```
from unittest.mock import MagicMock

import pytest
from pydantic import BaseModel

from swarms.tools.tool import (
    BaseTool,
    Runnable,
    StructuredTool,
    Tool,
    tool,
)

# Define test data
test_input = {"key1": "value1", "key2": "value2"}
expected_output = "expected_output_value"

# Test with global variables
global_var = "global"


# Basic tests for BaseTool
def test_base_tool_init():
    # Test BaseTool initialization
    tool = BaseTool()
    assert isinstance(tool, BaseTool)
```

```
def test_base_tool_invoke():
    # Test BaseTool invoke method
    tool = BaseTool()
    result = tool.invoke(test_input)
    assert result == expected_output


# Basic tests for Tool
def test_tool_init():
    # Test Tool initialization
    tool = Tool()
    assert isinstance(tool, Tool)


def test_tool_invoke():
    # Test Tool invoke method
    tool = Tool()
    result = tool.invoke(test_input)
    assert result == expected_output


# Basic tests for StructuredTool
def test_structured_tool_init():
    # Test StructuredTool initialization
```

```
    tool = StructuredTool()
    assert isinstance(tool, StructuredTool)


def test_structured_tool_invoke():
    # Test StructuredTool invoke method
    tool = StructuredTool()
    result = tool.invoke(test_input)
    assert result == expected_output


# Test additional functionality and edge cases as needed


def test_tool_creation():
    tool = Tool(
        name="test_tool", func=lambda x: x, description="Test tool"
    )
    assert tool.name == "test_tool"
    assert tool.func is not None
    assert tool.description == "Test tool"


def test_tool_ainvoke():
    tool = Tool(
        name="test_tool", func=lambda x: x, description="Test tool"
```

```
    )
    result = tool.ainvoke("input_data")
    assert result == "input_data"


def test_tool_ainvoke_with_coroutine():
    async def async_function(input_data):
        return input_data

    tool = Tool(
        name="test_tool",
        coroutine=async_function,
        description="Test tool",
    )
    result = tool.ainvoke("input_data")
    assert result == "input_data"


def test_tool_args():
    def sample_function(input_data):
        return input_data

    tool = Tool(
        name="test_tool",
        func=sample_function,
        description="Test tool",
```

```
    )
    assert tool.args == {"tool_input": {"type": "string"}}


# Basic tests for StructuredTool class


def test_structured_tool_creation():
    class SampleArgsSchema:
        pass

    tool = StructuredTool(
        name="test_tool",
        func=lambda x: x,
        description="Test tool",
        args_schema=SampleArgsSchema,
    )
    assert tool.name == "test_tool"
    assert tool.func is not None
    assert tool.description == "Test tool"
    assert tool.args_schema == SampleArgsSchema


def test_structured_tool_ainvoke():
    class SampleArgsSchema:
        pass
```

```
    tool = StructuredTool(
        name="test_tool",
        func=lambda x: x,
        description="Test tool",
        args_schema=SampleArgsSchema,
    )
    result = tool.ainvoke({"tool_input": "input_data"})
    assert result == "input_data"


def test_structured_tool_ainvoke_with_coroutine():
    class SampleArgsSchema:
        pass

    async def async_function(input_data):
        return input_data

    tool = StructuredTool(
        name="test_tool",
        coroutine=async_function,
        description="Test tool",
        args_schema=SampleArgsSchema,
    )
    result = tool.ainvoke({"tool_input": "input_data"})
    assert result == "input_data"
```

```
def test_structured_tool_args():
    class SampleArgsSchema:
        pass

    def sample_function(input_data):
        return input_data

    tool = StructuredTool(
        name="test_tool",
        func=sample_function,
        description="Test tool",
        args_schema=SampleArgsSchema,
    )
    assert tool.args == {"tool_input": {"type": "string"}}


# Additional tests for exception handling


def test_tool_ainvoke_exception():
    tool = Tool(name="test_tool", func=None, description="Test tool")
    with pytest.raises(NotImplementedError):
        tool.ainvoke("input_data")
```

```
def test_tool_ainvoke_with_coroutine_exception():
    tool = Tool(
        name="test_tool", coroutine=None, description="Test tool"
    )
    with pytest.raises(NotImplementedError):
        tool.ainvoke("input_data")



def test_structured_tool_ainvoke_exception():
    class SampleArgsSchema:
        pass

    tool = StructuredTool(
        name="test_tool",
        func=None,
        description="Test tool",
        args_schema=SampleArgsSchema,
    )
    with pytest.raises(NotImplementedError):
        tool.ainvoke({"tool_input": "input_data"})



def test_structured_tool_ainvoke_with_coroutine_exception():
    class SampleArgsSchema:
        pass
```

```
    tool = StructuredTool(
        name="test_tool",
        coroutine=None,
        description="Test tool",
        args_schema=SampleArgsSchema,
    )
    with pytest.raises(NotImplementedError):
        tool.ainvoke({"tool_input": "input_data"})


def test_tool_description_not_provided():
    tool = Tool(name="test_tool", func=lambda x: x)
    assert tool.name == "test_tool"
    assert tool.func is not None
    assert tool.description == ""


def test_tool_invoke_with_callbacks():
    def sample_function(input_data, callbacks=None):
        if callbacks:
            callbacks.on_start()
            callbacks.on_finish()
        return input_data

    tool = Tool(name="test_tool", func=sample_function)
```

```
    callbacks = MagicMock()
    result = tool.invoke("input_data", callbacks=callbacks)
    assert result == "input_data"
    callbacks.on_start.assert_called_once()
    callbacks.on_finish.assert_called_once()


def test_tool_invoke_with_new_argument():
    def sample_function(input_data, callbacks=None):
        return input_data

    tool = Tool(name="test_tool", func=sample_function)
    result = tool.invoke("input_data", callbacks=None)
    assert result == "input_data"


def test_tool_ainvoke_with_new_argument():
    async def async_function(input_data, callbacks=None):
        return input_data

    tool = Tool(name="test_tool", coroutine=async_function)
    result = tool.ainvoke("input_data", callbacks=None)
    assert result == "input_data"


def test_tool_description_from_docstring():
```

```
    def sample_function(input_data):
        """Sample function docstring"""
        return input_data

    tool = Tool(name="test_tool", func=sample_function)
    assert tool.description == "Sample function docstring"


def test_tool_ainvoke_with_exceptions():
    async def async_function(input_data):
        raise ValueError("Test exception")

    tool = Tool(name="test_tool", coroutine=async_function)
    with pytest.raises(ValueError):
        tool.ainvoke("input_data")


# Additional tests for StructuredTool class


def test_structured_tool_infer_schema_false():
    def sample_function(input_data):
        return input_data

    tool = StructuredTool(
        name="test_tool",
```

```
        func=sample_function,
        args_schema=None,
        infer_schema=False,
    )
    assert tool.args_schema is None


def test_structured_tool_ainvoke_with_callbacks():
    class SampleArgsSchema:
        pass

    def sample_function(input_data, callbacks=None):
        if callbacks:
            callbacks.on_start()
            callbacks.on_finish()
        return input_data

    tool = StructuredTool(
        name="test_tool",
        func=sample_function,
        args_schema=SampleArgsSchema,
    )
    callbacks = MagicMock()
    result = tool.ainvoke(
        {"tool_input": "input_data"}, callbacks=callbacks
    )
```

```
    assert result == "input_data"
    callbacks.on_start.assert_called_once()
    callbacks.on_finish.assert_called_once()


def test_structured_tool_description_not_provided():
    class SampleArgsSchema:
        pass

    tool = StructuredTool(
        name="test_tool",
        func=lambda x: x,
        args_schema=SampleArgsSchema,
    )
    assert tool.name == "test_tool"
    assert tool.func is not None
    assert tool.description == ""


def test_structured_tool_args_schema():
    class SampleArgsSchema:
        pass

    def sample_function(input_data):
        return input_data
```

```
    tool = StructuredTool(
        name="test_tool",
        func=sample_function,
        args_schema=SampleArgsSchema,
    )
    assert tool.args_schema == SampleArgsSchema


def test_structured_tool_args_schema_inference():
    def sample_function(input_data):
        return input_data

    tool = StructuredTool(
        name="test_tool",
        func=sample_function,
        args_schema=None,
        infer_schema=True,
    )
    assert tool.args_schema is not None


def test_structured_tool_ainvoke_with_new_argument():
    class SampleArgsSchema:
        pass

    def sample_function(input_data, callbacks=None):
```

```
        return input_data

    tool = StructuredTool(
        name="test_tool",
        func=sample_function,
        args_schema=SampleArgsSchema,
    )
    result = tool.ainvoke(
        {"tool_input": "input_data"}, callbacks=None
    )
    assert result == "input_data"


def test_structured_tool_ainvoke_with_exceptions():
    class SampleArgsSchema:
        pass

    async def async_function(input_data):
        raise ValueError("Test exception")

    tool = StructuredTool(
        name="test_tool",
        coroutine=async_function,
        args_schema=SampleArgsSchema,
    )
    with pytest.raises(ValueError):
```

```
        tool.ainvoke({"tool_input": "input_data"})


def test_base_tool_verbose_logging(caplog):
    # Test verbose logging in BaseTool
    tool = BaseTool(verbose=True)
    result = tool.invoke(test_input)
    assert result == expected_output
    assert "Verbose logging" in caplog.text


def test_tool_exception_handling():
    # Test exception handling in Tool
    tool = Tool()
    with pytest.raises(Exception):
        tool.invoke(test_input, raise_exception=True)


def test_structured_tool_async_invoke():
    # Test asynchronous invoke in StructuredTool
    tool = StructuredTool()
    result = tool.ainvoke(test_input)
    assert result == expected_output


# Add more tests for specific functionalities and edge cases as needed
```

```
# Import necessary libraries and modules



# Example of a mock function to be used in testing
def mock_function(arg: str) -> str:
    """A simple mock function for testing."""
    return f"Processed {arg}"



# Example of a Runnable class for testing
class MockRunnable(Runnable):
    # Define necessary methods and properties
    pass



# Fixture for creating a mock function
@pytest.fixture
def mock_func():
    return mock_function



# Fixture for creating a Runnable instance
@pytest.fixture
def mock_runnable():
    return MockRunnable()
```

```
# Basic functionality tests
def test_tool_with_callable(mock_func):
    # Test creating a tool with a simple callable
    tool_instance = tool(mock_func)
    assert isinstance(tool_instance, BaseTool)


def test_tool_with_runnable(mock_runnable):
    # Test creating a tool with a Runnable instance
    tool_instance = tool(mock_runnable)
    assert isinstance(tool_instance, BaseTool)


# ... more basic functionality tests ...


# Argument handling tests
def test_tool_with_invalid_argument():
    # Test passing an invalid argument type
    with pytest.raises(ValueError):
        tool(
            123
        )  # Using an integer instead of a string/callable/Runnable
```

```
def test_tool_with_multiple_arguments(mock_func):
    # Test passing multiple valid arguments
    tool_instance = tool("mock", mock_func)
    assert isinstance(tool_instance, BaseTool)


# ... more argument handling tests ...


# Schema inference and application tests
class TestSchema(BaseModel):
    arg: str


def test_tool_with_args_schema(mock_func):
    # Test passing a custom args_schema
    tool_instance = tool(mock_func, args_schema=TestSchema)
    assert tool_instance.args_schema == TestSchema


# ... more schema tests ...


# Exception handling tests
def test_tool_function_without_docstring():
    # Test that a ValueError is raised if the function lacks a docstring
```

```
    def no_doc_func(arg: str) -> str:
        return arg

    with pytest.raises(ValueError):
        tool(no_doc_func)



# Test suite starts here
class TestTool:
    # Basic Functionality Tests
    def test_tool_with_valid_callable_creates_base_tool(
        self, mock_func
    ):
        result = tool(mock_func)
        assert isinstance(result, BaseTool)

    def test_tool_returns_correct_function_name(self, mock_func):
        result = tool(mock_func)
        assert result.func.__name__ == "mock_function"

    # Argument Handling Tests
    def test_tool_with_string_and_runnable(self, mock_runnable):
        result = tool("mock_runnable", mock_runnable)
        assert isinstance(result, BaseTool)

    def test_tool_raises_error_with_invalid_arguments(self):
```

```
    with pytest.raises(ValueError):
        tool(123)


def test_tool_with_infer_schema_true(self, mock_func):
    tool(mock_func, infer_schema=True)
    # Assertions related to schema inference


# Return Direct Feature Tests
def test_tool_with_return_direct_true(self, mock_func):
    tool(mock_func, return_direct=True)
    # Assertions for return_direct behavior


# Error Handling Tests
def test_tool_raises_error_without_docstring(self):
    def no_doc_func(arg: str) -> str:
        return arg

    with pytest.raises(ValueError):
        tool(no_doc_func)


def test_tool_raises_error_runnable_without_object_schema(
    self, mock_runnable
):
    with pytest.raises(ValueError):
        tool(mock_runnable)
```

```
# Decorator Behavior Tests
@pytest.mark.asyncio
async def test_async_tool_function(self):
    @tool
    async def async_func(arg: str) -> str:
        return arg

    # Assertions for async behavior

# Integration with StructuredTool and Tool Classes
def test_integration_with_structured_tool(self, mock_func):
    result = tool(mock_func)
    assert isinstance(result, StructuredTool)

# Concurrency and Async Handling Tests
def test_concurrency_in_tool(self, mock_func):
    # Test related to concurrency
    pass

# Mocking and Isolation Tests
def test_mocking_external_dependencies(self, mocker):
    # Use mocker to mock external dependencies
    pass

def test_tool_with_different_return_types(self):
    @tool
```

```
    def return_int(arg: str) -> int:
        return int(arg)

    result = return_int("123")
    assert isinstance(result, int)
    assert result == 123

    @tool
    def return_bool(arg: str) -> bool:
        return arg.lower() in ["true", "yes"]

    result = return_bool("true")
    assert isinstance(result, bool)
    assert result is True

# Test with multiple arguments
def test_tool_with_multiple_args(self):
    @tool
    def concat_strings(a: str, b: str) -> str:
        return a + b

    result = concat_strings("Hello", "World")
    assert result == "HelloWorld"

# Test handling of optional arguments
def test_tool_with_optional_args(self):
```

```
    @tool
    def greet(name: str, greeting: str = "Hello") -> str:
        return f"{greeting} {name}"

    assert greet("Alice") == "Hello Alice"
    assert greet("Alice", greeting="Hi") == "Hi Alice"

# Test with variadic arguments
def test_tool_with_variadic_args(self):
    @tool
    def sum_numbers(*numbers: int) -> int:
        return sum(numbers)

    assert sum_numbers(1, 2, 3) == 6
    assert sum_numbers(10, 20) == 30

# Test with keyword arguments
def test_tool_with_kwargs(self):
    @tool
    def build_query(**kwargs) -> str:
        return "&".join(f"{k}={v}" for k, v in kwargs.items())

    assert build_query(a=1, b=2) == "a=1&b=2"
    assert build_query(foo="bar") == "foo=bar"

# Test with mixed types of arguments
```

```
def test_tool_with_mixed_args(self):
    @tool
    def mixed_args(a: int, b: str, *args, **kwargs) -> str:
        return f"{a}{b}{len(args)}{'-'.join(kwargs.values())}"

    assert mixed_args(1, "b", "c", "d", x="y", z="w") == "1b2y-w"

# Test error handling with incorrect types
def test_tool_error_with_incorrect_types(self):
    @tool
    def add_numbers(a: int, b: int) -> int:
        return a + b

    with pytest.raises(TypeError):
        add_numbers("1", "2")

# Test with nested tools
def test_nested_tools(self):
    @tool
    def inner_tool(arg: str) -> str:
        return f"Inner {arg}"

    @tool
    def outer_tool(arg: str) -> str:
        return f"Outer {inner_tool(arg)}"
```

```
    assert outer_tool("Test") == "Outer Inner Test"

def test_tool_with_global_variable(self):
    @tool
    def access_global(arg: str) -> str:
        return f"{global_var} {arg}"

    assert access_global("Var") == "global Var"

# Test with environment variables
def test_tool_with_env_variables(self, monkeypatch):
    monkeypatch.setenv("TEST_VAR", "Environment")

    @tool
    def access_env_variable(arg: str) -> str:
        import os

        return f"{os.environ['TEST_VAR']} {arg}"

    assert access_env_variable("Var") == "Environment Var"

# ... [Previous test cases] ...

# Test with complex data structures
def test_tool_with_complex_data_structures(self):
    @tool
```

```
    def process_data(data: dict) -> list:
        return [data[key] for key in sorted(data.keys())]

    result = process_data({"b": 2, "a": 1})
    assert result == [1, 2]

# Test handling exceptions within the tool function
def test_tool_handling_internal_exceptions(self):
    @tool
    def function_that_raises(arg: str):
        if arg == "error":
            raise ValueError("Error occurred")
        return arg

    with pytest.raises(ValueError):
        function_that_raises("error")
    assert function_that_raises("ok") == "ok"

# Test with functions returning None
def test_tool_with_none_return(self):
    @tool
    def return_none(arg: str):
        return None

    assert return_none("anything") is None
```

```
# Test with lambda functions
def test_tool_with_lambda(self):
    tool_lambda = tool(lambda x: x * 2)
    assert tool_lambda(3) == 6

# Test with class methods
def test_tool_with_class_method(self):
    class MyClass:
        @tool
        def method(self, arg: str) -> str:
            return f"Method {arg}"

    obj = MyClass()
    assert obj.method("test") == "Method test"

# Test tool function with inheritance
def test_tool_with_inheritance(self):
    class Parent:
        @tool
        def parent_method(self, arg: str) -> str:
            return f"Parent {arg}"

    class Child(Parent):
        @tool
        def child_method(self, arg: str) -> str:
            return f"Child {arg}"
```

```
    child_obj = Child()
    assert child_obj.parent_method("test") == "Parent test"
    assert child_obj.child_method("test") == "Child test"

# Test with decorators stacking
def test_tool_with_multiple_decorators(self):
    def another_decorator(func):
        def wrapper(*args, **kwargs):
            return f"Decorated {func(*args, **kwargs)}"

        return wrapper

    @tool
    @another_decorator
    def decorated_function(arg: str):
        return f"Function {arg}"

    assert decorated_function("test") == "Decorated Function test"

# Test tool function when used in a multi-threaded environment
def test_tool_in_multithreaded_environment(self):
    import threading

    @tool
    def threaded_function(arg: int) -> int:
```

```
        return arg * 2

    results = []

    def thread_target():
        results.append(threaded_function(5))

    threads = [
        threading.Thread(target=thread_target) for _ in range(10)
    ]
    for t in threads:
        t.start()
    for t in threads:
        t.join()

    assert results == [10] * 10

# Test with recursive functions
def test_tool_with_recursive_function(self):
    @tool
    def recursive_function(n: int) -> int:
        if n == 0:
            return 0
        else:
            return n + recursive_function(n - 1)
```

```
    assert recursive_function(5) == 15



# Additional tests can be added here to cover more scenarios
```