.callbacks = callbacks
    self.autosave = autosave
    self.logging = logging
    self.return_metadata = return_metadata
    self.metadata_filename = metadata_filename
    self.stopping_function = stopping_function
    self.stopping_condition = stopping_condition
    self.stopping_condition_args = stopping_condition_args
    self.agentops_on = agentops_on
    self.speaker_selection_func = speaker_selection_func
    self.rules = rules
    self.collective_memory_system = collective_memory_system
    self.agent_ops_on = agent_ops_on
    self.output_schema = output_schema

    logger.info("Reliability checks activated.")
    # Ensure that agents is exists
    if self.agents is None:
```

```
    logger.info("Agents must be provided.")
    raise ValueError("Agents must be provided.")

# Ensure that agents is a list
if not isinstance(self.agents, list):
    logger.error("Agents must be a list.")
    raise TypeError("Agents must be a list.")

# Ensure that agents is not empty
# if len(self.agents) == 0:
#     logger.error("Agents list must not be empty.")
#     raise ValueError("Agents list must not be empty.")

# Initialize conversation
self.conversation = Conversation(
    time_enabled=True, rules=self.rules, *args, **kwargs
)

# Handle callbacks
if callbacks is not None:
    for callback in self.callbacks:
        if not callable(callback):
            raise TypeError("Callback must be callable.")

# Handle autosave
if autosave:
```

```
    self.save_to_json(metadata_filename)


# Handle stopping function
if stopping_function is not None:
    if not callable(stopping_function):
        raise TypeError("Stopping function must be callable.")
    if stopping_condition_args is None:
        stopping_condition_args = {}
    self.stopping_condition_args = stopping_condition_args
    self.stopping_condition = stopping_condition
    self.stopping_function = stopping_function


# Handle stopping condition
if stopping_condition is not None:
    if stopping_condition_args is None:
        stopping_condition_args = {}
    self.stopping_condition_args = stopping_condition_args
    self.stopping_condition = stopping_condition


# If agentops is enabled, try to import agentops
if agentops_on is True:
    for agent in self.agents:
        agent.agent_ops_on = True


# Handle speaker selection function
if speaker_selection_func is not None:
```

```
        if not callable(speaker_selection_func):
            raise TypeError(
                "Speaker selection function must be callable."
            )
        self.speaker_selection_func = speaker_selection_func

    # Add the check for all the agents to see if agent ops is on!
    if agent_ops_on is True:
        for agent in self.agents:
            agent.agent_ops_on = True

    # Agents dictionary with agent name as key and agent object as value
    self.agents_dict = {
        agent.agent_name: agent for agent in self.agents
    }

def communicate(self):
        """Communicate with the swarm through the orchestrator, protocols, and the universal communication layer"""
    ...

def run(self):
    """Run the swarm"""
    ...

def __call__(
```

```
    self,
    task,
    *args,
    **kwargs,
):
    """Call self as a function

    Args:
        task (_type_): _description_

    Returns:
        _type_: _description_
    """
    try:
        return self.run(task, *args, **kwargs)
    except Exception as error:
        logger.error(f"Error running {self.__class__.__name__}")
        raise error

def step(self):
    """Step the swarm"""

def add_agent(self, agent: AgentType):
    """Add a agent to the swarm"""
    self.agents.append(agent)
```

```
def add_agents(self, agents: List[AgentType]):
    """Add a list of agents to the swarm"""
    self.agents.extend(agents)

def add_agent_by_id(self, agent_id: str):
    """Add a agent to the swarm by id"""
    agent = self.get_agent_by_id(agent_id)
    self.add_agent(agent)

def remove_agent(self, agent: AgentType):
    """Remove a agent from the swarm"""
    self.agents.remove(agent)

def get_agent_by_name(self, name: str):
    """Get a agent by name"""
    for agent in self.agents:
        if agent.name == name:
            return agent

def reset_all_agents(self):
    """Resets the state of all agents."""
    for agent in self.agents:
        agent.reset()

def broadcast(
    self, message: str, sender: Optional[AgentType] = None
```

```
):
    """Broadcast a message to all agents"""

def reset(self):
    """Reset the swarm"""

def plan(self, task: str):
    """agents must individually plan using a workflow or pipeline"""

def self_find_agent_by_name(self, name: str):
    """
    Find an agent by its name.

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

    Returns:
        Agent: The Agent object if found, None otherwise.
    """
    for agent in self.agents:
        if agent.agent_name == name:
            return agent
    return None

def self_find_agent_by_id(self, id: uuid.UUID):
    """
```

```
    Find an agent by its id.

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

    Returns:
        Agent: The Agent object if found, None otherwise.
    """
    for agent in self.agents:
        if agent.id == id:
            return agent
    return None

def agent_exists(self, name: str):
    """
    Check if an agent exists in the swarm.

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

    Returns:
        bool: True if the agent exists, False otherwise.
    """
    return self.self_find_agent_by_name(name) is not None

def direct_message(
```

```
    self,
    message: str,
    sender: AgentType,
    recipient: AgentType,
):
    """Send a direct message to a agent"""

def autoscaler(self, num_agents: int, agent: List[AgentType]):
    """Autoscaler that acts like kubernetes for autonomous agents"""

def get_agent_by_id(self, id: str) -> AgentType:
    """Locate a agent by id"""

def assign_task(self, agent: AgentType, task: Any) -> Dict:
    """Assign a task to a agent"""

def get_all_tasks(self, agent: AgentType, task: Any):
    """Get all tasks"""

def get_finished_tasks(self) -> List[Dict]:
    """Get all finished tasks"""

def get_pending_tasks(self) -> List[Dict]:
    """Get all pending tasks"""

def pause_agent(self, agent: AgentType, agent_id: str):
```

```
    """Pause a agent"""

def resume_agent(self, agent: AgentType, agent_id: str):
    """Resume a agent"""

def stop_agent(self, agent: AgentType, agent_id: str):
    """Stop a agent"""

def restart_agent(self, agent: AgentType):
    """Restart agent"""

def scale_up(self, num_agent: int):
    """Scale up the number of agents"""

def scale_down(self, num_agent: int):
    """Scale down the number of agents"""

def scale_to(self, num_agent: int):
    """Scale to a specific number of agents"""

def get_all_agents(self) -> List[AgentType]:
    """Get all agents"""

def get_swarm_size(self) -> int:
    """Get the size of the swarm"""
```

```
# #@abstractmethod
def get_swarm_status(self) -> Dict:
    """Get the status of the swarm"""

# #@abstractmethod
def save_swarm_state(self):
    """Save the swarm state"""

def batched_run(self, tasks: List[Any], *args, **kwargs):
    """_summary_

    Args:
        tasks (List[Any]): _description_
    """
    # Implement batched run
    return [self.run(task, *args, **kwargs) for task in tasks]

async def abatch_run(self, tasks: List[str], *args, **kwargs):
    """Asynchronous batch run with language model

    Args:
        tasks (List[str]): _description_

    Returns:
        _type_: _description_
    """
```

```
    return await asyncio.gather(
        *(self.arun(task, *args, **kwargs) for task in tasks)
    )

async def arun(self, task: Optional[str] = None, *args, **kwargs):
    """Asynchronous run

    Args:
        task (Optional[str], optional): _description_. Defaults to None.
    """
    loop = asyncio.get_event_loop()
    result = await loop.run_in_executor(
        None, self.run, task, *args, **kwargs
    )
    return result

def loop(
    self,
    task: Optional[str] = None,
    *args,
    **kwargs,
):
    """Loop through the swarm

    Args:
        task (Optional[str], optional): _description_. Defaults to None.
```

```
    """
    # Loop through the self.max_loops
    for i in range(self.max_loops):
        self.run(task, *args, **kwargs)

async def aloop(
    self,
    task: Optional[str] = None,
    *args,
    **kwargs,
):
    """Asynchronous loop through the swarm

    Args:
        task (Optional[str], optional): _description_. Defaults to None.
    """
    # Async Loop through the self.max_loops
    loop = asyncio.get_event_loop()
    result = await loop.run_in_executor(
        None, self.loop, task, *args, **kwargs
    )
    return result

def run_async(self, task: Optional[str] = None, *args, **kwargs):
    """Run the swarm asynchronously
```

```
    Args:
        task (Optional[str], optional): _description_. Defaults to None.
    """
    loop = asyncio.get_event_loop()
    result = loop.run_until_complete(
        self.arun(task, *args, **kwargs)
    )
    return result

def run_batch_async(self, tasks: List[str], *args, **kwargs):
    """Run the swarm asynchronously

    Args:
        task (Optional[str], optional): _description_. Defaults to None.
    """
    loop = asyncio.get_event_loop()
    result = loop.run_until_complete(
        self.abatch_run(tasks, *args, **kwargs)
    )
    return result

def run_batch(self, tasks: List[str], *args, **kwargs):
    """Run the swarm asynchronously

    Args:
        task (Optional[str], optional): _description_. Defaults to None.
```

```
    """
    return self.batched_run(tasks, *args, **kwargs)


def select_agent_by_name(self, agent_name: str):
    """
    Select an agent through their name
    """
    # Find agent with id
    for agent in self.agents:
        if agent.name == agent_name:
            return agent


def task_assignment_by_id(
    self, task: str, agent_id: str, *args, **kwargs
):
    """
    Assign a task to an agent
    """
    # Assign task to agent by their agent id
    agent = self.select_agent(agent_id)
    return agent.run(task, *args, **kwargs)


def task_assignment_by_name(
    self, task: str, agent_name: str, *args, **kwargs
):
    """
```

```
    Assign a task to an agent
    """
    # Assign task to agent by their agent id
    agent = self.select_agent_by_name(agent_name)
    return agent.run(task, *args, **kwargs)


def concurrent_run(self, task: str) -> List[str]:
    """Synchronously run the task on all llms and collect responses"""
    with ThreadPoolExecutor() as executor:
        future_to_llm = {
            executor.submit(agent, task): agent
            for agent in self.agents
        }
        responses = []
        for future in as_completed(future_to_llm):
            try:
                responses.append(future.result())
            except Exception as error:
                print(
                    f"{future_to_llm[future]} generated an"
                    f" exception: {error}"
                )
    self.last_responses = responses
    self.task_history.append(task)
    return responses
```

```
def add_llm(self, agent: Callable):
    """Add an llm to the god mode"""
    self.agents.append(agent)

def remove_llm(self, agent: Callable):
    """Remove an llm from the god mode"""
    self.agents.remove(agent)

def run_all(self, task: str = None, *args, **kwargs):
    """Run all agents

    Args:
        task (str, optional): _description_. Defaults to None.

    Returns:
        _type_: _description_
    """
    responses = []
    for agent in self.agents:
        responses.append(agent(task, *args, **kwargs))
    return responses

def run_on_all_agents(self, task: str = None, *args, **kwargs):
    """Run on all agents

    Args:
```

```
            task (str, optional): _description_. Defaults to None.

        Returns:
            _type_: _description_
        """
        with ThreadPoolExecutor() as executor:
            responses = executor.map(
                lambda agent: agent(task, *args, **kwargs),
                self.agents,
            )
        return list(responses)

    def add_swarm_entry(self, swarm):
        """
        Add the information of a joined Swarm to the registry.

        Args:
            swarm (SwarmManagerBase): Instance of SwarmManagerBase representing the joined Swarm.

        Returns:
            None
        """

    def add_agent_entry(self, agent: Agent):
        """
```

Add the information of an Agent to the registry.

Args:

agent (Agent): Instance of Agent representing the Agent.

Returns:

None

"""

def retrieve_swarm_information(self, swarm_id: str):

"""

Retrieve the information of a specific Swarm from the registry.

Args:

swarm_id (str): Unique identifier of the Swarm.

Returns:

SwarmManagerBase: Instance of SwarmManagerBase representing the retrieved Swarm, or None if not found.

"""

def retrieve_joined_agents(self, agent_id: str) -> List[Agent]:

"""

Retrieve the information the Agents which have joined the registry.

Returns:

```
            Agent: Instance of Agent representing the retrieved Agent, or None if not found.
        """

    def join_swarm(
        self, from_entity: Agent | Agent, to_entity: Agent
    ):
        """
        Add a relationship between a Swarm and an Agent or other Swarm to the registry.

        Args:
            from (Agent | SwarmManagerBase): Instance of Agent or SwarmManagerBase representing
the source of the relationship.
        """

    def metadata(self):
        """
        Get the metadata of the multi-agent structure.

        Returns:
            dict: The metadata of the multi-agent structure.
        """
        return {
            "agents": self.agents,
            "callbacks": self.callbacks,
            "autosave": self.autosave,
            "logging": self.logging,
```

```
        "conversation": self.conversation,
    }

def save_to_json(self, filename: str):
    """
    Save the current state of the multi-agent structure to a JSON file.

    Args:
        filename (str): The name of the file to save the multi-agent structure to.

    Returns:
        None
    """
    try:
        with open(filename, "w") as f:
            json.dump(self.__dict__, f)
    except Exception as e:
        logger.error(e)

def load_from_json(self, filename: str):
    """
    Load the state of the multi-agent structure from a JSON file.

    Args:
        filename (str): The name of the file to load the multi-agent structure from.
```

```
    Returns:
        None
    """
    try:
        with open(filename) as f:
            self.__dict__ = json.load(f)
    except Exception as e:
        logger.error(e)

def save_to_yaml(self, filename: str):
    """
    Save the current state of the multi-agent structure to a YAML file.

    Args:
        filename (str): The name of the file to save the multi-agent structure to.

    Returns:
        None
    """
    try:
        with open(filename, "w") as f:
            yaml.dump(self.__dict__, f)
    except Exception as e:
        logger.error(e)

def load_from_yaml(self, filename: str):
```

```
    """
    Load the state of the multi-agent structure from a YAML file.

    Args:
        filename (str): The name of the file to load the multi-agent structure from.

    Returns:
        None
    """
    try:
        with open(filename) as f:
            self.__dict__ = yaml.load(f)
    except Exception as e:
        logger.error(e)

def __repr__(self):
    return f"{self.__class__.__name__}({self.__dict__})"

def __str__(self):
    return f"{self.__class__.__name__}({self.__dict__})"

def __len__(self):
    return len(self.agents)

def __getitem__(self, index):
    return self.agents[index]
```

```
def __setitem__(self, index, value):
    self.agents[index] = value

def __delitem__(self, index):
    del self.agents[index]

def __iter__(self):
    return iter(self.agents)

def __reversed__(self):
    return reversed(self.agents)

def __contains__(self, value):
    return value in self.agents

def agent_error_handling_check(self):
    try:
        if self.agents is None:
            message = "You have not passed in any agents, you need to input agents to run a swarm"
            logger.info(message)
            raise ValueError(message)
    except Exception as error:
        logger.info(error)
        raise error
```

```
def swarm_initialization(self, *args, **kwargs):
    """
    Initializes the hierarchical swarm.

    Args:
        *args: Additional positional arguments.
        **kwargs: Additional keyword arguments.

    Returns:
        None

    """
    logger.info(
        f"Initializing the hierarchical swarm: {self.name}"
    )
    logger.info(f"Purpose of this swarm: {self.description}")

    # Now log number of agnets and their names
    logger.info(f"Number of agents: {len(self.agents)}")
    logger.info(
        f"Agent names: {[agent.name for agent in self.agents]}"
    )

    # Now see if agents is not empty
    if len(self.agents) == 0:
        logger.info(
```

```
            "No agents found. Please add agents to the swarm."
        )
        return None

    # Now see if director is not empty
    if self.director is None:
        logger.info(
            "No director found. Please add a director to the swarm."
        )
        return None

    logger.info(
        f"Initialization complete for the hierarchical swarm: {self.name}"
    )

def export_output_schema(self):
    """
    Export the output schema of the swarm.

    Returns:
        dict: The output schema of the swarm.

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

def export_output_schema_dict(self):
```

```
    return self.output_schema.model_dump()

def export_and_autosave(self):
    content = self.export_output_schema()

    create_file_in_folder(
        os.getenv("WORKSPACE_DIR"),
        self.metadata_filename,
        content=content,
    )

    return logger.info(
        f"Metadata saved to {self.metadata_filename}"
    )

def list_agents(self):
    """
    List all agents in the swarm.

    Returns:
        None
    """
    display_agents_info(self.agents)

def agents_to_dataframe(self):
    """
```

```
    Convert agents to a pandas DataFrame.
    """
    data = [agent.agent_output.dict() for agent in self.agents]
    return dict_to_dataframe(data)


def model_to_dataframe(self):
    """
    Convert the Pydantic model to a pandas DataFrame.
    """
    return pydantic_model_to_dataframe(self.output_schema)
```