```
import logging
import os
import queue
import threading
import time
from concurrent.futures import ThreadPoolExecutor, as_completed
from typing import List
from swarms.utils.calculate_func_metrics import profile_func
from pydantic import BaseModel

from swarms import Agent
from swarm_models import OpenAIChat
from swarms.prompts.finance_agent_sys_prompt import (
    FINANCIAL_AGENT_SYS_PROMPT,
)
from swarms.structs.agent import Agent
from swarms.structs.base_swarm import BaseSwarm
from swarms.structs.queue_swarm import TaskQueueSwarm


class AgentOutput(BaseModel):
    agent_name: str
    task: str
    result: str
    timestamp: str
```

```
class SwarmRunMetadata(BaseModel):
    run_id: str
    name: str
    description: str
    agents: List[str]
    start_time: str
    end_time: str
    tasks_completed: int
    outputs: List[AgentOutput]



class TaskQueueSwarm(BaseSwarm):
    def __init__(
        self,
        agents: List[Agent],
        name: str = "Task-Queue-Swarm",
            description: str = "A swarm that processes tasks from a queue using multiple agents on different threads.",
        autosave_on: bool = True,
        save_file_path: str = "swarm_run_metadata.json",
        workspace_dir: str = None,
        return_metadata_on: bool = False,
        max_loops: int = 1,
        max_threads: int = None,  # Control the max number of threads
        retry_attempts: int = 3,  # Retry failed tasks
```

```
    *args,
    **kwargs,
):
    super().__init__(
        name=name,
        description=description,
        agents=agents,
        *args,
        **kwargs,
    )
    self.agents = agents
    self.task_queue = queue.Queue()
    self.lock = threading.Lock()
    self.autosave_on = autosave_on
    self.save_file_path = save_file_path
    self.workspace_dir = workspace_dir or os.getenv(
        "WORKSPACE_DIR", "."
    )
    self.return_metadata_on = return_metadata_on
    self.max_loops = max_loops
    self.max_threads = (
        os.cpu_count()
    )  # Default to 2 threads per agent
    self.retry_attempts = retry_attempts
    self.logger = logging.getLogger(__name__)
```

```
    current_time = time.strftime("%Y%m%d%H%M%S")
    self.metadata = SwarmRunMetadata(
        run_id=f"swarm_run_{current_time}",
        name=name,
        description=description,
        agents=[agent.agent_name for agent in agents],
        start_time=current_time,
        end_time="",
        tasks_completed=0,
        outputs=[],
    )

    # Initialize ThreadPoolExecutor
    self.executor = ThreadPoolExecutor(
        max_workers=self.max_threads
    )

def reliability_checks(self):
    self.logger.info("Initializing reliability checks.")

    if not self.agents:
        raise ValueError(
            "You must provide a non-empty list of Agent instances."
        )

    if self.max_loops <= 0:
```

```
        raise ValueError("max_loops must be greater than zero.")

    self.logger.info(
        "Reliability checks successful. Swarm is ready for usage."
    )

def add_task(self, task: str):
    """Adds a task to the queue."""
    self.task_queue.put(task)

def _process_task(self, agent: Agent):
    """Processes tasks from the queue using the provided agent."""
    while not self.task_queue.empty():
        task = self.task_queue.get_nowait()
        attempt = 0
        success = False
        while attempt < self.retry_attempts and not success:
            try:
                self.logger.info(
                    f"Agent {agent.agent_name} is running task: {task}"
                )
                result = agent.run(task)
                with self.lock:
                    self.metadata.tasks_completed += 1
                    self.metadata.outputs.append(
                        AgentOutput(
```

```
                agent_name=agent.agent_name,
                task=task,
                result=result,
                timestamp=time.strftime(
                    "%Y-%m-%d %H:%M:%S"
                ),
            )
        )
    self.logger.info(
        f"Agent {agent.agent_name} completed task: {task}"
    )
    self.logger.debug(f"Result: {result}")
    success = True
except Exception as e:
    attempt += 1
    self.logger.error(
        f"Attempt {attempt} failed for task: {task}"
    )
    self.logger.exception(e)
    if attempt >= self.retry_attempts:
        self.logger.error(
            f"Task failed after {self.retry_attempts} attempts: {task}"
        )
finally:
    self.task_queue.task_done()
```

```
def run(self):
    """Runs the swarm by having agents pick up tasks from the queue."""
    self.logger.info(
        f"Starting swarm run: {self.metadata.run_id}"
    )

    futures = [
        self.executor.submit(self._process_task, agent)
        for agent in self.agents
    ]

    for future in as_completed(futures):
        try:
            future.result()
        except Exception as e:
            self.logger.exception(
                f"Task processing raised an exception: {e}"
            )

    self.executor.shutdown(wait=True)
    self.metadata.end_time = time.strftime("%Y%m%d%H%M%S")

    if self.autosave_on:
        self.save_json_to_file()

    if self.return_metadata_on:
```

```
            return self.metadata

    def save_json_to_file(self):
        json_string = self.export_metadata()
        file_path = os.path.join(
            self.workspace_dir, self.save_file_path
        )
        os.makedirs(os.path.dirname(file_path), exist_ok=True)
        with open(file_path, "w") as f:
            f.write(json_string)
        self.logger.info(f"Metadata saved to {file_path}")

    def export_metadata(self):
        return self.metadata.model_dump_json(indent=4)



# Example usage:
api_key = os.getenv("OPENAI_API_KEY")

# Model
model = OpenAIChat(
    openai_api_key=api_key, model_name="gpt-4o-mini", temperature=0.1
)



# Initialize your agents (assuming the Agent class and model are already defined)
```

```
agents = [
    Agent(
        agent_name=f"Financial-Analysis-Agent-Task-Queue-swarm-{i}",
        system_prompt=FINANCIAL_AGENT_SYS_PROMPT,
        llm=model,
        max_loops=1,
        autosave=True,
        dashboard=False,
        verbose=True,
        dynamic_temperature_enabled=True,
        saved_state_path="finance_agent.json",
        user_name="swarms_corp",
        retry_attempts=1,
        context_length=200000,
        return_step_meta=False,
    )
    for i in range(2)
]
# Create a Swarm with the list of agents
swarm = TaskQueueSwarm(
    agents=agents,
    return_metadata_on=True,
    autosave_on=True,
    save_file_path="swarm_run_metadata.json",
)
```

```
@profile_func
def execute_task_queue_swarm():
    # Add tasks to the swarm
    swarm.add_task(
        "How can I establish a ROTH IRA to buy stocks and get a tax break? What are the criteria?"
    )
    swarm.add_task(
        "Analyze the financial risks of investing in tech stocks."
    )

    # Keep adding tasks as needed...
    # swarm.add_task("...")

    # Run the swarm and get the output
    out = swarm.run()

    # Print the output
    print(out)

    # Export the swarm metadata
    swarm.export_metadata()


execute_task_queue_swarm()
# 2024-08-27T14:07:07.805850-0400 Function metrics: {
```

```
#     "execution_time": 11.02876901626587,
#     "memory_usage": -439.421875,
#     "cpu_usage": -7.600000000000001,
#     "io_operations": 4552,
#     "function_calls": 1
# }\
#     2024-08-27T14:56:03.613112-0400 Function metrics: {
#     "execution_time": 7.604920864105225,
#     "memory_usage": -416.5625,
#     "cpu_usage": -5.300000000000004,
#     "io_operations": 7270,
#     "function_calls": 1
# }
```