# MultiThreadedWorkflow Documentation

The `MultiThreadedWorkflow` class represents a multi-threaded workflow designed to execute tasks concurrently using a thread pool. This class is highly useful in scenarios where tasks need to be executed in parallel to improve performance and efficiency. The workflow ensures that tasks are managed in a priority-based queue, and it includes mechanisms for retrying failed tasks and optionally saving task results automatically.

## Class Definition

### `MultiThreadedWorkflow`

## Parameters

Parameter	Type	Default	Description
`max_workers`	`int`	`5`	The maximum number of worker threads in the thread pool.
`autosave`	`bool`	`True`	Flag indicating whether to automatically save task results.
`tasks`	`List[PriorityTask]`	`None`	List of priority tasks to be executed.
`retry_attempts`	`int`	`3`	The maximum number of retry attempts for failed tasks.
`*args`	`tuple`		Variable length argument list.
`**kwargs`	`dict`		Arbitrary keyword arguments.

## Attributes

Attribute	Type	Description
`max_workers`	`int`	The maximum number of worker threads in the thread pool.
`autosave`	`bool`	Flag indicating whether to automatically save task results.
`retry_attempts`	`int`	The maximum number of retry attempts for failed tasks.
`tasks_queue`	`PriorityQueue`	The queue that holds the priority tasks.
`lock`	`Lock`	The lock used for thread synchronization.

## Methods

### `run`

#### Description

The `run` method executes the tasks stored in the priority queue using a thread pool. It handles task completion, retries failed tasks up to a specified number of attempts, and optionally saves the results of tasks if the autosave flag is set.

#### Usage Example

```python
from swarms import MultiThreadedWorkflow, PriorityTask, Task
```

# Define some tasks

tasks = [PriorityTask(task=Task()), PriorityTask(task=Task())]

# Create a MultiThreadedWorkflow instance

workflow = MultiThreadedWorkflow(max_workers=3, autosave=True, tasks=tasks, retry_attempts=2)

# Run the workflow

results = workflow.run()

print(results)

~~~

`_autosave_task_result`

Description

The `_autosave_task_result` method is responsible for saving the results of a task. It uses a thread lock to ensure that the autosave operation is thread-safe.

Usage Example

This method is intended for internal use and is typically called by the `run` method. However, here is an example of how it might be used directly:

```python

# Create a task and result
~~~

```
task = Task()
result = task.run()

Autosave the result
workflow = MultiThreadedWorkflow()
workflow._autosave_task_result(task, result)
```

## Detailed Functionality and Usage

### Initialization

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

- **max_workers**: Sets the maximum number of threads that can run concurrently.
- **autosave**: Determines if the task results should be saved automatically.
- **tasks**: Accepts a list of tasks that need to be executed. If no tasks are provided, an empty list is used.
- **retry_attempts**: Sets the maximum number of retry attempts for failed tasks.
- **tasks_queue**: A priority queue to manage tasks based on their priority.
- **lock**: A threading lock to ensure thread-safe operations.

### Running Tasks

The `run` method performs the following steps:

1. **Initialize Results and Executor**: Creates a list to store results and a `ThreadPoolExecutor` to manage the threads.
2. **Submit Tasks**: Iterates over the tasks in the queue, submitting them to the executor for execution and storing the future objects.
3. **Monitor Completion**: Uses the `wait` function to monitor the completion of tasks. Once a task is completed, it retrieves the result or catches exceptions.
4. **Retry Mechanism**: If a task fails, it checks the number of attempts made and retries the task if the limit is not reached.
5. **Autosave**: If the `autosave` flag is set, the `_autosave_task_result` method is called to save the task results.

### Autosave Task Result

The `_autosave_task_result` method handles the saving of task results. It uses a threading lock to ensure that the save operation is not interrupted by other threads.

## Additional Information and Tips

- **Thread Safety**: The use of threading locks ensures that the operations are thread-safe, preventing race conditions.
- **Logging**: The class uses the logging module to log information about task completion, retries, and failures.
- **Error Handling**: The retry mechanism helps in handling transient errors by attempting to re-execute failed tasks.

## References and Resources

For more information on threading and concurrent execution in Python, refer to the following resources:

- [Python Threading Documentation](https://docs.python.org/3/library/threading.html)
- [Python Concurrent Futures Documentation](https://docs.python.org/3/library/concurrent.futures.html)