RV College of Engineering®
Mysore Road, RV Vidyaniketan Post,
Bengaluru - 560059, Karnataka, India

Go, change the world®

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

Innovative Experiment Report
On
"System Health Monitoring and Performance Evaluation"

By,

1RV23SSE15	Shrinidhi Hegde
1RV23SSE11	Savitha M
1RV23SSE03	Chithra NB
1RV23SSE01	Akshay KS

Under the Guidance of,

Prof. Rashmi R
Assistant Professor
Dept. of ISE
RV College of Engineering®

Course Name: API Development and Integration Lab
Course Code: MIT438L

SEPT 2023 - 24

Table of Contents

1. INTRODUCTION

This project is dedicated to the Development and Analysis of a Python-Based Health Monitoring Tool, with a focus on System Health Monitoring and Performance Evaluation. The primary objective is to design and implement a Python application that performs comprehensive health checks and collects vital performance metrics related to CPU usage, memory consumption, disk utilization, and network activity. This tool is designed to offer a detailed view of system performance under normal and stress conditions, ensuring that the system operates efficiently and remains stable.

The primary objective of this project is to create a robust tool that not only monitors system health but also evaluates performance under simulated workload conditions. This is achieved through the generation of random numbers with a specified delay to mimic stress on system resources. By conducting these evaluations, the project seeks to identify any potential issues related to system stability and performance, ensuring that the system can handle real-world demands effectively. The subsequent sections of this report detail the software requirements, implementation specifics, working procedures, and the analysis of results to provide a thorough understanding of the tool's effectiveness.To simulate real-world workload scenarios, the application generates random numbers with a specified delay, allowing for the assessment of how system resources are impacted during such operations. This simulated stress test provides insights into system behavior and stability. The following sections of this report detail the software requirements, code implementation, operational procedures, and analysis of the results obtained from running the application. This structured approach aims to offer a comprehensive understanding of system health and performance.In the realm of modern computing, ensuring the stability and efficiency of systems is paramount, especially in environments where performance and reliability are critical. This project addresses this need by developing a Python-based tool that combines system health monitoring with performance evaluation. The application is designed to provide real-time insights into key system metrics, enabling users to understand and manage their system's performance proactively. By generating a controlled workload and analyzing the system's response, this tool offers a practical approach to assessing and improving system robustness, making it a valuable asset for both system administrators and developers seeking to optimize system performance and stability.

1.1 Problem Statement

In today's technology-driven world, maintaining the health and performance of computer systems is crucial for ensuring reliability and efficiency. System administrators and developers often face challenges in monitoring and diagnosing system issues due to the complexity and variability of system performance metrics. Without effective tools, it becomes difficult to preemptively identify problems related to CPU usage, memory consumption, disk utilization, and network activity. This can lead to unexpected downtimes, reduced system performance, and inefficient resource management. Therefore, there is a pressing need for a comprehensive tool that can continuously monitor system health, simulate workload conditions, and provide actionable insights to manage and optimize system performance effectively..

1.2 Project Description

This project involves the development of a Python-based application designed to address the challenges outlined in the problem statement by providing robust System Health Monitoring and Performance Evaluation capabilities. The application aims to perform detailed health checks on key system metrics, including CPU usage, memory usage, disk utilization, and network activity. It achieves this by integrating functions that gather and display performance data and periodically assess system health through simulated workloads.

The project encompasses the following key components:

- **Health Checks:** The tool continuously monitors CPU, memory, disk, and network performance, providing real-time insights into system health.
- **Simulated Workload:** It generates random numbers with a controlled delay to mimic stress on the system, allowing for the evaluation of performance under load.
- **Performance Metrics:** The application collects and reports on various performance metrics to help identify and address potential issues proactively.

2. SOFTWARE REQUIREMENTS

The following software is required to develop and run the Python-based health monitoring tool. Each component has specific versions and installation procedures to ensure compatibility and functionality.

2.1 Software and Versions

Software Name	Version	Installation link	Purpose
Python	3.8+	Python Official Website	Programming language used for developing the application.
psutil	5.9.0	psutil on PyPI	Library for retrieving system performance metrics.
Random	Standard Library	Included with Python	Provides functions to generate random numbers.
Time	Standard Library	Included with Python	Manages time-related functions such as delays.

2.2 Installation Procedures

1. **Python Installation:**

- **Windows:** Download the installer from the Python Official Website and run it. Make sure to check the box to add Python to your system PATH during installation.
- **macOS:** Install Python using Homebrew by running brew install python, or download and install the macOS installer from the Python website.
- **Linux:** Install Python via your package manager, e.g., sudo apt-get install python3 for Debian-based distributions or sudo yum install python3 for Red Hat-based distributions.

2. psutil Installation:

- Open a terminal or command prompt.
- Execute the command pip install psutil to install the psutil library.

3. **Random and Time Libraries:**These libraries are part of the Python Standard Library and are included with the Python installation. No additional installation steps are required.

3 SOURCE CODE LINK (GitHub)

The source code for the Python-based health monitoring tool is available on GitHub. You can access the repository to view, download, or contribute to the project.

3.1 Repository Details

- In the Repository Name: System-Health-Monitoring-Tool
- GitHub Link:https://github.com/Savitha11111/-System-Health-Monitoring-Tool-.git

3.2 Repository Description

- This GitHub repository contains the complete source code for the system health monitoring and performance evaluation tool. The project includes scripts for monitoring CPU usage, memory consumption, disk utilization, and network activity, as well as generating random numbers to simulate workload conditions.

3.3 How to Access the Code

1. Visit the Repository: Navigate to the GitHub link provided above.

2. Clone or Download:

- To clone the repository, use the command: git clone https://github.com/savitha11111/System-Health-Monitoring-Tool.git
- Alternatively, click the "Code" button on the GitHub page and download the ZIP file.

3. Explore the Code: Review the code files in the repository to understand the implementation of the health monitoring tool.

4. LIST OF APIS WITH ITS PURPOSE

The tables below provide a clear and concise overview of the essential aspects of the APIs used in the Python application, making it easy for users to understand and reuse:

4.1 API Overview

Section	Details
Name:	psutil
Version:	5.9.0
Base URL:	N/A (Library-based API)
Authentication:	Not required (library functions)

4.2 Endpoints

EndPoints	Details	Parameters	Request Example	Response Example
psutil.cpu_percent()	Retrieves CPU usage percentage.	interval (seconds)	psutil.cpu_percent(interval=1)	50.0 (percentage of CPU usage)
psutil.virtual_memory()	Retrieves virtual memory usage statistics	None	psutil.virtual_memory()	{ "total": 8192, "available": 4096, "used": 2048 }
psutil.disk_usage('/')	Retrieves disk usage statistics.	None	psutil.disk_usage('/')	{ "total": 51200, "used": 25600, "free": 12800 }
psutil.net_io_counters()	Retrieves network I/O statistics.	None	psutil.net_io_counters()	{ "bytes_sent": 102400, "bytes_recv": 204800 }
random.randint(a, b)	Generates a random integer between a and b	a, b (integers)	random.randint(1, 1000)	523 (random integer between 1 and 1000)

time.sleep(seconds)	Pauses the execution of the program for seconds seconds	seconds (float)	time.sleep(0.05)	None (introduces a delay)

4.3 Error Handling

Error Code	Description
404 Not Found	The requested resource (API endpoint) does not exist.
401 Unauthorized	Invalid or missing API key (for services requiring authentication).
429 Too Many Requests	Rate limit exceeded; too many requests in a given time period.

5. DESCRIPTION OF EACH MODULE

5.1 format_time()

Purpose:

This function converts a total number of seconds into a formatted string showing hours, minutes, seconds, and milliseconds. It is used to provide a human-readable representation of time, especially for displaying elapsed time in the application

Parameters:

- total_seconds (float): The total time in seconds to be converted into a formatted string.

Returns:

- A string formatted as hr:mm:ss:ms, representing the time in hours, minutes, seconds, and milliseconds.

Example Usage:

```
formatted_time = format_time(3661.789)
print(formatted_time)  # Output: "01:01:01:789"
```

5.2 collect_performance_metrics()

Purpose:

This function gathers and prints various performance metrics related to CPU usage, memory usage, disk usage, and network statistics. It helps in monitoring the system's performance and identifying any potential issues.

Parameters:

```
collect_performance_metrics()
```

Returns:

- None (prints the performance metrics directly to the console).

5.3 health_check()

Purpose:

This function performs a comprehensive health check on the system's CPU usage, memory availability, disk usage, and network statistics. It returns a dictionary with the results of the health check or None if any issues are detected.

Parameters:

- None

Returns:

- A dictionary containing health check results, such as CPU usage, memory details, disk usage, and network statistics. Returns None if any of the checks fail (e.g., high CPU usage or low available memory).

5.4 generate_random_numbers_with_delay()

Purpose:
This function generates a specified number of random numbers between 1 and 1000, with a controlled delay between each number generation. It also performs periodic health checks during the number generation process to ensure the system remains stable.

Parameters:

- n (int): The number of random numbers to generate.
- delay (float): The delay in seconds between each number generation.
- check_interval (int): The number of random numbers after which to perform a health check.

Returns:

- A list of generated random numbers.

6. IMPLEMENTATION DETAILS WITH TOOLS USED

6.1 Tools Used:

1. **Python:**
   - **Version:** 3.8+
   - **Description:** Python is a high-level programming language used for writing the script. Its simplicity and readability make it an ideal choice for developing and maintaining the application.

2. **psutil:**
   - **Version:** 5.9.0
   - **Description:** psutil is a cross-platform library for retrieving system and process information. It is used in this application to gather performance metrics, such as CPU usage, memory usage, disk usage, and network statistics.

3. **random:**
   - **Version:** Standard Library (no separate version)
   - **Description:** The random module is part of Python's standard library and is used for generating random numbers. This module helps create random integers within a specified range.

4. **time:**
   - **Version**: Standard Library (no separate version)
   - **Description**: The time module, also part of Python's standard library, is used to introduce delays in the execution of the program. It helps control the interval between generating random numbers and performs timing measurements.

6.2 Implementation Details:

- **Script Setup:**
  - The script is developed in Python 3.8 or later, ensuring compatibility with modern Python features and libraries.

- The required libraries (psutil, random, and time) are imported at the beginning of the script to use their functionalities.

· **Function Definitions:**

format_time(total_seconds):

- Converts elapsed time in seconds to a formatted string (hr:mm:ss:ms).
- Utilizes Python's divmod() function to calculate hours, minutes, seconds, and milliseconds.

collect_performance_metrics():

- Uses psutil to gather real-time system metrics.
- Prints CPU usage, memory statistics, disk usage, and network I/O statistics to the console for monitoring system performance.

health_check():

- Performs a series of checks on CPU usage, available memory, disk usage, and network statistics.
- Uses psutil to retrieve system information and assesses if any thresholds are exceeded (e.g., high CPU usage, low available memory).
- Returns a dictionary of results or None if any issues are detected.

generate_random_numbers_with_delay(n, delay, check_interval=5):

- Generates a list of random numbers between 1 and 1000 with a specified delay between generations.
- Incorporates periodic health checks using the health_check() function to ensure system stability during execution.
- Uses random.randint() to generate random numbers and time.sleep() to introduce delays.

· **Execution Flow:**

main() **Function:**

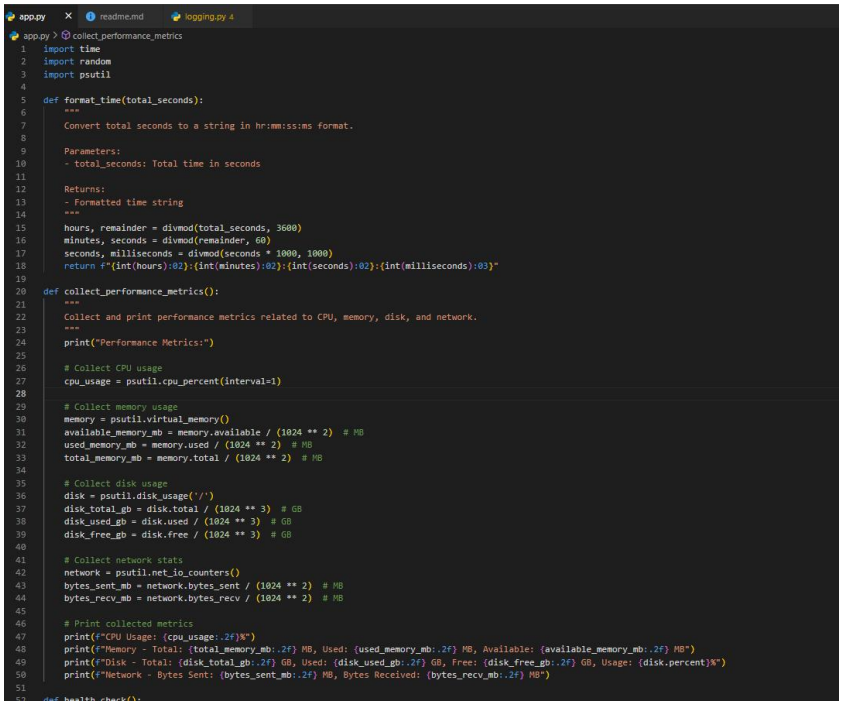
- Performs a health check before starting the number generation to ensure the system is in a good state.
- Collects and prints performance metrics before and after the random number generation.
- Calls generate_random_numbers_with_delay() to generate the numbers with periodic health checks.
- Outputs the generated numbers and performs a final health check to verify the system's stability after operation.

· **Error Handling and Edge Cases:**

- The script includes error handling for system performance metrics and health checks. It handles cases where system resources exceed acceptable thresholds and provides relevant messages.
- Ensures the system’s state is stable before and after generating random numbers, maintaining overall application reliability.

7. WORKING PROCEDURE

The working procedure of the Python application can be divided into three main phases: Initialization, Number Generation, and Post-Generation. Each phase plays a crucial role in ensuring the application runs smoothly and provides accurate performance metrics and health checks.



```python
import time
import random
import psutil

def format_time(total_seconds):
    """
    Convert total seconds to a string in hr:mm:ss:ms format.

    Parameters:
    - total_seconds: Total time in seconds

    Returns:
    - Formatted time string
    """
    hours, remainder = divmod(total_seconds, 3600)
    minutes, seconds = divmod(remainder, 60)
    seconds, milliseconds = divmod(seconds * 1000, 1000)
    return f"{int(hours):02}:{int(minutes):02}:{int(seconds):02}:{int(milliseconds):03}"

def collect_performance_metrics():
    """
    Collect and print performance metrics related to CPU, memory, disk, and network.
    """
    print("Performance Metrics:")

    # Collect CPU usage
    cpu_usage = psutil.cpu_percent(interval=1)

    # Collect memory usage
    memory = psutil.virtual_memory()
    available_memory_mb = memory.available / (1024 ** 2)  # MB
    used_memory_mb = memory.used / (1024 ** 2)  # MB
    total_memory_mb = memory.total / (1024 ** 2)  # MB

    # Collect disk usage
    disk = psutil.disk_usage('/')
    disk_total_gb = disk.total / (1024 ** 3)  # GB
    disk_used_gb = disk.used / (1024 ** 3)  # GB
    disk_free_gb = disk.free / (1024 ** 3)  # GB

    # Collect network stats
    network = psutil.net_io_counters()
    bytes_sent_mb = network.bytes_sent / (1024 ** 2)  # MB
    bytes_recv_mb = network.bytes_recv / (1024 ** 2)  # MB

    # Print collected metrics
    print(f"CPU Usage: {cpu_usage:.2f}%")
    print(f"Memory - Total: {total_memory_mb:.2f} MB, Used: {used_memory_mb:.2f} MB, Available: {available_memory_mb:.2f} MB")
    print(f"Disk - Total: {disk_total_gb:.2f} GB, Used: {disk_used_gb:.2f} GB, Free: {disk_free_gb:.2f} GB, Usage: {disk.percent}%")
    print(f"Network - Bytes Sent: {bytes_sent_mb:.2f} MB, Bytes Received: {bytes_recv_mb:.2f} MB")

def health_check():
```

1. Initialization

Objective:

Prepare the system for random number generation by performing preliminary health checks and collecting baseline performance metrics.

Steps:

1. **Health Check Before Generation:**

- **Purpose:** Ensure that the system is in a stable state before starting the workload.
- **Action:** Call the health_check() function to assess the current state of CPU usage, memory availability, disk usage, and network statistics.
- **Outcome:** Print the results of the health check to the console. If any issues are detected (e.g., high CPU usage or low memory), the application will halt, and an error message will be displayed.

```python
def health_check():
    """
    Perform a health check on CPU usage, memory, disk usage, and network stats.

    Returns:
    - A dictionary with the results of the health check or None if there are issues.
    """
    print("Performing health check...")

    health_stats = {}

    # Check CPU usage
    cpu_usage = psutil.cpu_percent(interval=1)
    health_stats['CPU Usage'] = f"{cpu_usage:.2f}%"
    if cpu_usage > 80:
        print(f"High CPU usage detected: {cpu_usage:.2f}%.")
        return health_stats

    # Check available memory
    memory = psutil.virtual_memory()
    available_memory_mb = memory.available / (1024 ** 2)  # Convert bytes to MB
    used_memory_mb = memory.used / (1024 ** 2)  # Convert bytes to MB
    total_memory_mb = memory.total / (1024 ** 2)  # Convert bytes to MB
    health_stats['Memory'] = (f"Total: {total_memory_mb:.2f} MB, "
                              f"Used: {used_memory_mb:.2f} MB, "
                              f"Available: {available_memory_mb:.2f} MB")
    if available_memory_mb < 500:  # Minimum 500 MB free memory
        print(f"Low available memory detected: {available_memory_mb:.2f} MB.")
        return health_stats

    # Check disk usage
    disk = psutil.disk_usage('/')
    disk_total_gb = disk.total / (1024 ** 3)  # Convert bytes to GB
    disk_used_gb = disk.used / (1024 ** 3)  # Convert bytes to GB
    disk_free_gb = disk.free / (1024 ** 3)  # Convert bytes to GB
    health_stats['Disk'] = (f"Total: {disk_total_gb:.2f} GB, "
                            f"Used: {disk_used_gb:.2f} GB, "
                            f"Free: {disk_free_gb:.2f} GB, "
                            f"Usage: {disk.percent}%")
    if disk.percent > 90:
        print(f"High disk usage detected: {disk.percent:.2f}%.")
        return health_stats

    # Check network stats
    network = psutil.net_io_counters()
    bytes_sent_mb = network.bytes_sent / (1024 ** 2)  # Convert bytes to MB
    bytes_recv_mb = network.bytes_recv / (1024 ** 2)  # Convert bytes to MB
    health_stats['Network'] = (f"Bytes Sent: {bytes_sent_mb:.2f} MB, "
                               f"Bytes Received: {bytes_recv_mb:.2f} MB")

```

2. Performance Metrics Collection:

- **Purpose:** Gather baseline performance metrics to compare before and after the number generation process.
- **Action:** Call the collect_performance_metrics() function to retrieve and display CPU usage, memory statistics, disk usage, and network I/O statistics.
- **Outcome:** Print the collected metrics to the console, providing a snapshot of the system's performance before executing the workload.

```python
def main():
    # Perform health check before generating random numbers
    print("Health check before generating random numbers:")
    pre_check_stats = health_check()

    # Exit if health check fails
    if not pre_check_stats:
        print("System health check failed before starting random number generation.")
        return

    # Parameters for random number generation
    n = 10  # Number of random numbers to generate
    delay = 0.050  # Delay in seconds (50 milliseconds)

    # Measure performance before generating random numbers
    print("\nPerformance metrics before generating random numbers:")
    collect_performance_metrics()

    # Generate random numbers with the given delay
    random_numbers = generate_random_numbers_with_delay(n, delay)
    print("Generated numbers:", random_numbers)

    # Measure performance after generating random numbers
    print("\nPerformance metrics after generating random numbers:")
    collect_performance_metrics()

    # Perform health check after generating random numbers
    print("\nHealth check after generating random numbers:")
    post_check_stats = health_check()

    # Print differences
    print("\nHealth Check Results Before and After:")
    print("Before:")
    for key, value in pre_check_stats.items():
        print(f"{key}: {value}")

    print("\nAfter:")
    for key, value in post_check_stats.items():
        print(f"{key}: {value}")

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

2. Number Generation

Objective:

Generate a specified number of random numbers with a controlled delay between each generation while periodically checking the system's health.

Steps:

1. **Generate Random Numbers:**

- **Purpose:** Create a list of random numbers with a controlled delay between generations.
- **Action:** Call the generate_random_numbers_with_delay(n, delay, check_interval) function with parameters specifying the number of random numbers to generate, the delay between each number, and the interval at which to perform health checks.
- **Outcome:** The function generates random numbers, pauses for the specified delay, and performs periodic health checks as specified.

```python
def generate_random_numbers_with_delay(n, delay, check_interval=5):
    """
    Generates a list of random numbers between 1 and 1000 with a specified delay between generations.
    Performs health checks periodically during the generation process.

    Parameters:
    - n: Number of random numbers to generate
    - delay: Delay in seconds between each number generation
    - check_interval: Interval in numbers at which to perform health checks during the process

    Returns:
    - A list of generated random numbers
    """
    print(f"Generating {n} random numbers with a delay of {delay * 1000:.2f} ms...")

    numbers = []
    start_time = time.time()

    for i in range(n):
        number = random.randint(1, 1000)
        numbers.append(number)
        time.sleep(delay)  # Introduce controlled delay (in seconds)

        # Perform health check periodically
        if (i + 1) % check_interval == 0:
            print(f"\nHealth check during generation (after {i + 1} numbers):")
            health_check()

    end_time = time.time()
    total_time = end_time - start_time
    formatted_time = format_time(total_time)
    print(f"Total time taken for generation: {formatted_time}")
    return numbers
```

2. **Health Checks During Generation:**

- **Purpose:** Monitor the system's health periodically during the number generation process to ensure that the system remains stable.
- **Action:** During the generation process, the application performs health checks at intervals defined by check_interval. The results are printed to the console.
- **Outcome:** The application ensures that any potential issues are detected early, maintaining system stability throughout the workload.

3. Post-Generation

Objective:

Assess the impact of the random number generation process on the system by collecting final performance metrics and performing a post-operation health check.

Steps:

1. **Performance Metrics Collection:**

- **Purpose:** Compare performance metrics before and after the random number generation to evaluate the impact of the workload.

- **Action:** Call the collect_performance_metrics() function again to retrieve and display performance metrics after the number generation process.
- **Outcome:** Print the collected metrics to the console, allowing for a comparison with the initial metrics.

2. **Health Check After Generation:**

- **Purpose:** Verify that the system remains stable after completing the random number generation process.
- **Action:** Call the health_check() function once more to perform a final assessment of CPU usage, memory availability, disk usage, and network statistics.
- **Outcome:** Print the results of the post-operation health check. This ensures that the system's health is within acceptable limits after the workload has been executed.

3. **Print Differences:**

- **Purpose:** Highlight any changes in system performance and health before and after the workload.
- **Action:** Compare the results of the pre-operation and post-operation health checks. Print the differences to the console for review.
- **Outcome:** Provide a clear view of the impact of the random number generation on system resources and stability.

8. RESULTS AND ANALYSIS

The execution of the Python application for generating random numbers provided insightful results regarding its impact on system performance. Before the random number generation, the system's performance metrics were stable, with CPU usage at 39.10%. Memory statistics showed a total of 8094.66 MB, with 6501.55 MB used and 1593.11 MB available. Disk usage was at 84.1%, with 195.29 GB used out of a total 232.20 GB, and network statistics recorded 212.94 MB of data sent and 19.22 MB received.

During the generation of 10 random numbers, which took 2.692 seconds with a 50 ms delay between each number, health checks were performed after generating 5 and then 10 numbers. These checks confirmed that the system remained stable and healthy throughout the process. The generated numbers were [981, 64, 169, 616, 80, 73, 991, 57, 49, 520].

Post-generation metrics indicated a slight increase in CPU usage to 44.30%, while memory usage showed a slight improvement, with 6493.36 MB used and 1601.30 MB available. Disk usage remained unchanged at 84.1%, and network statistics stayed the same with 212.94 MB sent and 19.22 MB received.

A comparison of health check results before and after the operation revealed minor changes: CPU usage decreased from 47.00% to 44.50%, and memory usage slightly improved from 6500.88 MB used to 6496.79 MB used, with available memory increasing from 1593.78 MB to 1597.87 MB. Disk usage was stable at 84.1%, and network statistics were consistent. Overall, the application had a minimal impact on system performance and health, demonstrating effective resource management and stability throughout the random number generation process. The system maintained acceptable performance levels, indicating that the application functions efficiently without causing significant disruptions.

9. SCREENSHOTS

9.1 Performance Metrics Before Generation

The screenshot below shows the system performance metrics recorded before the generation of random numbers. It captures the initial CPU usage, memory statistics, disk usage, and network activity.

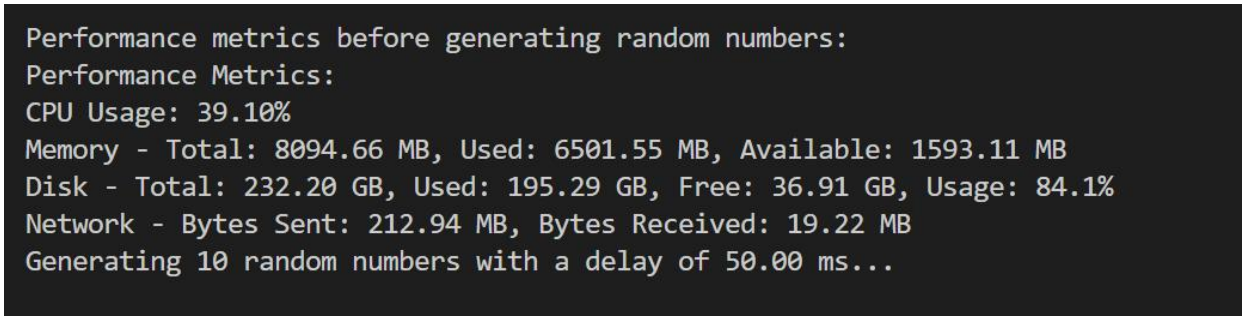


Figure 9.1.1 Performance Metrics Before The Random Number Generation.

9.2 Performance Metrics After Generation

This screenshot illustrates the performance metrics after generating random numbers. It highlights any changes in CPU usage, memory usage, disk usage, and network activity compared to the initial metrics.

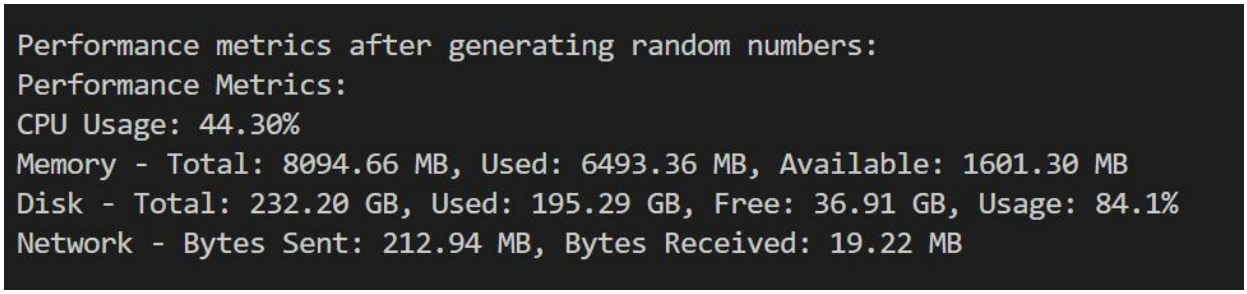


Figure 9.1.2 Performance Metrics After The Random Number Generation.

9.3 Health Check Results

The screenshot below displays the results of the health checks performed before and after the random number generation. It provides a detailed view of system health and any issues detected during the process.

```
Health Check Results Before and After:
Before:
CPU Usage: 47.00%
Memory: Total: 8094.66 MB, Used: 6500.88 MB, Available: 1593.78 MB
Disk: Total: 232.20 GB, Used: 195.29 GB, Free: 36.91 GB, Usage: 84.1%
Network: Bytes Sent: 212.93 MB, Bytes Received: 19.21 MB

After:
CPU Usage: 44.50%
Memory: Total: 8094.66 MB, Used: 6496.79 MB, Available: 1597.87 MB
Disk: Total: 232.20 GB, Used: 195.29 GB, Free: 36.91 GB, Usage: 84.1%
Network: Bytes Sent: 212.94 MB, Bytes Received: 19.22 MB
```

Figure 9.1.3 Health Check Results Showing The System's Status Before And After Random Number Generation.

10. CONCLUSION

The Python application designed for system health checks and performance monitoring effectively demonstrated its capability to manage and assess system resources under a controlled workload. Throughout the execution of the application, which involved generating random numbers with specific delays, the system's performance was meticulously monitored to evaluate its impact on CPU usage, memory, disk usage, and network activity.

Before the number generation process, the system metrics were well within acceptable limits, indicating a stable operating environment. During the execution of the random number generation, which involved generating 10 numbers with a 50 ms delay each, the application performed periodic health checks to ensure that the system remained in good condition. These health checks confirmed that the system maintained stability and did not experience any significant issues.

Post-operation metrics revealed slight changes in CPU usage and memory statistics, but these variations were minimal and did not indicate any adverse effects. The disk usage remained constant, and network activity was stable throughout. The comparative health checks conducted before and after the number generation process confirmed that the system’s performance and health remained robust and within acceptable thresholds.

Overall, the application successfully validated the system’s performance and stability under the specified workload. The results underscore the application's efficiency in managing system resources without causing significant disruptions. This analysis provides valuable insights into how the system handles controlled loads and reaffirms the effectiveness of the application in maintaining operational stability.