MultiProcessWorkflow Documentation

The `MultiProcessWorkflow` class provides a framework for executing tasks concurrently using multiple processes. This class leverages Python's `multiprocessing` module to parallelize task execution, thereby enhancing performance and efficiency. It includes features such as automatic task retry on failure and optional autosaving of results. This documentation details the class, its parameters, attributes, methods, and usage examples.

Class Definition

`MultiProcessWorkflow`

Parameters

Parameter	Type	Default	Description
`max_workers`	`int`	`5`	The maximum number of workers to use for parallel processing.
`autosave`	`bool`	`True`	Flag indicating whether to automatically save the workflow.
`agents`	`Sequence[Agent]`	`None`	A list of Agent objects representing the workflow agents.
`*args`	`tuple`		Additional positional arguments.
`**kwargs`	`dict`		Additional keyword arguments.

Attributes

Attribute	Type	Description
`max_workers`	`int`	The maximum number of workers to use for parallel processing.
`autosave`	`bool`	Flag indicating whether to automatically save the workflow.
`agents`	`Sequence[Agent]`	A list of Agent objects representing the workflow agents.

Methods

`execute_task`

Description

The `execute_task` method executes a given task and handles any exceptions that may occur during execution. If agents are defined, it will execute the task using each agent in sequence.

Usage Example

```python
# Define a task
task = Task()

# Execute the task
workflow = MultiProcessWorkflow()
```

result = workflow.execute_task(task)

print(result)

```

### `run`

#### Description

The `run` method executes the workflow by running the given task using multiple processes. It manages the task execution using a process pool and collects the results.

#### Usage Example

```python
from swarms.structs.multi_process_workflow import MultiProcessingWorkflow
from swarms.structs.task import Task
from datetime import datetime
from time import sleep

# Define a simple task
def simple_task():
    sleep(1)
    return datetime.now()

# Create a task object
task = Task(

```
    name="Simple Task",
    execute=simple_task,
    priority=1,
)

# Create a workflow with the task
workflow = MultiProcessWorkflow(max_workers=3, autosave=True, agents=[agent1, agent2])

# Run the workflow
results = workflow.run(task)

# Print the results
print(results)
```

Detailed Functionality and Usage

Initialization

When an instance of `MultiProcessWorkflow` is created, it initializes the following:

- **max_workers**: Sets the maximum number of processes that can run concurrently.
- **autosave**: Determines if the workflow results should be saved automatically.
- **agents**: Accepts a list of agents that will perform the tasks.

Running Tasks

The `run` method performs the following steps:

1. **Initialize Results and Manager**: Creates a list to store results and a `Manager` to manage shared state between processes.
2. **Initialize Process Pool**: Creates a pool of worker processes.
3. **Submit Tasks**: Iterates over the agents, submitting tasks to the pool for execution and collecting the results.
4. **Wait for Completion**: Waits for all tasks to complete and collects the results.
5. **Return Results**: Returns the list of results from all executed tasks.

Autosave Task Result

Although the autosave functionality is mentioned in the parameters, it is not explicitly defined in the given code. The implementation for autosaving should be added based on the specific requirements of the application.

Additional Information and Tips

- **Process Safety**: The use of `Manager` ensures that the list of results is managed safely across multiple processes.
- **Logging**: The class uses the `logger` module to log information about task execution, retries, and failures.
- **Error Handling**: The retry mechanism in the `execute_task` method helps in handling transient errors by attempting to re-execute failed tasks.

References and Resources

For more information on multiprocessing in Python, refer to the following resources:

- [Python Multiprocessing Documentation](https://docs.python.org/3/library/multiprocessing.html)
- [Python Logging Documentation](https://docs.python.org/3/library/logging.html)

By following this detailed documentation, users can effectively understand and utilize the `MultiProcessWorkflow` class to execute tasks concurrently with multiple processes. The examples provided help in demonstrating the practical usage of the class.