**Effectiveness of Algorithmic Optimizations:**

In developing this benchmark, I focused my algorithmic efforts on prioritization and stable failure management. For prioritization, the main optimization is the use of the sorted() algorithm within getCpuStatistics. This method efficiently sorts all system processes by their cpu\_percent utilization in descending order. The optimization is highly effective, since it instantly and accurately identifies and displays the top 5 resource consumers, allowing the user to diagnose the system bottlenecks without screening through unnecessary data. This fulfils the primary monitoring goal of the application.

Regarding failure states, the use of try…except blocks was implemented as a crucial flow control optimization to ensure stability. By implementing specific checks, such as catching the ZeroDivisionError in the demo function and handling environment errors like curses.error during startup, I prevented the program from crashing. Instead, execution is smoothly handled to a graceful failure state where a message is displayed. This mechanism is an essential optimization for the reliability of the terminal user interface application.

**OS Performance Metrics:**

My approach to monitoring system efficiency relies on the OS performance metrics collected in real-time using the psutil library, focusing on the three main system components: the CPU, memory, and disk. For CPU monitoring, I track both overall usage and per-core usage to understand how hard the processor is working and whether its load is evenly distributed. A critical feature is the use of sorting to immediately identify the top 5 processes by CPU percentage, which is essential since it flags the exact applications causing performance issues. For memory (RAM) management, I focus on the virtual memory usage percentage. This is important because if the percentage gets too high, the system starts using the hard drive as a slow-speed memory swap, causing a hit in performance. Finally, Disk efficiency is monitored using the disk usage percentage, which helps prevent system slowdowns and I/O issues that occur when the main storage is nearing its capacity. Together, these metrics provide a clear, real-time picture of system health, allowing for proactive maintenance and easy diagnosis of performance problems in the OS.

**Optimization Algorithm for Resource Management:**

The key optimization algorithm leveraged for scheduling and managing tasks is the prioritization model of the operating system’s thread scheduler, which I specifically control using the thread.join() method. This method prioritizes correctness and predictable output over speed. Using join() in the threading demonstration is essentially giving the main program a clear instruction to wait and not proceed further until similar tasks have finished. This is vital, since it guarantees the shared log is fully populated before the results are displayed, preventing the program from showing incomplete or incorrect data. In addition, the use of multiple threads is an efficient scheduling technique in itself, allowing concurrency to take place, which is a lot faster and better optimized in total execution time.

**Techniques for Secure and Stable Memory Management:**

The memory management techniques employed in this project prioritize stability and controlled resource access over complex allocation algorithms handled by the operating system itself. A key technique involves isolation through threading in the showThreadingExample function, where reliance on Python’s Global Interpreter Lock (GIL) ([python.org](http://python.org), 2025) acts as a security measure. Since the GIL only ensures one thread executes at a time, it effectively prevents classic memory corruption issues like race conditions when multiple threads concurrently write to the shared threadLog list. This controlled access maintains data integrity and system stability. In addition, stability is further improved with the use of proactive bound checking with Python’s curses across the application, with allocated window memory area management. Especially when resizing the terminal window without crashing ([python.org](http://python.org), 2025).

**Resource Allocation Techniques for OS Performance:**

The project optimizes OS performance by employing techniques that leverage and coordinate the operating system’s inherent resource management capabilities, which primarily focus on CPU time allocation through threading ([python.org](http://python.org), 2025).

The main technique is a concurrency control mechanism in the threading module. From the threading demonstration, tasks are divided into concurrent threads, where the operating system is instructed to schedule and allocate CPU time to these tasks simultaneously, therefore optimizing the overall execution time compared to running them sequentially. This is all coordinated by using the thread.join() method, which forces the main thread to yield its CPU time allocation until the concurrent tasks are fully complete. This ensures that the limited CPU resources are efficiently used for task completion before the program moves on.

For memory resource management, the application relies on and emphasizes prompt resource deallocation. While the operating system handles the actual memory allocation, the functions depend on system calls like those from psutil and os.stat to efficiently allocate and immediately free memory. All ensuring that memory resources aren’t unnecessarily retained and remain available for optimal system operations.

**References:**

[python.org](http://python.org) (2025), threading — Thread-based parallelism

<https://docs.python.org/3/library/threading.html>

[python.org](http://python.org) (2025), curses — Terminal handling for character-cell displays

<https://docs.python.org/3/library/curses.html>