```
import asyncio

import os

import uuid

from concurrent.futures import ThreadPoolExecutor

from datetime import datetime

from typing import Any, Dict, List, Optional, Union


from pydantic import BaseModel, Field

from tenacity import retry, stop_after_attempt, wait_exponential


from swarms.structs.agent import Agent

from swarms.structs.base_swarm import BaseSwarm

from swarms.utils.file_processing import create_file_in_folder

import concurrent

from clusterops import (

    execute_on_gpu,

    execute_with_cpu_cores,

    execute_on_multiple_gpus,

    list_available_gpus,

)

from swarms.utils.loguru_logger import initialize_logger

from swarms.structs.swarm_id_generator import generate_swarm_id


logger = initialize_logger(log_folder="concurrent_workflow")
```

```
class AgentOutputSchema(BaseModel):

    run_id: Optional[str] = Field(

        ..., description="Unique ID for the run"

    )

    agent_name: Optional[str] = Field(

        ..., description="Name of the agent"

    )

    task: Optional[str] = Field(

        ..., description="Task or query given to the agent"

    )

    output: Optional[str] = Field(

        ..., description="Output generated by the agent"

    )

    start_time: Optional[datetime] = Field(

        ..., description="Start time of the task"

    )

    end_time: Optional[datetime] = Field(

        ..., description="End time of the task"

    )

    duration: Optional[float] = Field(

        ...,

        description="Duration taken to complete the task (in seconds)",

    )



class MetadataSchema(BaseModel):
    swarm_id: Optional[str] = Field(
        generate_swarm_id(), description="Unique ID for the run"
    )
    task: Optional[str] = Field(
        ..., description="Task or query given to all agents"
    )
    description: Optional[str] = Field(
        "Concurrent execution of multiple agents",
        description="Description of the workflow",
    )
    agents: Optional[List[AgentOutputSchema]] = Field(
        ..., description="List of agent outputs and metadata"
    )
    timestamp: Optional[datetime] = Field(
        default_factory=datetime.now,
        description="Timestamp of the workflow execution",
    )


class ConcurrentWorkflow(BaseSwarm):
    """
    Represents a concurrent workflow that executes multiple agents concurrently in a production-grade manner.

    Args:
        name (str): The name of the workflow. Defaults to "ConcurrentWorkflow".
```

description (str): The description of the workflow. Defaults to "Execution of multiple agents concurrently".

agents (List[Agent]): The list of agents to be executed concurrently. Defaults to an empty list.

metadata_output_path (str): The path to save the metadata output. Defaults to "agent_metadata.json".

auto_save (bool): Flag indicating whether to automatically save the metadata. Defaults to False.

output_schema (BaseModel): The output schema for the metadata. Defaults to MetadataSchema.

max_loops (int): The maximum number of loops for each agent. Defaults to 1.

return_str_on (bool): Flag indicating whether to return the output as a string. Defaults to False.

agent_responses (list): The list of agent responses. Defaults to an empty list.

auto_generate_prompts (bool): Flag indicating whether to auto-generate prompts for agents. Defaults to False.

Raises:

ValueError: If the list of agents is empty or if the description is empty.

Attributes:

name (str): The name of the workflow.

description (str): The description of the workflow.

agents (List[Agent]): The list of agents to be executed concurrently.

metadata_output_path (str): The path to save the metadata output.

auto_save (bool): Flag indicating whether to automatically save the metadata.

output_schema (BaseModel): The output schema for the metadata.

max_loops (int): The maximum number of loops for each agent.

```
    return_str_on (bool): Flag indicating whether to return the output as a string.
    agent_responses (list): The list of agent responses.
    auto_generate_prompts (bool): Flag indicating whether to auto-generate prompts for agents.
    retry_attempts (int): The number of retry attempts for failed agent executions.
    retry_wait_time (int): The initial wait time for retries in seconds.
"""


def __init__(
    self,
    name: str = "ConcurrentWorkflow",
    description: str = "Execution of multiple agents concurrently",
    agents: List[Agent] = [],
    metadata_output_path: str = "agent_metadata.json",
    auto_save: bool = True,
    output_schema: BaseModel = MetadataSchema,
    max_loops: int = 1,
    return_str_on: bool = False,
    agent_responses: list = [],
    auto_generate_prompts: bool = False,
    max_workers: int = None,
    *args,
    **kwargs,
):
    super().__init__(
        name=name,
        description=description,
            agents=agents,
            *args,
            **kwargs,
        )
        self.name = name
        self.description = description
        self.agents = agents
        self.metadata_output_path = metadata_output_path
        self.auto_save = auto_save
        self.output_schema = output_schema
        self.max_loops = max_loops
        self.return_str_on = return_str_on
        self.agent_responses = agent_responses
        self.auto_generate_prompts = auto_generate_prompts
        self.max_workers = max_workers or os.cpu_count()
        self.tasks = []  # Initialize tasks list

        self.reliability_check()

    def reliability_check(self):
        try:
            logger.info("Starting reliability checks")

            if self.name is None:
                logger.error("A name is required for the swarm")
                raise ValueError("A name is required for the swarm")
            if not self.agents:
                logger.error("The list of agents must not be empty.")
                raise ValueError(
                    "The list of agents must not be empty."
                )

            if not self.description:
                logger.error("A description is required.")
                raise ValueError("A description is required.")

            logger.info("Reliability checks completed successfully")
        except ValueError as e:
            logger.error(f"Reliability check failed: {e}")
            raise
        except Exception as e:
            logger.error(
                f"An unexpected error occurred during reliability checks: {e}"
            )
            raise

    def activate_auto_prompt_engineering(self):
        """
        Activates the auto-generate prompts feature for all agents in the workflow.

        Example:
            >>> workflow = ConcurrentWorkflow(agents=[Agent()])
            >>> workflow.activate_auto_prompt_engineering()
            >>> # All agents in the workflow will now auto-generate prompts.
        """
        if self.auto_generate_prompts is True:
            for agent in self.agents:
                agent.auto_generate_prompt = True

    @retry(wait=wait_exponential(min=2), stop=stop_after_attempt(3))
    async def _run_agent(
        self,
        agent: Agent,
        task: str,
        img: str,
        executor: ThreadPoolExecutor,
        *args,
        **kwargs,
    ) -> AgentOutputSchema:
        """
        Runs a single agent with the given task and tracks its output and metadata with retry logic.

        Args:
            agent (Agent): The agent instance to run.
            task (str): The task or query to give to the agent.
            img (str): The image to be processed by the agent.
            executor (ThreadPoolExecutor): The thread pool executor to use for running the agent task.
            Returns:
                AgentOutputSchema: The metadata and output from the agent's execution.

            Example:
                >>> async def run_agent_example():
                >>>     executor = ThreadPoolExecutor()
                    >>>       agent_output = await workflow._run_agent(agent=Agent(), task="Example task",
    img="example.jpg", executor=executor)
                >>>     print(agent_output)
            """
            start_time = datetime.now()
            try:
                loop = asyncio.get_running_loop()
                output = await loop.run_in_executor(
                    executor, agent.run, task, img, *args, **kwargs
                )
            except Exception as e:
                logger.error(
                    f"Error running agent {agent.agent_name}: {e}"
                )
                raise

            end_time = datetime.now()
            duration = (end_time - start_time).total_seconds()
            agent_output = AgentOutputSchema(
                run_id=uuid.uuid4().hex,
                agent_name=agent.agent_name,
                task=task,
                output=output,
                start_time=start_time,
                end_time=end_time,
                duration=duration,
            )

            logger.info(
                f"Agent {agent.agent_name} completed task: {task} in {duration:.2f} seconds."
            )

            return agent_output

        def transform_metadata_schema_to_str(
            self, schema: MetadataSchema
        ):
            """
            Converts the metadata swarm schema into a string format with the agent name, response, and
    time.

            Args:
                schema (MetadataSchema): The metadata schema to convert.
            Returns:
                str: The string representation of the metadata schema.

            Example:
                >>> metadata_schema = MetadataSchema()
                >>> metadata_str = workflow.transform_metadata_schema_to_str(metadata_schema)
                >>> print(metadata_str)
            """
            self.agent_responses = [
                f"Agent Name: {agent.agent_name}\nResponse: {agent.output}\n\n"
                for agent in schema.agents
            ]

            # Return the agent responses as a string
            return "\n".join(self.agent_responses)

        @retry(wait=wait_exponential(min=2), stop=stop_after_attempt(3))
        async def _execute_agents_concurrently(
            self, task: str, img: str, *args, **kwargs
        ) -> MetadataSchema:
            """
              Executes multiple agents concurrently with the same task, incorporating retry logic for failed
    executions.

            Args:
                task (str): The task or query to give to all agents.
                img (str): The image to be processed by the agents.


            Returns:
                MetadataSchema: The aggregated metadata and outputs from all agents.


            Example:
                >>> async def execute_agents_example():
                  >>>     metadata_schema = await workflow._execute_agents_concurrently(task="Example
    task", img="example.jpg")
                >>>    print(metadata_schema)
            """
            with ThreadPoolExecutor(
                max_workers=os.cpu_count()
            ) as executor:
                tasks_to_run = [
                    self._run_agent(
                        agent, task, img, executor, *args, **kwargs
                    )
                    for agent in self.agents
                ]

                agent_outputs = await asyncio.gather(*tasks_to_run)
            return MetadataSchema(
                swarm_id=uuid.uuid4().hex,
                task=task,
                description=self.description,
                agents=agent_outputs,
            )


        def save_metadata(self):
            """
            Saves the metadata to a JSON file based on the auto_save flag.


            Example:
                >>> workflow.save_metadata()
                >>> # Metadata will be saved to the specified path if auto_save is True.
            """
            # Save metadata to a JSON file
            if self.auto_save:
                logger.info(
                    f"Saving metadata to {self.metadata_output_path}"
                )
                create_file_in_folder(
                    os.getenv("WORKSPACE_DIR"),
                    self.metadata_output_path,
                    self.output_schema.model_dump_json(indent=4),
                )


        def _run(
            self, task: str, img: str, *args, **kwargs
        ) -> Union[Dict[str, Any], str]:
            """
```

```
        Runs the workflow for the given task, executes agents concurrently, and saves metadata in a
production-grade manner.

        Args:
            task (str): The task or query to give to all agents.
            img (str): The image to be processed by the agents.

        Returns:
            Dict[str, Any]: The final metadata as a dictionary.
            str: The final metadata as a string if return_str_on is True.

        Example:
            >>> metadata = workflow.run(task="Example task", img="example.jpg")
            >>> print(metadata)
        """
        logger.info(
            f"Running concurrent workflow with {len(self.agents)} agents."
        )
        self.output_schema = asyncio.run(
            self._execute_agents_concurrently(
                task, img, *args, **kwargs
            )
        )

        self.save_metadata()
        if self.return_str_on:
            return self.transform_metadata_schema_to_str(
                self.output_schema
            )

        else:
            # Return metadata as a dictionary
            return self.output_schema.model_dump_json(indent=4)

    def run(
        self,
        task: Optional[str] = None,
        img: Optional[str] = None,
        is_last: bool = False,
        device: str = "cpu",  # gpu
        device_id: int = 0,
        all_cores: bool = True,  # Defaults to using all available cores
        all_gpus: bool = False,
        *args,
        **kwargs,
    ) -> Any:
        """
        Executes the agent's run method on a specified device.
```

This method attempts to execute the agent's run method on a specified device, either CPU or GPU. It logs the device selection and the number of cores or GPU ID used. If the device is set to

```
CPU, it can use all available cores or a specific core specified by `device_id`. If the device is set to
GPU, it uses the GPU specified by `device_id`.

    Args:
        task (Optional[str], optional): The task to be executed. Defaults to None.
        img (Optional[str], optional): The image to be processed. Defaults to None.
        is_last (bool, optional): Indicates if this is the last task. Defaults to False.
        device (str, optional): The device to use for execution. Defaults to "cpu".
        device_id (int, optional): The ID of the GPU to use if device is set to "gpu". Defaults to 0.
        all_cores (bool, optional): If True, uses all available CPU cores. Defaults to True.
        all_gpus (bool, optional): If True, uses all available GPUS. Defaults to True.
        *args: Additional positional arguments to be passed to the execution method.
        **kwargs: Additional keyword arguments to be passed to the execution method.

    Returns:
        Any: The result of the execution.

    Raises:
        ValueError: If an invalid device is specified.
        Exception: If any other error occurs during execution.
    """
    if task is not None:
        self.tasks.append(task)

    try:
        logger.info(f"Attempting to run on device: {device}")
```

```
if device == "cpu":
    logger.info("Device set to CPU")
    if all_cores is True:
        count = os.cpu_count()
        logger.info(
            f"Using all available CPU cores: {count}"
        )
    else:
        count = device_id
        logger.info(f"Using specific CPU core: {count}")

    return execute_with_cpu_cores(
        count, self._run, task, img, *args, **kwargs
    )

elif device == "gpu":
    logger.info("Device set to GPU")
    return execute_on_gpu(
        device_id, self._run, task, img, *args, **kwargs
    )

elif all_gpus is True:
    return execute_on_multiple_gpus(
        [int(gpu) for gpu in list_available_gpus()],
        self._run,
        task,
                img,
                *args,
                **kwargs,
            )
        else:
            raise ValueError(
                f"Invalid device specified: {device}. Supported devices are 'cpu' and 'gpu'."
            )
    except ValueError as e:
        logger.error(f"Invalid device specified: {e}")
        raise e
    except Exception as e:
        logger.error(f"An error occurred during execution: {e}")
        raise e

def run_batched(
    self, tasks: List[str]
) -> List[Union[Dict[str, Any], str]]:
    """
     Runs the workflow for a batch of tasks, executes agents concurrently for each task, and saves
metadata in a production-grade manner.

    Args:
        tasks (List[str]): A list of tasks or queries to give to all agents.

    Returns:
        List[Union[Dict[str, Any], str]]: A list of final metadata for each task, either as a dictionary or a
string.

    Example:
        >>> tasks = ["Task 1", "Task 2"]
        >>> results = workflow.run_batched(tasks)
        >>> print(results)
    """
    results = []
    for task in tasks:
        result = self.run(task)
        results.append(result)
    return results

def run_async(self, task: str) -> asyncio.Future:
    """
    Runs the workflow asynchronously for the given task, executes agents concurrently, and saves
metadata in a production-grade manner.

    Args:
        task (str): The task or query to give to all agents.

    Returns:
        asyncio.Future: A future object representing the asynchronous operation.

    Example:
```

```
        >>> async def run_async_example():
        >>>     future = workflow.run_async(task="Example task")
        >>>     result = await future
        >>>     print(result)
        """
        logger.info(
            f"Running concurrent workflow asynchronously with {len(self.agents)} agents."
        )
        return asyncio.ensure_future(self.run(task))


    def run_batched_async(
        self, tasks: List[str]
    ) -> List[asyncio.Future]:
        """
        Runs the workflow asynchronously for a batch of tasks, executes agents concurrently for each task, and saves metadata in a production-grade manner.


        Args:
            tasks (List[str]): A list of tasks or queries to give to all agents.


        Returns:
            List[asyncio.Future]: A list of future objects representing the asynchronous operations for each task.


        Example:
            >>> tasks = ["Task 1", "Task 2"]
            >>> futures = workflow.run_batched_async(tasks)

            >>> results = await asyncio.gather(*futures)

            >>> print(results)

        """

        futures = []

        for task in tasks:

            future = self.run_async(task)

            futures.append(future)

        return futures


    def run_parallel(

        self, tasks: List[str]

    ) -> List[Union[Dict[str, Any], str]]:

        """

         Runs the workflow in parallel for a batch of tasks, executes agents concurrently for each task,
and saves metadata in a production-grade manner.


        Args:

            tasks (List[str]): A list of tasks or queries to give to all agents.


        Returns:

            List[Union[Dict[str, Any], str]]: A list of final metadata for each task, either as a dictionary or a
string.


        Example:

            >>> tasks = ["Task 1", "Task 2"]
            >>> results = workflow.run_parallel(tasks)
            >>> print(results)
        """
        with ThreadPoolExecutor(
            max_workers=os.cpu_count()
        ) as executor:
            futures = {
                executor.submit(self.run, task): task
                for task in tasks
            }
            results = []
            for future in concurrent.futures.as_completed(futures):
                result = future.result()
                results.append(result)
        return results

    def run_parallel_async(
        self, tasks: List[str]
    ) -> List[asyncio.Future]:
        """
```

Runs the workflow in parallel asynchronously for a batch of tasks, executes agents concurrently for each task, and saves metadata in a production-grade manner.

```
        Args:
            tasks (List[str]): A list of tasks or queries to give to all agents.
        Returns:
                List[asyncio.Future]: A list of future objects representing the asynchronous operations for
each task.

        Example:
            >>> tasks = ["Task 1", "Task 2"]
            >>> futures = workflow.run_parallel_async(tasks)
            >>> results = await asyncio.gather(*futures)
            >>> print(results)
        """
        futures = []
        for task in tasks:
            future = self.run_async(task)
            futures.append(future)
        return futures


# if __name__ == "__main__":
#     # Assuming you've already initialized some agents outside of this class
#     model = OpenAIChat(
#         api_key=os.getenv("OPENAI_API_KEY"),
#         model_name="gpt-4o-mini",
#         temperature=0.1,
#     )
#     agents = [
#         Agent(
#             agent_name=f"Financial-Analysis-Agent-{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=f"finance_agent_{i}.json",
#             user_name="swarms_corp",
#             retry_attempts=1,
#             context_length=200000,
#             return_step_meta=False,
#         )
#         for i in range(3)  # Adjust number of agents as needed
#     ]

#     # Initialize the workflow with the list of agents
#     workflow = ConcurrentWorkflow(
#         agents=agents,
#         metadata_output_path="agent_metadata_4.json",
#         return_str_on=True,
#     )

#     # Define the task for all agents
#       task = "How can I establish a ROTH IRA to buy stocks and get a tax break? What are the
```

criteria?"

```
#     # Run the workflow and save metadata

#     metadata = workflow.run(task)

#     print(metadata)
```