```
import queue
import threading
from typing import List
from swarms.structs.agent import Agent
from pydantic import BaseModel
import os
from swarms.utils.loguru_logger import logger
from swarms.structs.base_swarm import BaseSwarm
import time


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):
    """
    A swarm that processes tasks from a queue using multiple agents on different threads.

    Args:
        agents (List[Agent]): A list of agents of class Agent.
        name (str, optional): The name of the swarm. Defaults to "Task-Queue-Swarm".
        description (str, optional): The description of the swarm. Defaults to "A swarm that processes tasks from a queue using multiple agents on different threads.".
        autosave_on (bool, optional): Whether to automatically save the swarm metadata. Defaults to True.
        save_file_path (str, optional): The file path to save the swarm metadata. Defaults to "swarm_run_metadata.json".
        workspace_dir (str, optional): The directory path of the workspace. Defaults to os.getenv("WORKSPACE_DIR").
        return_metadata_on (bool, optional): Whether to return the swarm metadata after running. Defaults to False.
        max_loops (int, optional): The maximum number of loops to run the swarm. Defaults to 1.

    Attributes:
        agents (List[Agent]): A list of agents of class Agent.
        task_queue (queue.Queue): A queue to store the tasks.
        lock (threading.Lock): A lock for thread synchronization.
        autosave_on (bool): Whether to automatically save the swarm metadata.
        save_file_path (str): The file path to save the swarm metadata.
        workspace_dir (str): The directory path of the workspace.
        return_metadata_on (bool): Whether to return the swarm metadata after running.
        max_loops (int): The maximum number of loops to run the swarm.
        metadata (SwarmRunMetadata): The metadata of the swarm run.
    """


    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 = os.getenv("WORKSPACE_DIR"),
        return_metadata_on: bool = False,
        max_loops: int = 1,
        *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", "agent_workspace"
        )
        self.return_metadata_on = return_metadata_on
        self.max_loops = max_loops

        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=[],
        )
```

```
def reliability_checks(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.")

    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 True:
        try:
            task = self.task_queue.get_nowait()
        except queue.Empty:
            break
```

```
try:
    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"
                ),
            )
        )
    logger.info(
        f"Agent {agent.agent_name} completed task: {task}"
    )
    logger.debug(f"Result: {result}")
except Exception as e:
    logger.error(
        f"Agent {agent.agent_name} failed to complete task: {task}"
    )
    logger.exception(e)
```

```
        finally:
            self.task_queue.task_done()

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

        threads = [
            threading.Thread(
                target=self._process_task, args=(agent,), daemon=True
            )
            for agent in self.agents
        ]

        for thread in threads:
            thread.start()

        self.task_queue.join()

        for thread in threads:
            thread.join()

        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.model_dump_json(indent=4)
        return self.export_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)
        logger.info(f"Metadata saved to {file_path}")

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