```
import asyncio

import json

import os

import tempfile

import time


import yaml

from swarm_models import OpenAIChat


from swarms import Agent




def test_basic_agent_functionality():

    """Test basic agent initialization and simple task execution"""

    print("\nTesting basic agent functionality...")


    model = OpenAIChat(model_name="gpt-4o")

    agent = Agent(agent_name="Test-Agent", llm=model, max_loops=1)


    response = agent.run("What is 2+2?")

    assert response is not None, "Agent response should not be None"


    # Test agent properties

    assert (

        agent.agent_name == "Test-Agent"

    ), "Agent name not set correctly"
    assert agent.max_loops == 1, "Max loops not set correctly"
    assert agent.llm is not None, "LLM not initialized"

    print(" Basic agent functionality test passed")


def test_memory_management():
    """Test agent memory management functionality"""
    print("\nTesting memory management...")

    model = OpenAIChat(model_name="gpt-4o")
    agent = Agent(
        agent_name="Memory-Test-Agent",
        llm=model,
        max_loops=1,
        context_length=8192,
    )

    # Test adding to memory
    agent.add_memory("Test memory entry")
    assert (
        "Test memory entry"
        in agent.short_memory.return_history_as_string()
    )

    # Test memory query
    agent.memory_query("Test query")

    # Test token counting
    tokens = agent.check_available_tokens()
    assert isinstance(tokens, int), "Token count should be an integer"

    print(" Memory management test passed")


def test_agent_output_formats():
    """Test all available output formats"""
    print("\nTesting all output formats...")

    model = OpenAIChat(model_name="gpt-4o")
    test_task = "Say hello!"

    output_types = {
        "str": str,
        "string": str,
        "list": str,  # JSON string containing list
        "json": str,  # JSON string
        "dict": dict,
        "yaml": str,
    }

    for output_type, expected_type in output_types.items():
```

```
agent = Agent(
    agent_name=f"{output_type.capitalize()}-Output-Agent",
    llm=model,
    max_loops=1,
    output_type=output_type,
)

response = agent.run(test_task)
assert (
    response is not None
), f"{output_type} output should not be None"

if output_type == "yaml":
    # Verify YAML can be parsed
    try:
        yaml.safe_load(response)
        print(f" {output_type} output valid")
    except yaml.YAMLError:
        assert False, f"Invalid YAML output for {output_type}"
elif output_type in ["json", "list"]:
    # Verify JSON can be parsed
    try:
        json.loads(response)
        print(f" {output_type} output valid")
    except json.JSONDecodeError:
        assert False, f"Invalid JSON output for {output_type}"
    print(" Output formats test passed")



def test_agent_state_management():
    """Test comprehensive state management functionality"""
    print("\nTesting state management...")

    model = OpenAIChat(model_name="gpt-4o")

    # Create temporary directory for test files
    with tempfile.TemporaryDirectory() as temp_dir:
        state_path = os.path.join(temp_dir, "agent_state.json")

        # Create agent with initial state
        agent1 = Agent(
            agent_name="State-Test-Agent",
            llm=model,
            max_loops=1,
            saved_state_path=state_path,
        )

        # Add some data to the agent
        agent1.run("Remember this: Test message 1")
        agent1.add_memory("Test message 2")
```

```
# Save state
agent1.save()
assert os.path.exists(state_path), "State file not created"

# Create new agent and load state
agent2 = Agent(
    agent_name="State-Test-Agent", llm=model, max_loops=1
)
agent2.load(state_path)

# Verify state loaded correctly
history2 = agent2.short_memory.return_history_as_string()
assert (
    "Test message 1" in history2
), "State not loaded correctly"
assert (
    "Test message 2" in history2
), "Memory not loaded correctly"

# Test autosave functionality
agent3 = Agent(
    agent_name="Autosave-Test-Agent",
    llm=model,
    max_loops=1,
    saved_state_path=os.path.join(
        temp_dir, "autosave_state.json"
```

```
            ),
            autosave=True,
        )

        agent3.run("Test autosave")
        time.sleep(2)  # Wait for autosave
        assert os.path.exists(
            os.path.join(temp_dir, "autosave_state.json")
        ), "Autosave file not created"

    print(" State management test passed")


def test_agent_tools_and_execution():
    """Test agent tool handling and execution"""
    print("\nTesting tools and execution...")

    def sample_tool(x: int, y: int) -> int:
        """Sample tool that adds two numbers"""
        return x + y

    model = OpenAIChat(model_name="gpt-4o")
    agent = Agent(
        agent_name="Tools-Test-Agent",
        llm=model,
        max_loops=1,
        tools=[sample_tool],
    )

    # Test adding tools
    agent.add_tool(lambda x: x * 2)
    assert len(agent.tools) == 2, "Tool not added correctly"

    # Test removing tools
    agent.remove_tool(sample_tool)
    assert len(agent.tools) == 1, "Tool not removed correctly"

    # Test tool execution
    response = agent.run("Calculate 2 + 2 using the sample tool")
    assert response is not None, "Tool execution failed"

    print(" Tools and execution test passed")


def test_agent_concurrent_execution():
    """Test agent concurrent execution capabilities"""
    print("\nTesting concurrent execution...")

    model = OpenAIChat(model_name="gpt-4o")
    agent = Agent(
        agent_name="Concurrent-Test-Agent", llm=model, max_loops=1
    )
    # Test bulk run
    tasks = [
        {"task": "Count to 3"},
        {"task": "Say hello"},
        {"task": "Tell a short joke"},
    ]

    responses = agent.bulk_run(tasks)
    assert len(responses) == len(tasks), "Not all tasks completed"
    assert all(
        response is not None for response in responses
    ), "Some tasks failed"

    # Test concurrent tasks
    concurrent_responses = agent.run_concurrent_tasks(
        ["Task 1", "Task 2", "Task 3"]
    )
    assert (
        len(concurrent_responses) == 3
    ), "Not all concurrent tasks completed"

    print(" Concurrent execution test passed")


def test_agent_error_handling():
```

```
"""Test agent error handling and recovery"""
print("\nTesting error handling...")

model = OpenAIChat(model_name="gpt-4o")
agent = Agent(
    agent_name="Error-Test-Agent",
    llm=model,
    max_loops=1,
    retry_attempts=3,
    retry_interval=1,
)

# Test invalid tool execution
try:
    agent.parse_and_execute_tools("invalid_json")
    print(" Invalid tool execution handled")
except Exception:
    assert True, "Expected error caught"

# Test recovery after error
response = agent.run("Continue after error")
assert response is not None, "Agent failed to recover after error"

print(" Error handling test passed")
```

```
def test_agent_configuration():
    """Test agent configuration and parameters"""
    print("\nTesting agent configuration...")

    model = OpenAIChat(model_name="gpt-4o")
    agent = Agent(
        agent_name="Config-Test-Agent",
        llm=model,
        max_loops=1,
        temperature=0.7,
        max_tokens=4000,
        context_length=8192,
    )

    # Test configuration methods
    agent.update_system_prompt("New system prompt")
    agent.update_max_loops(2)
    agent.update_loop_interval(2)

    # Verify updates
    assert agent.max_loops == 2, "Max loops not updated"
    assert agent.loop_interval == 2, "Loop interval not updated"

    # Test configuration export
    config_dict = agent.to_dict()
    assert isinstance(
        config_dict, dict
    ), "Configuration export failed"

    # Test YAML export
    yaml_config = agent.to_yaml()
    assert isinstance(yaml_config, str), "YAML export failed"

    print(" Configuration test passed")


def test_agent_with_stopping_condition():
    """Test agent with custom stopping condition"""
    print("\nTesting agent with stopping condition...")

    def custom_stopping_condition(response: str) -> bool:
        return "STOP" in response.upper()

    model = OpenAIChat(model_name="gpt-4o")
    agent = Agent(
        agent_name="Stopping-Condition-Agent",
        llm=model,
        max_loops=5,
        stopping_condition=custom_stopping_condition,
    )

    response = agent.run("Count up until you see the word STOP")
    assert response is not None, "Stopping condition test failed"
    print(" Stopping condition test passed")


def test_agent_with_retry_mechanism():
    """Test agent retry mechanism"""
    print("\nTesting agent retry mechanism...")

    model = OpenAIChat(model_name="gpt-4o")
    agent = Agent(
        agent_name="Retry-Test-Agent",
        llm=model,
        max_loops=1,
        retry_attempts=3,
        retry_interval=1,
    )

    response = agent.run("Tell me a joke.")
    assert response is not None, "Retry mechanism test failed"
    print(" Retry mechanism test passed")


def test_bulk_and_filtered_operations():
    """Test bulk operations and response filtering"""
    print("\nTesting bulk and filtered operations...")
```

```
model = OpenAIChat(model_name="gpt-4o")
agent = Agent(
    agent_name="Bulk-Filter-Test-Agent", llm=model, max_loops=1
)


# Test bulk run
bulk_tasks = [
    {"task": "What is 2+2?"},
    {"task": "Name a color"},
    {"task": "Count to 3"},
]
bulk_responses = agent.bulk_run(bulk_tasks)
assert len(bulk_responses) == len(
    bulk_tasks
), "Bulk run should return same number of responses as tasks"


# Test response filtering
agent.add_response_filter("color")
filtered_response = agent.filtered_run(
    "What is your favorite color?"
)
assert (
    "[FILTERED]" in filtered_response
), "Response filter not applied"


print(" Bulk and filtered operations test passed")
```

```
async def test_async_operations():
    """Test asynchronous operations"""
    print("\nTesting async operations...")

    model = OpenAIChat(model_name="gpt-4o")
    agent = Agent(
        agent_name="Async-Test-Agent", llm=model, max_loops=1
    )

    # Test single async run
    response = await agent.arun("What is 1+1?")
    assert response is not None, "Async run failed"

    # Test concurrent async runs
    tasks = ["Task 1", "Task 2", "Task 3"]
    responses = await asyncio.gather(
        *[agent.arun(task) for task in tasks]
    )
    assert len(responses) == len(
        tasks
    ), "Not all async tasks completed"

    print(" Async operations test passed")
```

```
def test_memory_and_state_persistence():
    """Test memory management and state persistence"""
    print("\nTesting memory and state persistence...")

    with tempfile.TemporaryDirectory() as temp_dir:
        state_path = os.path.join(temp_dir, "test_state.json")

        # Create agent with memory configuration
        model = OpenAIChat(model_name="gpt-4o")
        agent1 = Agent(
            agent_name="Memory-State-Test-Agent",
            llm=model,
            max_loops=1,
            saved_state_path=state_path,
            context_length=8192,
            autosave=True,
        )

        # Test memory operations
        agent1.add_memory("Important fact: The sky is blue")
        agent1.memory_query("What color is the sky?")

        # Save state
        agent1.save()
        # Create new agent and load state
        agent2 = Agent(
            agent_name="Memory-State-Test-Agent",
            llm=model,
            max_loops=1,
        )
        agent2.load(state_path)

        # Verify memory persistence
        memory_content = (
            agent2.short_memory.return_history_as_string()
        )
        assert (
            "sky is blue" in memory_content
        ), "Memory not properly persisted"

        print(" Memory and state persistence test passed")


def test_sentiment_and_evaluation():
    """Test sentiment analysis and response evaluation"""
    print("\nTesting sentiment analysis and evaluation...")

    def mock_sentiment_analyzer(text):
        """Mock sentiment analyzer that returns a score between 0 and 1"""
        return 0.7 if "positive" in text.lower() else 0.3
    def mock_evaluator(response):
        """Mock evaluator that checks response quality"""
        return "GOOD" if len(response) > 10 else "BAD"

    model = OpenAIChat(model_name="gpt-4o")
    agent = Agent(
        agent_name="Sentiment-Eval-Test-Agent",
        llm=model,
        max_loops=1,
        sentiment_analyzer=mock_sentiment_analyzer,
        sentiment_threshold=0.5,
        evaluator=mock_evaluator,
    )

    # Test sentiment analysis
    agent.run("Generate a positive message")

    # Test evaluation
    agent.run("Generate a detailed response")

    print(" Sentiment and evaluation test passed")


def test_tool_management():
    """Test tool management functionality"""
```

```
print("\nTesting tool management...")


def tool1(x: int) -> int:
    """Sample tool 1"""
    return x * 2


def tool2(x: int) -> int:
    """Sample tool 2"""
    return x + 2


model = OpenAIChat(model_name="gpt-4o")
agent = Agent(
    agent_name="Tool-Test-Agent",
    llm=model,
    max_loops=1,
    tools=[tool1],
)


# Test adding tools
agent.add_tool(tool2)
assert len(agent.tools) == 2, "Tool not added correctly"


# Test removing tools
agent.remove_tool(tool1)
assert len(agent.tools) == 1, "Tool not removed correctly"
```

```
    # Test adding multiple tools
    agent.add_tools([tool1, tool2])
    assert len(agent.tools) == 3, "Multiple tools not added correctly"

    print(" Tool management test passed")


def test_system_prompt_and_configuration():
    """Test system prompt and configuration updates"""
    print("\nTesting system prompt and configuration...")

    model = OpenAIChat(model_name="gpt-4o")
    agent = Agent(
        agent_name="Config-Test-Agent", llm=model, max_loops=1
    )

    # Test updating system prompt
    new_prompt = "You are a helpful assistant."
    agent.update_system_prompt(new_prompt)
    assert (
        agent.system_prompt == new_prompt
    ), "System prompt not updated"

    # Test configuration updates
    agent.update_max_loops(5)
    assert agent.max_loops == 5, "Max loops not updated"
    agent.update_loop_interval(2)
    assert agent.loop_interval == 2, "Loop interval not updated"

    # Test configuration export
    config_dict = agent.to_dict()
    assert isinstance(
        config_dict, dict
    ), "Configuration export failed"

    print(" System prompt and configuration test passed")


def test_agent_with_dynamic_temperature():
    """Test agent with dynamic temperature"""
    print("\nTesting agent with dynamic temperature...")

    model = OpenAIChat(model_name="gpt-4o")
    agent = Agent(
        agent_name="Dynamic-Temp-Agent",
        llm=model,
        max_loops=2,
        dynamic_temperature_enabled=True,
    )

    response = agent.run("Generate a creative story.")
    assert response is not None, "Dynamic temperature test failed"
    print(" Dynamic temperature test passed")


def run_all_tests():
    """Run all test functions"""
    print("Starting Extended Agent functional tests...\n")

    test_functions = [
        test_basic_agent_functionality,
        test_memory_management,
        test_agent_output_formats,
        test_agent_state_management,
        test_agent_tools_and_execution,
        test_agent_concurrent_execution,
        test_agent_error_handling,
        test_agent_configuration,
        test_agent_with_stopping_condition,
        test_agent_with_retry_mechanism,
        test_agent_with_dynamic_temperature,
        test_bulk_and_filtered_operations,
        test_memory_and_state_persistence,
        test_sentiment_and_evaluation,
        test_tool_management,
        test_system_prompt_and_configuration,
    ]
```

```
# Run synchronous tests
total_tests = len(test_functions) + 1  # +1 for async test
passed_tests = 0

for test in test_functions:
    try:
        test()
        passed_tests += 1
    except Exception as e:
        print(f" Test {test.__name__} failed: {str(e)}")

# Run async test
try:
    asyncio.run(test_async_operations())
    passed_tests += 1
except Exception as e:
    print(f" Async operations test failed: {str(e)}")

print("\nExtended Test Summary:")
print(f"Total Tests: {total_tests}")
print(f"Passed: {passed_tests}")
print(f"Failed: {total_tests - passed_tests}")
print(f"Success Rate: {(passed_tests/total_tests)*100:.2f}%")
```

```
if __name__ == "__main__":
    run_all_tests()
```