```
import asyncio
import os
from unittest.mock import patch

import pytest

from swarm_models import OpenAIChat
from swarms.structs.agent import Agent
from swarms.structs.sequential_workflow import (
    SequentialWorkflow,
    Task,
)

# Mock the OpenAI API key using environment variables
os.environ["OPENAI_API_KEY"] = "mocked_api_key"


# Mock OpenAIChat class for testing
class MockOpenAIChat:
    def __init__(self, *args, **kwargs):
        pass

    def run(self, *args, **kwargs):
        return "Mocked result"
```

```
# Mock Agent class for testing
class MockAgent:
    def __init__(self, *args, **kwargs):
        pass

    def run(self, *args, **kwargs):
        return "Mocked result"


# Mock SequentialWorkflow class for testing
class MockSequentialWorkflow:
    def __init__(self, *args, **kwargs):
        pass

    def add(self, *args, **kwargs):
        pass

    def run(self):
        pass


# Test Task class
def test_task_initialization():
    description = "Sample Task"
    agent = MockOpenAIChat()
    task = Task(description=description, agent=agent)
```

```
    assert task.description == description
    assert task.agent == agent


def test_task_execute():
    description = "Sample Task"
    agent = MockOpenAIChat()
    task = Task(description=description, agent=agent)
    task.run()
    assert task.result == "Mocked result"


# Test SequentialWorkflow class
def test_sequential_workflow_initialization():
    workflow = SequentialWorkflow()
    assert isinstance(workflow, SequentialWorkflow)
    assert len(workflow.tasks) == 0
    assert workflow.max_loops == 1
    assert workflow.autosave is False
    assert (
        workflow.saved_state_filepath
        == "sequential_workflow_state.json"
    )
    assert workflow.restore_state_filepath is None
    assert workflow.dashboard is False
```

```
def test_sequential_workflow_add_task():
    workflow = SequentialWorkflow()
    task_description = "Sample Task"
    task_flow = MockOpenAIChat()
    workflow.add(task_description, task_flow)
    assert len(workflow.tasks) == 1
    assert workflow.tasks[0].description == task_description
    assert workflow.tasks[0].agent == task_flow


def test_sequential_workflow_reset_workflow():
    workflow = SequentialWorkflow()
    task_description = "Sample Task"
    task_flow = MockOpenAIChat()
    workflow.add(task_description, task_flow)
    workflow.reset_workflow()
    assert workflow.tasks[0].result is None


def test_sequential_workflow_get_task_results():
    workflow = SequentialWorkflow()
    task_description = "Sample Task"
    task_flow = MockOpenAIChat()
    workflow.add(task_description, task_flow)
    workflow.run()
```

```
    results = workflow.get_task_results()
    assert len(results) == 1
    assert task_description in results
    assert results[task_description] == "Mocked result"


def test_sequential_workflow_remove_task():
    workflow = SequentialWorkflow()
    task1_description = "Task 1"
    task2_description = "Task 2"
    task1_flow = MockOpenAIChat()
    task2_flow = MockOpenAIChat()
    workflow.add(task1_description, task1_flow)
    workflow.add(task2_description, task2_flow)
    workflow.remove_task(task1_description)
    assert len(workflow.tasks) == 1
    assert workflow.tasks[0].description == task2_description


def test_sequential_workflow_update_task():
    workflow = SequentialWorkflow()
    task_description = "Sample Task"
    task_flow = MockOpenAIChat()
    workflow.add(task_description, task_flow)
    workflow.update_task(task_description, max_tokens=1000)
    assert workflow.tasks[0].kwargs["max_tokens"] == 1000
```

```
def test_sequential_workflow_save_workflow_state():
    workflow = SequentialWorkflow()
    task_description = "Sample Task"
    task_flow = MockOpenAIChat()
    workflow.add(task_description, task_flow)
    workflow.save_workflow_state("test_state.json")
    assert os.path.exists("test_state.json")
    os.remove("test_state.json")


def test_sequential_workflow_load_workflow_state():
    workflow = SequentialWorkflow()
    task_description = "Sample Task"
    task_flow = MockOpenAIChat()
    workflow.add(task_description, task_flow)
    workflow.save_workflow_state("test_state.json")
    workflow.load_workflow_state("test_state.json")
    assert len(workflow.tasks) == 1
    assert workflow.tasks[0].description == task_description
    os.remove("test_state.json")


def test_sequential_workflow_run():
    workflow = SequentialWorkflow()
```

```
    task_description = "Sample Task"
    task_flow = MockOpenAIChat()
    workflow.add(task_description, task_flow)
    workflow.run()
    assert workflow.tasks[0].result == "Mocked result"


def test_sequential_workflow_workflow_bootup(capfd):
    workflow = SequentialWorkflow()
    workflow.workflow_bootup()
    out, _ = capfd.readouterr()
    assert "Sequential Workflow Initializing..." in out


def test_sequential_workflow_workflow_dashboard(capfd):
    workflow = SequentialWorkflow()
    workflow.workflow_dashboard()
    out, _ = capfd.readouterr()
    assert "Sequential Workflow Dashboard" in out


# Mock Agent class for async testing
class MockAsyncAgent:
    def __init__(self, *args, **kwargs):
        pass
```

```
    async def arun(self, *args, **kwargs):
        return "Mocked result"


# Test async execution in SequentialWorkflow
@pytest.mark.asyncio
async def test_sequential_workflow_arun():
    workflow = SequentialWorkflow()
    task_description = "Sample Task"
    task_flow = MockAsyncAgent()
    workflow.add(task_description, task_flow)
    await workflow.arun()
    assert workflow.tasks[0].result == "Mocked result"


def test_real_world_usage_with_openai_key():
    # Initialize the language model
    llm = OpenAIChat()
    assert isinstance(llm, OpenAIChat)


def test_real_world_usage_with_flow_and_openai_key():
    # Initialize a agent with the language model
    agent = Agent(llm=OpenAIChat())
    assert isinstance(agent, Agent)
```

```
def test_real_world_usage_with_sequential_workflow():
    # Initialize a sequential workflow
    workflow = SequentialWorkflow()
    assert isinstance(workflow, SequentialWorkflow)


def test_real_world_usage_add_tasks():
    # Create a sequential workflow and add tasks
    workflow = SequentialWorkflow()
    task1_description = "Task 1"
    task2_description = "Task 2"
    task1_flow = OpenAIChat()
    task2_flow = OpenAIChat()
    workflow.add(task1_description, task1_flow)
    workflow.add(task2_description, task2_flow)
    assert len(workflow.tasks) == 2
    assert workflow.tasks[0].description == task1_description
    assert workflow.tasks[1].description == task2_description


def test_real_world_usage_run_workflow():
    # Create a sequential workflow, add a task, and run the workflow
    workflow = SequentialWorkflow()
    task_description = "Sample Task"
    task_flow = OpenAIChat()
```

```
    workflow.add(task_description, task_flow)
    workflow.run()
    assert workflow.tasks[0].result is not None


def test_real_world_usage_dashboard_display():
    # Create a sequential workflow, add tasks, and display the dashboard
    workflow = SequentialWorkflow()
    task1_description = "Task 1"
    task2_description = "Task 2"
    task1_flow = OpenAIChat()
    task2_flow = OpenAIChat()
    workflow.add(task1_description, task1_flow)
    workflow.add(task2_description, task2_flow)
    with patch("builtins.print") as mock_print:
        workflow.workflow_dashboard()
        mock_print.assert_called()


def test_real_world_usage_async_execution():
    # Create a sequential workflow, add an async task, and run the workflow asynchronously
    workflow = SequentialWorkflow()
    task_description = "Sample Task"
    async_task_flow = OpenAIChat()

    async def async_run_workflow():
```

```
        await workflow.arun()


    workflow.add(task_description, async_task_flow)
    asyncio.run(async_run_workflow())
    assert workflow.tasks[0].result is not None



def test_real_world_usage_multiple_loops():
    # Create a sequential workflow with multiple loops, add a task, and run the workflow
    workflow = SequentialWorkflow(max_loops=3)
    task_description = "Sample Task"
    task_flow = OpenAIChat()
    workflow.add(task_description, task_flow)
    workflow.run()
    assert workflow.tasks[0].result is not None



def test_real_world_usage_autosave_state():
     # Create a sequential workflow with autosave, add a task, run the workflow, and check if state is
saved
    workflow = SequentialWorkflow(autosave=True)
    task_description = "Sample Task"
    task_flow = OpenAIChat()
    workflow.add(task_description, task_flow)
    workflow.run()
    assert workflow.tasks[0].result is not None
```

```
    assert os.path.exists("sequential_workflow_state.json")
    os.remove("sequential_workflow_state.json")


def test_real_world_usage_load_state():
    # Create a sequential workflow, add a task, save state, load state, and run the workflow
    workflow = SequentialWorkflow()
    task_description = "Sample Task"
    task_flow = OpenAIChat()
    workflow.add(task_description, task_flow)
    workflow.run()
    workflow.save_workflow_state("test_state.json")
    workflow.load_workflow_state("test_state.json")
    workflow.run()
    assert workflow.tasks[0].result is not None
    os.remove("test_state.json")


def test_real_world_usage_update_task_args():
    # Create a sequential workflow, add a task, and update task arguments
    workflow = SequentialWorkflow()
    task_description = "Sample Task"
    task_flow = OpenAIChat()
    workflow.add(task_description, task_flow)
    workflow.update_task(task_description, max_tokens=1000)
    assert workflow.tasks[0].kwargs["max_tokens"] == 1000
```

```
def test_real_world_usage_remove_task():
    # Create a sequential workflow, add tasks, remove a task, and run the workflow
    workflow = SequentialWorkflow()
    task1_description = "Task 1"
    task2_description = "Task 2"
    task1_flow = OpenAIChat()
    task2_flow = OpenAIChat()
    workflow.add(task1_description, task1_flow)
    workflow.add(task2_description, task2_flow)
    workflow.remove_task(task1_description)
    workflow.run()
    assert len(workflow.tasks) == 1
    assert workflow.tasks[0].description == task2_description


def test_real_world_usage_with_environment_variables():
    # Ensure that the OpenAI API key is set using environment variables
    assert "OPENAI_API_KEY" in os.environ
    assert os.environ["OPENAI_API_KEY"] == "mocked_api_key"
    del os.environ["OPENAI_API_KEY"]  # Clean up after the test


def test_real_world_usage_no_openai_key():
    # Ensure that an exception is raised when the OpenAI API key is not set
```

```
with pytest.raises(ValueError):
    OpenAIChat()  # API key not provided, should raise an exception
```