```
import time
import tracemalloc
from functools import wraps
from typing import Any, Callable

import psutil
from pydantic import BaseModel

from swarms.utils.loguru_logger import initialize_logger

logger = initialize_logger(log_folder="calculate_func_metrics")


class FunctionMetrics(BaseModel):
    execution_time: float
    memory_usage: float
    cpu_usage: float
    io_operations: int
    function_calls: int


def profile_func(func):
    """
    Decorator function that profiles the execution of a given function.

    Args:
```

```
        func: The function to be profiled.

    Returns:
        A wrapper function that profiles the execution of the given function and returns the result along with the metrics.

    """

    def wrapper(*args, **kwargs):
        # Record the initial time, memory usage, CPU usage, and I/O operations
        start_time = time.time()
        start_mem = psutil.Process().memory_info().rss
        start_cpu = psutil.cpu_percent()
        start_io = (
            psutil.disk_io_counters().read_count
            + psutil.disk_io_counters().write_count
        )

        # Call the function
        result = func(*args, **kwargs)

        # Record the final time, memory usage, CPU usage, and I/O operations
        end_time = time.time()
        end_mem = psutil.Process().memory_info().rss
        end_cpu = psutil.cpu_percent()
        end_io = (
```

```
    psutil.disk_io_counters().read_count
    + psutil.disk_io_counters().write_count
)

# Calculate the execution time, memory usage, CPU usage, and I/O operations
execution_time = end_time - start_time
memory_usage = (end_mem - start_mem) / (
    1024**2
)  # Convert bytes to MiB
cpu_usage = end_cpu - start_cpu
io_operations = end_io - start_io

# Return the metrics as a FunctionMetrics object
metrics = FunctionMetrics(
    execution_time=execution_time,
    memory_usage=memory_usage,
    cpu_usage=cpu_usage,
    io_operations=io_operations,
    function_calls=1,  # Each call to the function counts as one function call
)

json_data = metrics.model_dump_json(indent=4)

logger.info(f"Function metrics: {json_data}")

return result, metrics
```

```
    return wrapper



def profile_all(func: Callable) -> Callable:
    """
    A decorator to profile memory usage, CPU usage, and I/O operations
    of a function and log the data using loguru.

    It combines tracemalloc for memory profiling, psutil for CPU and I/O operations,
    and measures execution time.

    Args:
        func (Callable): The function to be profiled.

    Returns:
        Callable: The wrapped function with profiling enabled.
    """

    @wraps(func)
    def wrapper(*args: Any, **kwargs: Any) -> Any:
        # Start memory tracking
        tracemalloc.start()

        # Get initial CPU stats
        process = psutil.Process()
```

```
initial_cpu_times = process.cpu_times()

# Get initial I/O stats if available
try:
    initial_io_counters = process.io_counters()
    io_tracking_available = True
except AttributeError:
    logger.warning(
        "I/O counters not available on this platform."
    )
    io_tracking_available = False

# Start timing the function execution
start_time = time.time()

# Execute the function
result = func(*args, **kwargs)

# Stop timing
end_time = time.time()
execution_time = end_time - start_time

# Get final CPU stats
final_cpu_times = process.cpu_times()

# Get final I/O stats if available
```

```
if io_tracking_available:
    final_io_counters = process.io_counters()
    io_read_count = (
        final_io_counters.read_count
        - initial_io_counters.read_count
    )
    io_write_count = (
        final_io_counters.write_count
        - initial_io_counters.write_count
    )
else:
    io_read_count = io_write_count = 0

# Get memory usage statistics
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics("lineno")

# Calculate CPU usage
cpu_usage = (
    final_cpu_times.user
    - initial_cpu_times.user
    + final_cpu_times.system
    - initial_cpu_times.system
)

# Log the data
```

```
    logger.info(f"Execution time: {execution_time:.4f} seconds")
    logger.info(f"CPU usage: {cpu_usage:.2f} seconds")
    if io_tracking_available:
        logger.info(
            f"I/O Operations - Read: {io_read_count}, Write: {io_write_count}"
        )
    logger.info("Top memory usage:")
    for stat in top_stats[:10]:
        logger.info(stat)

    # Stop memory tracking
    tracemalloc.stop()

    return result

return wrapper
```