```
import json
import os
from unittest import mock
from unittest.mock import MagicMock, patch

import pytest
from dotenv import load_dotenv

from swarm_models import OpenAIChat
from swarms.structs.agent import Agent, stop_when_repeats
from swarms.utils.loguru_logger import logger

load_dotenv()

openai_api_key = os.getenv("OPENAI_API_KEY")


# Mocks and Fixtures
@pytest.fixture
def mocked_llm():
    return OpenAIChat(
        openai_api_key=openai_api_key,
    )


@pytest.fixture
```

```
def basic_flow(mocked_llm):
    return Agent(llm=mocked_llm, max_loops=5)



@pytest.fixture
def flow_with_condition(mocked_llm):
    return Agent(
        llm=mocked_llm,
        max_loops=5,
        stopping_condition=stop_when_repeats,
    )



# Basic Tests
def test_stop_when_repeats():
    assert stop_when_repeats("Please Stop now")
    assert not stop_when_repeats("Continue the process")



def test_flow_initialization(basic_flow):
    assert basic_flow.max_loops == 5
    assert basic_flow.stopping_condition is None
    assert basic_flow.loop_interval == 1
    assert basic_flow.retry_attempts == 3
    assert basic_flow.retry_interval == 1
    assert basic_flow.feedback == []
    assert basic_flow.memory == []

    assert basic_flow.task is None

    assert basic_flow.stopping_token == "<DONE>"

    assert not basic_flow.interactive




def test_provide_feedback(basic_flow):

    feedback = "Test feedback"

    basic_flow.provide_feedback(feedback)

    assert feedback in basic_flow.feedback




@patch("time.sleep", return_value=None)  # to speed up tests

def test_run_without_stopping_condition(mocked_sleep, basic_flow):

    response = basic_flow.run("Test task")

    assert (

        response == "Test task"

    )  # since our mocked llm doesn't modify the response




@patch("time.sleep", return_value=None)  # to speed up tests

def test_run_with_stopping_condition(

    mocked_sleep, flow_with_condition

):

    response = flow_with_condition.run("Stop")

    assert response == "Stop"
```

```
@patch("time.sleep", return_value=None)  # to speed up tests
def test_run_with_exception(mocked_sleep, basic_flow):
    basic_flow.llm.side_effect = Exception("Test Exception")
    with pytest.raises(Exception, match="Test Exception"):
        basic_flow.run("Test task")


def test_bulk_run(basic_flow):
    inputs = [{"task": "Test1"}, {"task": "Test2"}]
    responses = basic_flow.bulk_run(inputs)
    assert responses == ["Test1", "Test2"]


# Tests involving file IO
def test_save_and_load(basic_flow, tmp_path):
    file_path = tmp_path / "memory.json"
    basic_flow.memory.append(["Test1", "Test2"])
    basic_flow.save(file_path)

    new_flow = Agent(llm=mocked_llm, max_loops=5)
    new_flow.load(file_path)
    assert new_flow.memory == [["Test1", "Test2"]]
```

```
# Environment variable mock test
def test_env_variable_handling(monkeypatch):
    monkeypatch.setenv("API_KEY", "test_key")
    assert os.getenv("API_KEY") == "test_key"


# TODO: Add more tests, especially edge cases and exception cases. Implement parametrized tests for varied inputs.


# Test initializing the agent with different stopping conditions
def test_flow_with_custom_stopping_condition(mocked_llm):
    def stopping_condition(x):
        return "terminate" in x.lower()

    agent = Agent(
        llm=mocked_llm,
        max_loops=5,
        stopping_condition=stopping_condition,
    )
    assert agent.stopping_condition("Please terminate now")
    assert not agent.stopping_condition("Continue the process")


# Test calling the agent directly
def test_flow_call(basic_flow):
    response = basic_flow("Test call")
    assert response == "Test call"


# Test formatting the prompt
def test_format_prompt(basic_flow):
    formatted_prompt = basic_flow.format_prompt(
        "Hello {name}", name="John"
    )
    assert formatted_prompt == "Hello John"


# Test with max loops
@patch("time.sleep", return_value=None)
def test_max_loops(mocked_sleep, basic_flow):
    basic_flow.max_loops = 3
    response = basic_flow.run("Looping")
    assert response == "Looping"


# Test stopping token
@patch("time.sleep", return_value=None)
def test_stopping_token(mocked_sleep, basic_flow):
    basic_flow.stopping_token = "Terminate"
    response = basic_flow.run("Loop until Terminate")
    assert response == "Loop until Terminate"
```

```
# Test interactive mode

def test_interactive_mode(basic_flow):

    basic_flow.interactive = True

    assert basic_flow.interactive




# Test bulk run with varied inputs

def test_bulk_run_varied_inputs(basic_flow):

    inputs = [

        {"task": "Test1"},

        {"task": "Test2"},

        {"task": "Stop now"},

    ]

    responses = basic_flow.bulk_run(inputs)

    assert responses == ["Test1", "Test2", "Stop now"]




# Test loading non-existent file

def test_load_non_existent_file(basic_flow, tmp_path):

    file_path = tmp_path / "non_existent.json"

    with pytest.raises(FileNotFoundError):

        basic_flow.load(file_path)
```

```
# Test saving with different memory data

def test_save_different_memory(basic_flow, tmp_path):

    file_path = tmp_path / "memory.json"

    basic_flow.memory.append(["Task1", "Task2", "Task3"])

    basic_flow.save(file_path)

    with open(file_path) as f:

        data = json.load(f)

    assert data == [["Task1", "Task2", "Task3"]]



# Test the stopping condition check

def test_check_stopping_condition(flow_with_condition):

    assert flow_with_condition._check_stopping_condition(

        "Stop this process"

    )

    assert not flow_with_condition._check_stopping_condition(

        "Continue the task"

    )



# Test without providing max loops (default value should be 5)

def test_default_max_loops(mocked_llm):

    agent = Agent(llm=mocked_llm)

    assert agent.max_loops == 5
```

```
# Test creating agent from llm and template
def test_from_llm_and_template(mocked_llm):
    agent = Agent.from_llm_and_template(mocked_llm, "Test template")
    assert isinstance(agent, Agent)


# Mocking the OpenAIChat for testing
@patch("swarms.models.OpenAIChat", autospec=True)
def test_mocked_openai_chat(MockedOpenAIChat):
    llm = MockedOpenAIChat(openai_api_key=openai_api_key)
    llm.return_value = MagicMock()
    agent = Agent(llm=llm, max_loops=5)
    agent.run("Mocked run")
    assert MockedOpenAIChat.called


# Test retry attempts
@patch("time.sleep", return_value=None)
def test_retry_attempts(mocked_sleep, basic_flow):
    basic_flow.retry_attempts = 2
    basic_flow.llm.side_effect = [
        Exception("Test Exception"),
        "Valid response",
    ]
    response = basic_flow.run("Test retry")
    assert response == "Valid response"
```

```
# Test different loop intervals
@patch("time.sleep", return_value=None)
def test_different_loop_intervals(mocked_sleep, basic_flow):
    basic_flow.loop_interval = 2
    response = basic_flow.run("Test loop interval")
    assert response == "Test loop interval"


# Test different retry intervals
@patch("time.sleep", return_value=None)
def test_different_retry_intervals(mocked_sleep, basic_flow):
    basic_flow.retry_interval = 2
    response = basic_flow.run("Test retry interval")
    assert response == "Test retry interval"


# Test invoking the agent with additional kwargs
@patch("time.sleep", return_value=None)
def test_flow_call_with_kwargs(mocked_sleep, basic_flow):
    response = basic_flow(
        "Test call", param1="value1", param2="value2"
    )
    assert response == "Test call"
```

```
# Test initializing the agent with all parameters

def test_flow_initialization_all_params(mocked_llm):

    agent = Agent(

        llm=mocked_llm,

        max_loops=10,

        stopping_condition=stop_when_repeats,

        loop_interval=2,

        retry_attempts=4,

        retry_interval=2,

        interactive=True,

        param1="value1",

        param2="value2",

    )

    assert agent.max_loops == 10

    assert agent.loop_interval == 2

    assert agent.retry_attempts == 4

    assert agent.retry_interval == 2

    assert agent.interactive



# Test the stopping token is in the response

@patch("time.sleep", return_value=None)

def test_stopping_token_in_response(mocked_sleep, basic_flow):

    response = basic_flow.run("Test stopping token")

    assert basic_flow.stopping_token in response
```

```
@pytest.fixture
def flow_instance():
    # Create an instance of the Agent class with required parameters for testing
    # You may need to adjust this based on your actual class initialization
    llm = OpenAIChat(
        openai_api_key=openai_api_key,
    )
    agent = Agent(
        llm=llm,
        max_loops=5,
        interactive=False,
        dashboard=False,
        dynamic_temperature=False,
    )
    return agent


def test_flow_run(flow_instance):
    # Test the basic run method of the Agent class
    response = flow_instance.run("Test task")
    assert isinstance(response, str)
    assert len(response) > 0
```

```
def test_flow_interactive_mode(flow_instance):

    # Test the interactive mode of the Agent class

    flow_instance.interactive = True

    response = flow_instance.run("Test task")

    assert isinstance(response, str)

    assert len(response) > 0



def test_flow_dashboard_mode(flow_instance):

    # Test the dashboard mode of the Agent class

    flow_instance.dashboard = True

    response = flow_instance.run("Test task")

    assert isinstance(response, str)

    assert len(response) > 0



def test_flow_autosave(flow_instance):

    # Test the autosave functionality of the Agent class

    flow_instance.autosave = True

    response = flow_instance.run("Test task")

    assert isinstance(response, str)

    assert len(response) > 0

    # Ensure that the state is saved (you may need to implement this logic)

    assert flow_instance.saved_state_path is not None
```

```
def test_flow_response_filtering(flow_instance):
    # Test the response filtering functionality
    flow_instance.add_response_filter("filter_this")
    response = flow_instance.filtered_run(
        "This message should filter_this"
    )
    assert "filter_this" not in response


def test_flow_undo_last(flow_instance):
    # Test the undo functionality
    response1 = flow_instance.run("Task 1")
    flow_instance.run("Task 2")
    previous_state, message = flow_instance.undo_last()
    assert response1 == previous_state
    assert "Restored to" in message


def test_flow_dynamic_temperature(flow_instance):
    # Test dynamic temperature adjustment
    flow_instance.dynamic_temperature = True
    response = flow_instance.run("Test task")
    assert isinstance(response, str)
    assert len(response) > 0
```

```
def test_flow_streamed_generation(flow_instance):

    # Test streamed generation

    response = flow_instance.streamed_generation("Generating...")

    assert isinstance(response, str)

    assert len(response) > 0



def test_flow_step(flow_instance):

    # Test the step method

    response = flow_instance.step("Test step")

    assert isinstance(response, str)

    assert len(response) > 0



def test_flow_graceful_shutdown(flow_instance):

    # Test graceful shutdown

    result = flow_instance.graceful_shutdown()

    assert result is not None



# Add more test cases as needed to cover various aspects of your Agent class



def test_flow_max_loops(flow_instance):

    # Test setting and getting the maximum number of loops

    flow_instance.set_max_loops(10)
    assert flow_instance.get_max_loops() == 10


def test_flow_autosave_path(flow_instance):
    # Test setting and getting the autosave path
    flow_instance.set_autosave_path("text.txt")
    assert flow_instance.get_autosave_path() == "txt.txt"


def test_flow_response_length(flow_instance):
    # Test checking the length of the response
    response = flow_instance.run(
        "Generate a 10,000 word long blog on mental clarity and the"
        " benefits of meditation."
    )
    assert (
        len(response) > flow_instance.get_response_length_threshold()
    )


def test_flow_set_response_length_threshold(flow_instance):
    # Test setting and getting the response length threshold
    flow_instance.set_response_length_threshold(100)
    assert flow_instance.get_response_length_threshold() == 100
```

```
def test_flow_add_custom_filter(flow_instance):
    # Test adding a custom response filter
    flow_instance.add_response_filter("custom_filter")
    assert "custom_filter" in flow_instance.get_response_filters()


def test_flow_remove_custom_filter(flow_instance):
    # Test removing a custom response filter
    flow_instance.add_response_filter("custom_filter")
    flow_instance.remove_response_filter("custom_filter")
    assert "custom_filter" not in flow_instance.get_response_filters()


def test_flow_dynamic_pacing(flow_instance):
    # Test dynamic pacing
    flow_instance.enable_dynamic_pacing()
    assert flow_instance.is_dynamic_pacing_enabled() is True


def test_flow_disable_dynamic_pacing(flow_instance):
    # Test disabling dynamic pacing
    flow_instance.disable_dynamic_pacing()
    assert flow_instance.is_dynamic_pacing_enabled() is False


def test_flow_change_prompt(flow_instance):
    # Test changing the current prompt
    flow_instance.change_prompt("New prompt")
    assert flow_instance.get_current_prompt() == "New prompt"


def test_flow_add_instruction(flow_instance):
    # Test adding an instruction to the conversation
    flow_instance.add_instruction("Follow these steps:")
    assert "Follow these steps:" in flow_instance.get_instructions()


def test_flow_clear_instructions(flow_instance):
    # Test clearing all instructions from the conversation
    flow_instance.add_instruction("Follow these steps:")
    flow_instance.clear_instructions()
    assert len(flow_instance.get_instructions()) == 0


def test_flow_add_user_message(flow_instance):
    # Test adding a user message to the conversation
    flow_instance.add_user_message("User message")
    assert "User message" in flow_instance.get_user_messages()


def test_flow_clear_user_messages(flow_instance):
    # Test clearing all user messages from the conversation
    flow_instance.add_user_message("User message")

    flow_instance.clear_user_messages()

    assert len(flow_instance.get_user_messages()) == 0




def test_flow_get_response_history(flow_instance):

    # Test getting the response history

    flow_instance.run("Message 1")

    flow_instance.run("Message 2")

    history = flow_instance.get_response_history()

    assert len(history) == 2

    assert "Message 1" in history[0]

    assert "Message 2" in history[1]




def test_flow_clear_response_history(flow_instance):

    # Test clearing the response history

    flow_instance.run("Message 1")

    flow_instance.run("Message 2")

    flow_instance.clear_response_history()

    assert len(flow_instance.get_response_history()) == 0




def test_flow_get_conversation_log(flow_instance):

    # Test getting the entire conversation log

    flow_instance.run("Message 1")
    flow_instance.run("Message 2")
    conversation_log = flow_instance.get_conversation_log()
    assert (
        len(conversation_log) == 4
    )  # Including system and user messages


def test_flow_clear_conversation_log(flow_instance):
    # Test clearing the entire conversation log
    flow_instance.run("Message 1")
    flow_instance.run("Message 2")
    flow_instance.clear_conversation_log()
    assert len(flow_instance.get_conversation_log()) == 0


def test_flow_get_state(flow_instance):
    # Test getting the current state of the Agent instance
    state = flow_instance.get_state()
    assert isinstance(state, dict)
    assert "current_prompt" in state
    assert "instructions" in state
    assert "user_messages" in state
    assert "response_history" in state
    assert "conversation_log" in state
    assert "dynamic_pacing_enabled" in state
    assert "response_length_threshold" in state
    assert "response_filters" in state
    assert "max_loops" in state
    assert "autosave_path" in state


def test_flow_load_state(flow_instance):
    # Test loading the state into the Agent instance
    state = {
        "current_prompt": "Loaded prompt",
        "instructions": ["Step 1", "Step 2"],
        "user_messages": ["User message 1", "User message 2"],
        "response_history": ["Response 1", "Response 2"],
        "conversation_log": [
            "System message 1",
            "User message 1",
            "System message 2",
            "User message 2",
        ],
        "dynamic_pacing_enabled": True,
        "response_length_threshold": 50,
        "response_filters": ["filter1", "filter2"],
        "max_loops": 10,
        "autosave_path": "/path/to/load",
    }
    flow_instance.load(state)
    assert flow_instance.get_current_prompt() == "Loaded prompt"
    assert "Step 1" in flow_instance.get_instructions()

    assert "User message 1" in flow_instance.get_user_messages()

    assert "Response 1" in flow_instance.get_response_history()

    assert "System message 1" in flow_instance.get_conversation_log()

    assert flow_instance.is_dynamic_pacing_enabled() is True

    assert flow_instance.get_response_length_threshold() == 50

    assert "filter1" in flow_instance.get_response_filters()

    assert flow_instance.get_max_loops() == 10

    assert flow_instance.get_autosave_path() == "/path/to/load"



def test_flow_save_state(flow_instance):

    # Test saving the state of the Agent instance

    flow_instance.change_prompt("New prompt")

    flow_instance.add_instruction("Step 1")

    flow_instance.add_user_message("User message")

    flow_instance.run("Response")

    state = flow_instance.save_state()

    assert "current_prompt" in state

    assert "instructions" in state

    assert "user_messages" in state

    assert "response_history" in state

    assert "conversation_log" in state

    assert "dynamic_pacing_enabled" in state

    assert "response_length_threshold" in state

    assert "response_filters" in state
    assert "max_loops" in state
    assert "autosave_path" in state


def test_flow_rollback(flow_instance):
    # Test rolling back to a previous state
    state1 = flow_instance.get_state()
    flow_instance.change_prompt("New prompt")
    flow_instance.get_state()
    flow_instance.rollback_to_state(state1)
    assert (
        flow_instance.get_current_prompt() == state1["current_prompt"]
    )
    assert flow_instance.get_instructions() == state1["instructions"]
    assert (
        flow_instance.get_user_messages() == state1["user_messages"]
    )
    assert (
        flow_instance.get_response_history()
        == state1["response_history"]
    )
    assert (
        flow_instance.get_conversation_log()
        == state1["conversation_log"]
    )
    assert (
        flow_instance.is_dynamic_pacing_enabled()
        == state1["dynamic_pacing_enabled"]
    )
    assert (
        flow_instance.get_response_length_threshold()
        == state1["response_length_threshold"]
    )
    assert (
        flow_instance.get_response_filters()
        == state1["response_filters"]
    )
    assert flow_instance.get_max_loops() == state1["max_loops"]
    assert (
        flow_instance.get_autosave_path() == state1["autosave_path"]
    )
    assert flow_instance.get_state() == state1


def test_flow_contextual_intent(flow_instance):
    # Test contextual intent handling
    flow_instance.add_context("location", "New York")
    flow_instance.add_context("time", "tomorrow")
    response = flow_instance.run(
        "What's the weather like in {location} at {time}?"
    )
    assert "New York" in response
    assert "tomorrow" in response


def test_flow_contextual_intent_override(flow_instance):
    # Test contextual intent override
    flow_instance.add_context("location", "New York")
    response1 = flow_instance.run(
        "What's the weather like in {location}?"
    )
    flow_instance.add_context("location", "Los Angeles")
    response2 = flow_instance.run(
        "What's the weather like in {location}?"
    )
    assert "New York" in response1
    assert "Los Angeles" in response2


def test_flow_contextual_intent_reset(flow_instance):
    # Test resetting contextual intent
    flow_instance.add_context("location", "New York")
    response1 = flow_instance.run(
        "What's the weather like in {location}?"
    )
    flow_instance.reset_context()
    response2 = flow_instance.run(
        "What's the weather like in {location}?"
    )
    assert "New York" in response1
    assert "New York" in response2


# Add more test cases as needed to cover various aspects of your Agent class
def test_flow_interruptible(flow_instance):
    # Test interruptible mode
    flow_instance.interruptible = True
    response = flow_instance.run("Interrupt me!")
    assert "Interrupted" in response
    assert flow_instance.is_interrupted() is True


def test_flow_non_interruptible(flow_instance):
    # Test non-interruptible mode
    flow_instance.interruptible = False
    response = flow_instance.run("Do not interrupt me!")
    assert "Do not interrupt me!" in response
    assert flow_instance.is_interrupted() is False


def test_flow_timeout(flow_instance):
    # Test conversation timeout
    flow_instance.timeout = 60  # Set a timeout of 60 seconds
    response = flow_instance.run(
        "This should take some time to respond."
    )
    assert "Timed out" in response
    assert flow_instance.is_timed_out() is True


def test_flow_no_timeout(flow_instance):
    # Test no conversation timeout
    flow_instance.timeout = None
    response = flow_instance.run("This should not time out.")
    assert "This should not time out." in response
    assert flow_instance.is_timed_out() is False


def test_flow_custom_delimiter(flow_instance):
    # Test setting and getting a custom message delimiter
    flow_instance.set_message_delimiter("|||")
    assert flow_instance.get_message_delimiter() == "|||"


def test_flow_message_history(flow_instance):
    # Test getting the message history
    flow_instance.run("Message 1")
    flow_instance.run("Message 2")
    history = flow_instance.get_message_history()
    assert len(history) == 2
    assert "Message 1" in history[0]

    assert "Message 2" in history[1]




def test_flow_clear_message_history(flow_instance):

    # Test clearing the message history

    flow_instance.run("Message 1")

    flow_instance.run("Message 2")

    flow_instance.clear_message_history()

    assert len(flow_instance.get_message_history()) == 0




def test_flow_save_and_load_conversation(flow_instance):

    # Test saving and loading the conversation

    flow_instance.run("Message 1")

    flow_instance.run("Message 2")

    saved_conversation = flow_instance.save_conversation()

    flow_instance.clear_conversation()

    flow_instance.load_conversation(saved_conversation)

    assert len(flow_instance.get_message_history()) == 2




def test_flow_inject_custom_system_message(flow_instance):

    # Test injecting a custom system message into the conversation

    flow_instance.inject_custom_system_message(

        "Custom system message"
    )
    assert (
        "Custom system message" in flow_instance.get_message_history()
    )


def test_flow_inject_custom_user_message(flow_instance):
    # Test injecting a custom user message into the conversation
    flow_instance.inject_custom_user_message("Custom user message")
    assert (
        "Custom user message" in flow_instance.get_message_history()
    )


def test_flow_inject_custom_response(flow_instance):
    # Test injecting a custom response into the conversation
    flow_instance.inject_custom_response("Custom response")
    assert "Custom response" in flow_instance.get_message_history()


def test_flow_clear_injected_messages(flow_instance):
    # Test clearing injected messages from the conversation
    flow_instance.inject_custom_system_message(
        "Custom system message"
    )
    flow_instance.inject_custom_user_message("Custom user message")
    flow_instance.inject_custom_response("Custom response")
    flow_instance.clear_injected_messages()
    assert (
        "Custom system message"
        not in flow_instance.get_message_history()
    )
    assert (
        "Custom user message"
        not in flow_instance.get_message_history()
    )
    assert (
        "Custom response" not in flow_instance.get_message_history()
    )


def test_flow_disable_message_history(flow_instance):
    # Test disabling message history recording
    flow_instance.disable_message_history()
    response = flow_instance.run(
        "This message should not be recorded in history."
    )
    assert (
        "This message should not be recorded in history." in response
    )
    assert (
        len(flow_instance.get_message_history()) == 0
    )  # History is empty


def test_flow_enable_message_history(flow_instance):
    # Test enabling message history recording
    flow_instance.enable_message_history()
    response = flow_instance.run(
        "This message should be recorded in history."
    )
    assert "This message should be recorded in history." in response
    assert len(flow_instance.get_message_history()) == 1


def test_flow_custom_logger(flow_instance):
    # Test setting and using a custom logger
    custom_logger = logger  # Replace with your custom logger class
    flow_instance.set_logger(custom_logger)
    response = flow_instance.run("Custom logger test")
    assert (
        "Logged using custom logger" in response
    )  # Verify logging message


def test_flow_batch_processing(flow_instance):
    # Test batch processing of messages
    messages = ["Message 1", "Message 2", "Message 3"]
    responses = flow_instance.process_batch(messages)

    assert isinstance(responses, list)

    assert len(responses) == len(messages)

    for response in responses:

        assert isinstance(response, str)



def test_flow_custom_metrics(flow_instance):

    # Test tracking custom metrics

    flow_instance.track_custom_metric("custom_metric_1", 42)

    flow_instance.track_custom_metric("custom_metric_2", 3.14)

    metrics = flow_instance.get_custom_metrics()

    assert "custom_metric_1" in metrics

    assert "custom_metric_2" in metrics

    assert metrics["custom_metric_1"] == 42

    assert metrics["custom_metric_2"] == 3.14



def test_flow_reset_metrics(flow_instance):

    # Test resetting custom metrics

    flow_instance.track_custom_metric("custom_metric_1", 42)

    flow_instance.track_custom_metric("custom_metric_2", 3.14)

    flow_instance.reset_custom_metrics()

    metrics = flow_instance.get_custom_metrics()

    assert len(metrics) == 0
```

```
def test_flow_retrieve_context(flow_instance):
    # Test retrieving context
    flow_instance.add_context("location", "New York")
    context = flow_instance.get_context("location")
    assert context == "New York"


def test_flow_update_context(flow_instance):
    # Test updating context
    flow_instance.add_context("location", "New York")
    flow_instance.update_context("location", "Los Angeles")
    context = flow_instance.get_context("location")
    assert context == "Los Angeles"


def test_flow_remove_context(flow_instance):
    # Test removing context
    flow_instance.add_context("location", "New York")
    flow_instance.remove_context("location")
    context = flow_instance.get_context("location")
    assert context is None


def test_flow_clear_context(flow_instance):
    # Test clearing all context
    flow_instance.add_context("location", "New York")
    flow_instance.add_context("time", "tomorrow")
    flow_instance.clear_context()
    context_location = flow_instance.get_context("location")
    context_time = flow_instance.get_context("time")
    assert context_location is None
    assert context_time is None


def test_flow_input_validation(flow_instance):
    # Test input validation for invalid agent configurations
    with pytest.raises(ValueError):
        Agent(config=None)  # Invalid config, should raise ValueError

    with pytest.raises(ValueError):
        flow_instance.set_message_delimiter(
            ""
        )  # Empty delimiter, should raise ValueError

    with pytest.raises(ValueError):
        flow_instance.set_message_delimiter(
            None
        )  # None delimiter, should raise ValueError

    with pytest.raises(ValueError):
        flow_instance.set_message_delimiter(
            123
        )  # Invalid delimiter type, should raise ValueError

    with pytest.raises(ValueError):
        flow_instance.set_logger(
            "invalid_logger"
        )  # Invalid logger type, should raise ValueError

    with pytest.raises(ValueError):
        flow_instance.add_context(
            None, "value"
        )  # None key, should raise ValueError

    with pytest.raises(ValueError):
        flow_instance.add_context(
            "key", None
        )  # None value, should raise ValueError

    with pytest.raises(ValueError):
        flow_instance.update_context(
            None, "value"
        )  # None key, should raise ValueError

    with pytest.raises(ValueError):
        flow_instance.update_context(
            "key", None
    )  # None value, should raise ValueError


def test_flow_conversation_reset(flow_instance):
    # Test conversation reset
    flow_instance.run("Message 1")
    flow_instance.run("Message 2")
    flow_instance.reset_conversation()
    assert len(flow_instance.get_message_history()) == 0


def test_flow_conversation_persistence(flow_instance):
    # Test conversation persistence across instances
    flow_instance.run("Message 1")
    flow_instance.run("Message 2")
    conversation = flow_instance.get_conversation()

    new_flow_instance = Agent()
    new_flow_instance.load_conversation(conversation)
    assert len(new_flow_instance.get_message_history()) == 2
    assert "Message 1" in new_flow_instance.get_message_history()[0]
    assert "Message 2" in new_flow_instance.get_message_history()[1]


def test_flow_custom_event_listener(flow_instance):
    # Test custom event listener
    class CustomEventListener:
        def on_message_received(self, message):
            pass

        def on_response_generated(self, response):
            pass

    custom_event_listener = CustomEventListener()
    flow_instance.add_event_listener(custom_event_listener)

    # Ensure that the custom event listener methods are called during a conversation
    with mock.patch.object(
        custom_event_listener, "on_message_received"
    ) as mock_received, mock.patch.object(
        custom_event_listener, "on_response_generated"
    ) as mock_response:
        flow_instance.run("Message 1")
        mock_received.assert_called_once()
        mock_response.assert_called_once()


def test_flow_multiple_event_listeners(flow_instance):
    # Test multiple event listeners
    class FirstEventListener:
        def on_message_received(self, message):
            pass
        def on_response_generated(self, response):
            pass


    class SecondEventListener:
        def on_message_received(self, message):
            pass


        def on_response_generated(self, response):
            pass


    first_event_listener = FirstEventListener()
    second_event_listener = SecondEventListener()
    flow_instance.add_event_listener(first_event_listener)
    flow_instance.add_event_listener(second_event_listener)


    # Ensure that both event listeners receive events during a conversation
    with mock.patch.object(
        first_event_listener, "on_message_received"
    ) as mock_first_received, mock.patch.object(
        first_event_listener, "on_response_generated"
    ) as mock_first_response, mock.patch.object(
        second_event_listener, "on_message_received"
    ) as mock_second_received, mock.patch.object(
        second_event_listener, "on_response_generated"
    ) as mock_second_response:
        flow_instance.run("Message 1")
        mock_first_received.assert_called_once()
        mock_first_response.assert_called_once()
        mock_second_received.assert_called_once()
        mock_second_response.assert_called_once()



# Add more test cases as needed to cover various aspects of your Agent class
def test_flow_error_handling(flow_instance):
    # Test error handling and exceptions
    with pytest.raises(ValueError):
        flow_instance.set_message_delimiter(
            ""
        )  # Empty delimiter, should raise ValueError

    with pytest.raises(ValueError):
        flow_instance.set_message_delimiter(
            None
        )  # None delimiter, should raise ValueError

    with pytest.raises(ValueError):
        flow_instance.set_logger(
            "invalid_logger"
        )  # Invalid logger type, should raise ValueError

    with pytest.raises(ValueError):
        flow_instance.add_context(
            None, "value"
        )  # None key, should raise ValueError

    with pytest.raises(ValueError):
        flow_instance.add_context(
            "key", None
        )  # None value, should raise ValueError

    with pytest.raises(ValueError):
        flow_instance.update_context(
            None, "value"
        )  # None key, should raise ValueError

    with pytest.raises(ValueError):
        flow_instance.update_context(
            "key", None
        )  # None value, should raise ValueError


def test_flow_context_operations(flow_instance):
    # Test context operations
    flow_instance.add_context("user_id", "12345")
    assert flow_instance.get_context("user_id") == "12345"
    flow_instance.update_context("user_id", "54321")
    assert flow_instance.get_context("user_id") == "54321"
    flow_instance.remove_context("user_id")
    assert flow_instance.get_context("user_id") is None


# Add more test cases as needed to cover various aspects of your Agent class


def test_flow_long_messages(flow_instance):
    # Test handling of long messages
    long_message = "A" * 10000  # Create a very long message
    flow_instance.run(long_message)
    assert len(flow_instance.get_message_history()) == 1
    assert flow_instance.get_message_history()[0] == long_message


def test_flow_custom_response(flow_instance):
    # Test custom response generation
    def custom_response_generator(message):
        if message == "Hello":
            return "Hi there!"
        elif message == "How are you?":
            return "I'm doing well, thank you."
        else:
            return "I don't understand."

    flow_instance.set_response_generator(custom_response_generator)
    assert flow_instance.run("Hello") == "Hi there!"
    assert (
        flow_instance.run("How are you?")
        == "I'm doing well, thank you."
    )
    assert (
        flow_instance.run("What's your name?")
        == "I don't understand."
    )



def test_flow_message_validation(flow_instance):
    # Test message validation
    def custom_message_validator(message):
        return len(message) > 0  # Reject empty messages

    flow_instance.set_message_validator(custom_message_validator)

    assert flow_instance.run("Valid message") is not None
    assert (
        flow_instance.run("") is None
    )  # Empty message should be rejected
    assert (
        flow_instance.run(None) is None
    )  # None message should be rejected
```

```
def test_flow_custom_logging(flow_instance):
    custom_logger = logger
    flow_instance.set_logger(custom_logger)

    with mock.patch.object(custom_logger, "log") as mock_log:
        flow_instance.run("Message")
        mock_log.assert_called_once_with("Message")


def test_flow_performance(flow_instance):
    # Test the performance of the Agent class by running a large number of messages
    num_messages = 1000
    for i in range(num_messages):
        flow_instance.run(f"Message {i}")
    assert len(flow_instance.get_message_history()) == num_messages


def test_flow_complex_use_case(flow_instance):
    # Test a complex use case scenario
    flow_instance.add_context("user_id", "12345")
    flow_instance.run("Hello")
    flow_instance.run("How can I help you?")
    assert (
        flow_instance.get_response() == "Please provide more details."
    )
    flow_instance.update_context("user_id", "54321")
    flow_instance.run("I need help with my order")
    assert (
        flow_instance.get_response()
        == "Sure, I can assist with that."
    )
    flow_instance.reset_conversation()
    assert len(flow_instance.get_message_history()) == 0
    assert flow_instance.get_context("user_id") is None


# Add more test cases as needed to cover various aspects of your Agent class
def test_flow_context_handling(flow_instance):
    # Test context handling
    flow_instance.add_context("user_id", "12345")
    assert flow_instance.get_context("user_id") == "12345"
    flow_instance.update_context("user_id", "54321")
    assert flow_instance.get_context("user_id") == "54321"
    flow_instance.remove_context("user_id")
    assert flow_instance.get_context("user_id") is None


def test_flow_concurrent_requests(flow_instance):
    # Test concurrent message processing
    import threading
    def send_messages():
        for i in range(100):
            flow_instance.run(f"Message {i}")

    threads = []
    for _ in range(5):
        thread = threading.Thread(target=send_messages)
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    assert len(flow_instance.get_message_history()) == 500


def test_flow_custom_timeout(flow_instance):
    # Test custom timeout handling
    flow_instance.set_timeout(
        10
    )  # Set a custom timeout of 10 seconds
    assert flow_instance.get_timeout() == 10

    import time
    start_time = time.time()
    flow_instance.run("Long-running operation")
    end_time = time.time()
    execution_time = end_time - start_time
    assert execution_time >= 10  # Ensure the timeout was respected


# Add more test cases as needed to thoroughly cover your Agent class


def test_flow_interactive_run(flow_instance, capsys):
    # Test interactive run mode
    # Simulate user input and check if the AI responds correctly
    user_input = ["Hello", "How can you help me?", "Exit"]

    def simulate_user_input(input_list):
        input_index = 0
        while input_index < len(input_list):
            user_response = input_list[input_index]
            flow_instance.interactive_run(max_loops=1)

            # Capture the AI's response
            captured = capsys.readouterr()
            ai_response = captured.out.strip()

            assert f"You: {user_response}" in captured.out
            assert "AI:" in captured.out

            # Check if the AI's response matches the expected response
            expected_response = f"AI: {ai_response}"
            assert expected_response in captured.out

            input_index += 1

    simulate_user_input(user_input)


# Assuming you have already defined your Agent class and created an instance for testing


def test_flow_agent_history_prompt(flow_instance):
    # Test agent history prompt generation
    system_prompt = "This is the system prompt."
    history = ["User: Hi", "AI: Hello"]

    agent_history_prompt = flow_instance.agent_history_prompt(
        system_prompt, history
    )

    assert (
        "SYSTEM_PROMPT: This is the system prompt."
        in agent_history_prompt
    )
    assert (
        "History: ['User: Hi', 'AI: Hello']" in agent_history_prompt
    )


async def test_flow_run_concurrent(flow_instance):
    # Test running tasks concurrently
    tasks = ["Task 1", "Task 2", "Task 3"]
    completed_tasks = await flow_instance.run_concurrent(tasks)

    # Ensure that all tasks are completed
    assert len(completed_tasks) == len(tasks)


def test_flow_bulk_run(flow_instance):
    # Test bulk running of tasks
    input_data = [
        {"task": "Task 1", "param1": "value1"},
        {"task": "Task 2", "param2": "value2"},
        {"task": "Task 3", "param3": "value3"},
    ]
    responses = flow_instance.bulk_run(input_data)

    # Ensure that the responses match the input tasks
    assert responses[0] == "Response for Task 1"
    assert responses[1] == "Response for Task 2"
    assert responses[2] == "Response for Task 3"


def test_flow_from_llm_and_template():
    # Test creating Agent instance from an LLM and a template
    llm_instance = mocked_llm  # Replace with your LLM class
    template = "This is a template for testing."

    flow_instance = Agent.from_llm_and_template(
        llm_instance, template
    )

    assert isinstance(flow_instance, Agent)


def test_flow_from_llm_and_template_file():
    # Test creating Agent instance from an LLM and a template file
    llm_instance = mocked_llm  # Replace with your LLM class
    template_file = (
        "template.txt"  # Create a template file for testing
    )

    flow_instance = Agent.from_llm_and_template_file(
        llm_instance, template_file
    )
    assert isinstance(flow_instance, Agent)



def test_flow_save_and_load(flow_instance, tmp_path):
    # Test saving and loading the agent state
    file_path = tmp_path / "flow_state.json"

    # Save the state
    flow_instance.save(file_path)

    # Create a new instance and load the state
    new_flow_instance = Agent(llm=mocked_llm, max_loops=5)
    new_flow_instance.load(file_path)

    # Ensure that the loaded state matches the original state
    assert new_flow_instance.memory == flow_instance.memory



def test_flow_validate_response(flow_instance):
    # Test response validation
    valid_response = "This is a valid response."
    invalid_response = "Short."

    assert flow_instance.validate_response(valid_response) is True
    assert flow_instance.validate_response(invalid_response) is False
```

```python
# Add more test cases as needed for other methods and features of your Agent class

# Finally, don't forget to run your tests using a testing framework like pytest

# Assuming you have already defined your Agent class and created an instance for testing


def test_flow_print_history_and_memory(capsys, flow_instance):
    # Test printing the history and memory of the agent
    history = ["User: Hi", "AI: Hello"]
    flow_instance.memory = [history]

    flow_instance.print_history_and_memory()

    captured = capsys.readouterr()
    assert "Agent History and Memory" in captured.out
    assert "Loop 1:" in captured.out
    assert "User: Hi" in captured.out
    assert "AI: Hello" in captured.out


def test_flow_run_with_timeout(flow_instance):
    # Test running with a timeout
    task = "Task with a long response time"
    response = flow_instance.run_with_timeout(task, timeout=1)

    # Ensure that the response is either the actual response or "Timeout"
    assert response in ["Actual Response", "Timeout"]


# Add more test cases as needed for other methods and features of your Agent class

# Finally, don't forget to run your tests using a testing framework like pytest
```