ConcurrentWorkflow Documentation

Overview

The `ConcurrentWorkflow` class is designed to facilitate the concurrent execution of multiple agents, each tasked with solving a specific query or problem. This class is particularly useful in scenarios where multiple agents need to work in parallel, allowing for efficient resource utilization and faster completion of tasks. The workflow manages the execution, collects metadata, and optionally saves the results in a structured format.

Key Features

- **Concurrent Execution**: Runs multiple agents simultaneously using Python's `asyncio` and `ThreadPoolExecutor`.
- **Metadata Collection**: Gathers detailed metadata about each agent's execution, including start and end times, duration, and output.
- **Customizable Output**: Allows the user to save metadata to a file or return it as a string or dictionary.
- **Error Handling**: Implements retry logic for improved reliability.
- **Batch Processing**: Supports running tasks in batches and parallel execution.
- **Asynchronous Execution**: Provides asynchronous run options for improved performance.

Class Definitions

AgentOutputSchema

The `AgentOutputSchema` class is a data model that captures the output and metadata for each agent's execution. It inherits from `pydantic.BaseModel` and provides structured fields to store essential information.

Attribute	Type	Description
`run_id`	`Optional[str]`	Unique ID for the run, automatically generated using `uuid`.
`agent_name`	`Optional[str]`	Name of the agent that executed the task.
`task`	`Optional[str]`	The task or query given to the agent.
`output`	`Optional[str]`	The output generated by the agent.
`start_time`	`Optional[datetime]`	The time when the agent started the task.
`end_time`	`Optional[datetime]`	The time when the agent completed the task.
`duration`	`Optional[float]`	The total time taken to complete the task, in seconds.

MetadataSchema

The `MetadataSchema` class is another data model that aggregates the outputs from all agents involved in the workflow. It also inherits from `pydantic.BaseModel` and includes fields for additional workflow-level metadata.

Attribute	Type	Description
`swarm_id`	`Optional[str]`	Unique ID for the workflow run, generated using `uuid`.
`task`	`Optional[str]`	The task or query given to all agents.
`description`	`Optional[str]`	A description of the workflow, typically indicating concurrent execution.

| `agents` | `Optional[List[AgentOutputSchema]]` | A list of agent outputs and metadata. |
| `timestamp` | `Optional[datetime]` | The timestamp when the workflow was executed. |

ConcurrentWorkflow

The `ConcurrentWorkflow` class is the core class that manages the concurrent execution of agents. It inherits from `BaseSwarm` and includes several key attributes and methods to facilitate this process.

Attributes

Attribute	Type	Description
`name`	`str`	The name of the workflow. Defaults to `"ConcurrentWorkflow"`.
`description`	`str`	A brief description of the workflow.
`agents`	`List[Agent]`	A list of agents to be executed concurrently.
`metadata_output_path`	`str`	Path to save the metadata output. Defaults to `"agent_metadata.json"`.
`auto_save`	`bool`	Flag indicating whether to automatically save the metadata.
`output_schema`	`BaseModel`	The output schema for the metadata, defaults to `MetadataSchema`.
`max_loops`	`int`	Maximum number of loops for the workflow, defaults to `1`.
`return_str_on`	`bool`	Flag to return output as string. Defaults to `False`.
`agent_responses`	`List[str]`	List of agent responses as strings.

| `auto_generate_prompts`| `bool` | Flag indicating whether to auto-generate prompts for agents. |

Methods

ConcurrentWorkflow.__init__

Initializes the `ConcurrentWorkflow` class with the provided parameters.

Parameters

Parameter	Type	Default Value	Description
`name`	`str`	`"ConcurrentWorkflow"`	The name of the workflow.
`description`	`str`	`"Execution of multiple agents concurrently"`	A brief description of the workflow.
`agents`	`List[Agent]`	`[]`	A list of agents to be executed concurrently.
`metadata_output_path`	`str`	`"agent_metadata.json"`	Path to save the metadata output.
`auto_save`	`bool`	`False`	Flag indicating whether to automatically save the metadata.
`output_schema`	`BaseModel`	`MetadataSchema`	The output schema for

the metadata. |

| `max_loops` | `int` | `1` | Maximum number of loops for the workflow. |

| `return_str_on` | `bool` | `False` | Flag to return output as string. |

| `agent_responses` | `List[str]` | `[]` | List of agent responses as strings. |

| `auto_generate_prompts`| `bool` | `False` | Flag indicating whether to auto-generate prompts for agents. |

Raises

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

ConcurrentWorkflow.activate_auto_prompt_engineering

Activates the auto-generate prompts feature for all agents in the workflow.

Example

```python
workflow = ConcurrentWorkflow(agents=[Agent()])
workflow.activate_auto_prompt_engineering()
# All agents in the workflow will now auto-generate prompts.
```

ConcurrentWorkflow._run_agent

Runs a single agent with the provided task and tracks its output and metadata.

Parameters

Parameter	Type	Description
`agent`	`Agent`	The agent instance to run.
`task`	`str`	The task or query to give to 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.

Detailed Explanation

This method handles the execution of a single agent by offloading the task to a thread using `ThreadPoolExecutor`. It also tracks the time taken by the agent to complete the task and logs relevant information. If an exception occurs during execution, it captures the error and includes it in the output. The method implements retry logic for improved reliability.

ConcurrentWorkflow.transform_metadata_schema_to_str

Transforms the metadata schema into a string format.

Parameters

Parameter	Type	Description
`schema`	`MetadataSchema`	The metadata schema to transform.

Returns

- `str`: The metadata schema as a formatted string.

Detailed Explanation

This method converts the metadata stored in `MetadataSchema` into a human-readable string format, particularly focusing on the agent names and their respective outputs. This is useful for quickly reviewing the results of the concurrent workflow in a more accessible format.

ConcurrentWorkflow._execute_agents_concurrently

Executes multiple agents concurrently with the same task.

Parameters

Parameter	Type	Description
`task`	`str`	The task or query to give to all agents.

Returns

- `MetadataSchema`: The aggregated metadata and outputs from all agents.

Detailed Explanation

This method is responsible for managing the concurrent execution of all agents. It uses `asyncio.gather` to run multiple agents simultaneously and collects their outputs into a `MetadataSchema` object. This aggregated metadata can then be saved or returned depending on the workflow configuration. The method includes retry logic for improved reliability.

ConcurrentWorkflow.save_metadata

Saves the metadata to a JSON file based on the `auto_save` flag.

Example

```python
workflow.save_metadata()
# Metadata will be saved to the specified path if auto_save is True.
```

ConcurrentWorkflow.run

Runs the workflow for the provided task, executes agents concurrently, and saves metadata.

Parameters

Parameter	Type	Description
`task`	`str`	The task or query to give to all agents.

Returns

- `Union[Dict[str, Any], str]`: The final metadata as a dictionary or a string, depending on the `return_str_on` flag.

Detailed Explanation

This is the main method that a user will call to execute the workflow. It manages the entire process from starting the agents to collecting and optionally saving the metadata. The method also provides flexibility in how the results are returnedeither as a JSON dictionary or as a formatted string.

ConcurrentWorkflow.run_batched

Runs the workflow for a batch of tasks, executing agents concurrently for each task.

Parameters

Parameter	Type	Description

| `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

```python
tasks = ["Task 1", "Task 2"]
results = workflow.run_batched(tasks)
print(results)
```

ConcurrentWorkflow.run_async

Runs the workflow asynchronously for the given task.

Parameters

Parameter	Type	Description
`task`	`str`	The task or query to give to all agents.

Returns

- `asyncio.Future`: A future object representing the asynchronous operation.

Example

```python
async def run_async_example():
    future = workflow.run_async(task="Example task")
    result = await future
    print(result)
```

ConcurrentWorkflow.run_batched_async

Runs the workflow asynchronously for a batch of tasks.

Parameters

Parameter	Type	Description
`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

```python
tasks = ["Task 1", "Task 2"]
futures = workflow.run_batched_async(tasks)
results = await asyncio.gather(*futures)
print(results)
```

ConcurrentWorkflow.run_parallel

Runs the workflow in parallel for a batch of tasks.

Parameters

Parameter	Type	Description
`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

```python
tasks = ["Task 1", "Task 2"]
results = workflow.run_parallel(tasks)
print(results)
```

ConcurrentWorkflow.run_parallel_async

Runs the workflow in parallel asynchronously for a batch of tasks.

Parameters

Parameter	Type	Description
`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

```python
tasks = ["Task 1", "Task 2"]
```

```
futures = workflow.run_parallel_async(tasks)
results = await asyncio.gather(*futures)
print(results)
```

Usage Examples

Example 1: Basic Usage

```python
import os

from swarms import Agent, ConcurrentWorkflow, OpenAIChat

# Initialize agents
model = OpenAIChat(
    api_key=os.getenv("OPENAI_API_KEY"),
    model_name="gpt-4o-mini",
    temperature=0.1,
)


# Define custom system prompts for each social media platform
TWITTER_AGENT_SYS_PROMPT = """
You are a Twitter marketing expert specializing in real estate. Your task is to create engaging, concise tweets to promote properties, analyze trends to maximize engagement, and use appropriate
```

hashtags and timing to reach potential buyers.

"""

INSTAGRAM_AGENT_SYS_PROMPT = """

You are an Instagram marketing expert focusing on real estate. Your task is to create visually appealing posts with engaging captions and hashtags to showcase properties, targeting specific demographics interested in real estate.

"""

FACEBOOK_AGENT_SYS_PROMPT = """

You are a Facebook marketing expert for real estate. Your task is to craft posts optimized for engagement and reach on Facebook, including using images, links, and targeted messaging to attract potential property buyers.

"""

LINKEDIN_AGENT_SYS_PROMPT = """

You are a LinkedIn marketing expert for the real estate industry. Your task is to create professional and informative posts, highlighting property features, market trends, and investment opportunities, tailored to professionals and investors.

"""

EMAIL_AGENT_SYS_PROMPT = """

You are an Email marketing expert specializing in real estate. Your task is to write compelling email campaigns to promote properties, focusing on personalization, subject lines, and effective call-to-action strategies to drive conversions.

"""

```
# Initialize your agents for different social media platforms
agents = [
    Agent(
        agent_name="Twitter-RealEstate-Agent",
        system_prompt=TWITTER_AGENT_SYS_PROMPT,
        llm=model,
        max_loops=1,
        dynamic_temperature_enabled=True,
        saved_state_path="twitter_realestate_agent.json",
        user_name="swarm_corp",
        retry_attempts=1,
    ),
    Agent(
        agent_name="Instagram-RealEstate-Agent",
        system_prompt=INSTAGRAM_AGENT_SYS_PROMPT,
        llm=model,
        max_loops=1,
        dynamic_temperature_enabled=True,
        saved_state_path="instagram_realestate_agent.json",
        user_name="swarm_corp",
        retry_attempts=1,
    ),
    Agent(
        agent_name="Facebook-RealEstate-Agent",
        system_prompt=FACEBOOK_AGENT_SYS_PROMPT,
```

```
    llm=model,
    max_loops=1,
    dynamic_temperature_enabled=True,
    saved_state_path="facebook_realestate_agent.json",
    user_name="swarm_corp",
    retry_attempts=1,
),
Agent(
    agent_name="LinkedIn-RealEstate-Agent",
    system_prompt=LINKEDIN_AGENT_SYS_PROMPT,
    llm=model,
    max_loops=1,
    dynamic_temperature_enabled=True,
    saved_state_path="linkedin_realestate_agent.json",
    user_name="swarm_corp",
    retry_attempts=1,
),
Agent(
    agent_name="Email-RealEstate-Agent",
    system_prompt=EMAIL_AGENT_SYS_PROMPT,
    llm=model,
    max_loops=1,
    dynamic_temperature_enabled=True,
    saved_state_path="email_realestate_agent.json",
    user_name="swarm_corp",
    retry_attempts=1,
```

```
    ),
]

# Initialize workflow
workflow = ConcurrentWorkflow(
    name="Real Estate Marketing Swarm",
    agents=agents,
    metadata_output_path="metadata.json",
    description="Concurrent swarm of content generators for real estate!",
    auto_save=True,
)

# Run workflow
task = "Create a marketing campaign for a luxury beachfront property in Miami, focusing on its stunning ocean views, private beach access, and state-of-the-art amenities."
metadata = workflow.run(task)
print(metadata)
```

Example 2: Custom Output Handling

```python
# Initialize workflow with string output
workflow = ConcurrentWorkflow(
    name="Real Estate Marketing Swarm",
    agents=agents,
```

```
    metadata_output_path="metadata.json",
    description="Concurrent swarm of content generators for real estate!",
    auto_save=True,
    return_str_on=True
)

# Run workflow
task = "Develop a marketing strategy for a newly renovated historic townhouse in Boston, emphasizing its blend of classic architecture and modern amenities."
metadata_str = workflow.run(task)
print(metadata_str)
```

Example 3: Error Handling and Debugging

```python
import logging

# Set up logging
logging.basicConfig(level=logging.INFO)

# Initialize workflow
workflow = ConcurrentWorkflow(
    name="Real Estate Marketing Swarm",
    agents=agents,
    metadata_output_path="metadata.json",
```

```
    description="Concurrent swarm of content generators for real estate!",
    auto_save=True
)

# Run workflow with error handling
try:
    task = "Create a marketing campaign for a eco-friendly tiny house community in Portland, Oregon."
    metadata = workflow.run(task)
    print(metadata)
except Exception as e:
    logging.error(f"An error occurred during workflow execution: {str(e)}")
    # Additional error handling or debugging steps can be added here
```

Example 4: Batch Processing

```python
# Initialize workflow
workflow = ConcurrentWorkflow(
    name="Real Estate Marketing Swarm",
    agents=agents,
    metadata_output_path="metadata_batch.json",
    description="Concurrent swarm of content generators for real estate!",
    auto_save=True
)
```

```
# Define a list of tasks
tasks = [
    "Market a family-friendly suburban home with a large backyard and excellent schools nearby.",
    "Promote a high-rise luxury apartment in New York City with panoramic skyline views.",
    "Advertise a ski-in/ski-out chalet in Aspen, Colorado, perfect for winter sports enthusiasts."
]

# Run workflow in batch mode
results = workflow.run_batched(tasks)

# Process and print results
for task, result in zip(tasks, results):
    print(f"Task: {task}")
    print(f"Result: {result}\n")
```

Example 5: Asynchronous Execution

```python
import asyncio

# Initialize workflow
workflow = ConcurrentWorkflow(
    name="Real Estate Marketing Swarm",
    agents=agents,
```

```
    metadata_output_path="metadata_async.json",
    description="Concurrent swarm of content generators for real estate!",
    auto_save=True
)

async def run_async_workflow():
    task = "Develop a marketing strategy for a sustainable, off-grid mountain retreat in Colorado."
    result = await workflow.run_async(task)
    print(result)

# Run the async workflow
asyncio.run(run_async_workflow())
```

Tips and Best Practices

- **Agent Initialization**: Ensure that all agents are correctly initialized with their required configurations before passing them to `ConcurrentWorkflow`.
- **Metadata Management**: Use the `auto_save` flag to automatically save metadata if you plan to run multiple workflows in succession.
- **Concurrency Limits**: Adjust the number of agents based on your system's capabilities to avoid overloading resources.
- **Error Handling**: Implement try-except blocks when running workflows to catch and handle exceptions gracefully.
- **Batch Processing**: For large numbers of tasks, consider using `run_batched` or `run_parallel` methods to improve overall throughput.

- **Asynchronous Operations**: Utilize asynchronous methods (`run_async`, `run_batched_async`, `run_parallel_async`) when dealing with I/O-bound tasks or when you need to maintain responsiveness in your application.
- **Logging**: Implement detailed logging to track the progress of your workflows and troubleshoot any issues that may arise.
- **Resource Management**: Be mindful of API rate limits and resource consumption, especially when running large batches or parallel executions.
- **Testing**: Thoroughly test your workflows with various inputs and edge cases to ensure robust performance in production environments.

References and Resources

- [Python's `asyncio` Documentation](https://docs.python.org/3/library/asyncio.html)
- [Pydantic Documentation](https://pydantic-docs.helpmanual.io/)
- [ThreadPoolExecutor in Python](https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.ThreadPoolExecutor)
- [Loguru for Logging in Python](https://loguru.readthedocs.io/en/stable/)
- [Tenacity: Retry library for Python](https://tenacity.readthedocs.io/en/latest/)