```python
import asyncio
import math
from typing import List, Union

from pydantic import BaseModel

from swarms.structs.agent import Agent
from swarms.structs.omni_agent_types import AgentListType
from swarms.utils.loguru_logger import initialize_logger

logger = initialize_logger(log_folder="swarming_architectures")


# Define Pydantic schema for logging agent responses
class AgentLog(BaseModel):
    agent_name: str
    task: str
    response: str


class Conversation(BaseModel):
    logs: List[AgentLog] = []

    def add_log(
        self, agent_name: str, task: str, response: str
    ) -> None:
```

```python
        log_entry = AgentLog(
            agent_name=agent_name, task=task, response=response
        )
        self.logs.append(log_entry)
        logger.info(
            f"Agent: {agent_name} | Task: {task} | Response: {response}"
        )

    def return_history(self) -> dict:
        return {
            "history": [
                {
                    "agent_name": log.agent_name,
                    "task": log.task,
                    "response": log.response,
                }
                for log in self.logs
            ]
        }


def circular_swarm(
    agents: AgentListType,
    tasks: List[str],
    return_full_history: bool = True,
) -> Union[dict, List[str]]:
```

```
    """
    Implements a circular swarm where agents pass tasks in a circular manner.

    Args:
    - agents (AgentListType): A list of Agent objects to participate in the swarm.
    - tasks (List[str]): A list of tasks to be processed by the agents.
    - return_full_history (bool, optional): If True, returns the full conversation history. Defaults to True.

    Returns:
    - Union[dict, List[str]]: If return_full_history is True, returns a dictionary containing the conversation
history. Otherwise, returns a list of responses.
    """
    # Ensure agents is a flat list of Agent objects
    flat_agents = (
        [agent for sublist in agents for agent in sublist]
        if isinstance(agents[0], list)
        else agents
    )

    if not flat_agents or not tasks:
        raise ValueError("Agents and tasks lists cannot be empty.")

    conversation = Conversation()
    responses = []

    for task in tasks:
```

```
        for agent in flat_agents:
            response = agent.run(task)
            conversation.add_log(
                agent_name=agent.agent_name,
                task=task,
                response=response,
            )
            responses.append(response)

    if return_full_history:
        return conversation.return_history()
    else:
        return responses


def grid_swarm(agents: AgentListType, tasks: List[str]):
    grid_size = int(
        len(agents) ** 0.5
    )  # Assuming agents can form a perfect square grid
    for i in range(grid_size):
        for j in range(grid_size):
            if tasks:
                task = tasks.pop(0)
                agents[i * grid_size + j].run(task)
```

```
# Linear Swarm: Agents process tasks in a sequential linear manner
def linear_swarm(
    agents: AgentListType,
    tasks: List[str],
    return_full_history: bool = True,
) -> Union[str, List[str]]:
    if not agents or not tasks:
        raise ValueError("Agents and tasks lists cannot be empty.")

    conversation = Conversation()
    responses = []

    for agent in agents:
        if tasks:
            task = tasks.pop(0)
            response = agent.run(task)
            conversation.add_log(
                agent_name=agent.agent_name,
                task=task,
                response=response,
            )
            responses.append(response)

    return (
        conversation.return_history()
        if return_full_history
```

```
        else responses
    )



# Star Swarm: A central agent first processes all tasks, followed by others
def star_swarm(
    agents: AgentListType,
    tasks: List[str],
    return_full_history: bool = True,
) -> Union[str, List[str]]:
    if not agents or not tasks:
        raise ValueError("Agents and tasks lists cannot be empty.")

    conversation = Conversation()
    center_agent = agents[0]  # The central agent
    responses = []

    for task in tasks:
        # Central agent processes the task
        center_response = center_agent.run(task)
        conversation.add_log(
            agent_name=center_agent.agent_name,
            task=task,
            response=center_response,
        )
        responses.append(center_response)
```

```python
        # Other agents process the same task
        for agent in agents[1:]:
            response = agent.run(task)
            conversation.add_log(
                agent_name=agent.agent_name,
                task=task,
                response=response,
            )
            responses.append(response)

    return (
        conversation.return_history()
        if return_full_history
        else responses
    )


# Mesh Swarm: Agents work on tasks randomly from a task queue until all tasks are processed
def mesh_swarm(
    agents: AgentListType,
    tasks: List[str],
    return_full_history: bool = True,
) -> Union[str, List[str]]:
    if not agents or not tasks:
        raise ValueError("Agents and tasks lists cannot be empty.")
```

```
    conversation = Conversation()
    task_queue = tasks.copy()
    responses = []

    while task_queue:
        for agent in agents:
            if task_queue:
                task = task_queue.pop(0)
                response = agent.run(task)
                conversation.add_log(
                    agent_name=agent.agent_name,
                    task=task,
                    response=response,
                )
                responses.append(response)

    return (
        conversation.return_history()
        if return_full_history
        else responses
    )


# Pyramid Swarm: Agents are arranged in a pyramid structure
def pyramid_swarm(
```

```
    agents: AgentListType,
    tasks: List[str],
    return_full_history: bool = True,
) -> Union[str, List[str]]:
    if not agents or not tasks:
        raise ValueError("Agents and tasks lists cannot be empty.")

    conversation = Conversation()
    responses = []

    levels = int(
        (-1 + (1 + 8 * len(agents)) ** 0.5) / 2
    )  # Number of levels in the pyramid

    for i in range(levels):
        for j in range(i + 1):
            if tasks:
                task = tasks.pop(0)
                agent_index = int(i * (i + 1) / 2 + j)
                response = agents[agent_index].run(task)
                conversation.add_log(
                    agent_name=agents[agent_index].agent_name,
                    task=task,
                    response=response,
                )
                responses.append(response)
```

```
    return (
        conversation.return_history()
        if return_full_history
        else responses
    )


def fibonacci_swarm(agents: AgentListType, tasks: List[str]):
    fib = [1, 1]
    while len(fib) < len(agents):
        fib.append(fib[-1] + fib[-2])
    for i in range(len(fib)):
        for j in range(fib[i]):
            if tasks:
                task = tasks.pop(0)
                agents[int(sum(fib[:i]) + j)].run(task)


def prime_swarm(agents: AgentListType, tasks: List[str]):
    primes = [
        2,
        3,
        5,
        7,
        11,
```

```
    13,
    17,
    19,
    23,
    29,
    31,
    37,
    41,
    43,
    47,
    53,
    59,
    61,
    67,
    71,
    73,
    79,
    83,
    89,
    97,
]  # First 25 prime numbers
for prime in primes:
    if prime < len(agents) and tasks:
        task = tasks.pop(0)
        agents[prime].run(task)
```

```
def power_swarm(agents: List[str], tasks: List[str]):
    powers = [2**i for i in range(int(len(agents) ** 0.5))]
    for power in powers:
        if power < len(agents) and tasks:
            task = tasks.pop(0)
            agents[power].run(task)


def log_swarm(agents: AgentListType, tasks: List[str]):
    for i in range(len(agents)):
        if 2**i < len(agents) and tasks:
            task = tasks.pop(0)
            agents[2**i].run(task)


def exponential_swarm(agents: AgentListType, tasks: List[str]):
    for i in range(len(agents)):
        index = min(int(2**i), len(agents) - 1)
        if tasks:
            task = tasks.pop(0)
            agents[index].run(task)


def geometric_swarm(agents, tasks):
    ratio = 2
```

```python
    for i in range(range(len(agents))):
        index = min(int(ratio**2), len(agents) - 1)
        if tasks:
            task = tasks.pop(0)
            agents[index].run(task)


def harmonic_swarm(agents: AgentListType, tasks: List[str]):
    for i in range(1, len(agents) + 1):
        index = min(int(len(agents) / i), len(agents) - 1)
        if tasks:
            task = tasks.pop(0)
            agents[index].run(task)


def staircase_swarm(agents: AgentListType, task: str):
    step = len(agents) // 5
    for i in range(len(agents)):
        index = (i // step) * step
        agents[index].run(task)


def sigmoid_swarm(agents: AgentListType, task: str):
    for i in range(len(agents)):
        index = int(len(agents) / (1 + math.exp(-i)))
        agents[index].run(task)
```

```python
def sinusoidal_swarm(agents: AgentListType, task: str):
    for i in range(len(agents)):
        index = int((math.sin(i) + 1) / 2 * len(agents))
        agents[index].run(task)



async def one_to_three(
    sender: Agent, agents: AgentListType, task: str
):
    """
    Sends a message from the sender agent to three other agents.

    Args:
        sender (Agent): The agent sending the message.
        agents (AgentListType): The list of agents to receive the message.
        task (str): The message to be sent.

    Raises:
        Exception: If there is an error while sending the message.

    Returns:
        None
    """
    if len(agents) != 3:
```

```
        raise ValueError("The number of agents must be exactly 3.")

    if not task:
        raise ValueError("The task cannot be empty.")

    if not sender:
        raise ValueError("The sender cannot be empty.")

    try:
        receive_tasks = []
        for agent in agents:
            receive_tasks.append(
                agent.receive_message(sender.agent_name, task)
            )

        await asyncio.gather(*receive_tasks)
    except Exception as error:
        logger.error(
            f"[ERROR][CLASS: Agent][METHOD: one_to_three] {error}"
        )
        raise error


"""
This module contains functions for facilitating communication between agents in a swarm. It includes
methods for one-to-one communication, broadcasting, and other swarm architectures.
```

```
"""

# One-to-One Communication between two agents
def one_to_one(
    sender: Agent, receiver: Agent, task: str, max_loops: int = 1
) -> str:
    """
    Facilitates one-to-one communication between two agents. The sender and receiver agents exchange messages for a specified number of loops.

    Args:
        sender (Agent): The agent sending the message.
        receiver (Agent): The agent receiving the message.
        task (str): The message to be sent.
        max_loops (int, optional): The number of times the sender and receiver exchange messages. Defaults to 1.

    Returns:
        str: The conversation history between the sender and receiver.

    Raises:
        Exception: If there is an error during the communication process.
    """
    conversation = Conversation()
    responses = []
```

```
try:
    for _ in range(max_loops):
        # Sender processes the task
        sender_response = sender.run(task)
        conversation.add_log(
            agent_name=sender.agent_name,
            task=task,
            response=sender_response,
        )
        responses.append(sender_response)

        # Receiver processes the result of the sender
        receiver_response = receiver.run(sender_response)
        conversation.add_log(
            agent_name=receiver.agent_name,
            task=task,
            response=receiver_response,
        )
        responses.append(receiver_response)

except Exception as error:
    logger.error(
        f"Error during one_to_one communication: {error}"
    )
    raise error
```

```
    return conversation.return_history()


# Broadcasting: A message from one agent to many
async def broadcast(
    sender: Agent, agents: AgentListType, task: str
) -> None:
    """
    Facilitates broadcasting of a message from one agent to multiple agents.

    Args:
        sender (Agent): The agent sending the message.
        agents (AgentListType): The list of agents to receive the message.
        task (str): The message to be sent.

    Raises:
        ValueError: If the sender, agents, or task is empty.
        Exception: If there is an error during the broadcasting process.
    """
    conversation = Conversation()

    if not sender or not agents or not task:
        raise ValueError("Sender, agents, and task cannot be empty.")

    try:
```

```
    receive_tasks = []
    for agent in agents:
        receive_tasks.append(agent.run(task))
        conversation.add_log(
            agent_name=agent.agent_name, task=task, response=task
        )

    await asyncio.gather(*receive_tasks)
except Exception as error:
    logger.error(f"Error during broadcast: {error}")
    raise error
```