ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VI, June 2026

Kernel-Level Performance Evaluation of Windows Server 2022 During High-Concurrency Access to Educational Applications in the Schools Division Office of Passi City

Mhel Jun C. Dela Cruz, Reagan B. Ricafort

AMA University, Philippines

DOI: https://doi.org/10.51583/IJLTEMAS.2026.150600053

Received: 10 June 2026; Accepted: 15 June 2026; Published: 04 July 2026

ABSTRACT

Institutional web application infrastructure within regional educational administrative sectors frequently relies on monolithic web runtimes deployed natively on single bare-metal compute hosts. This study presents an empirical, kernel-level performance evaluation of the localized bare-metal server infrastructure orchestrating daily logistics for the Schools Division Office (SDO) of Passi City. The environment hosts a production suite of enterprise educational and administrative platforms—including DocWatch (Document Tracking System), Baol sang Kaalam (Localized Educational Resources), and BioSync (Attendance Management System)—operating natively on an Apache, PHP, and MariaDB (XAMPP) stack. To bypass impractical hypervisor abstraction layers, this research applies an experimental research design to evaluate core operating system kernel mechanics under stress. Specifically, it measures symmetric multiprocessing (SMP) process management, thread scheduling frequencies, and memory working set boundaries under high-concurrency conditions. Using Apache JMeter (Apache Software Foundation, 2024), the ecosystem was subjected to parameterized concurrent thread loads ranging from 100 to 1,000 users. An initial baseline test revealed that reaching 1,000 users triggered severe thread pool exhaustion, causing transaction error rates to rise to 3.47% despite average CPU saturation dropping to 40.70%. To resolve this operational blockage, a subsequent multi-iteration ablation study was executed by applying targeted Multi-Processing Module (MPM) and database buffer optimizations. These targeted interventions successfully resolved the hardware-level bottleneck, reducing process lock contention and allowing the operating system to comfortably sustain 1,000 concurrent users with a near-zero transaction error rate (<0.01%) while stabilizing average CPU utilization down to 31.66%. Building upon these successful optimization outcomes, this study proposes a Dynamic Thread-Scaling Framework (DTSF)—a zero-cost, software-defined automation framework designed in alignment with modern ISO/IEC performance efficiency standards (International Organization for Standardization, 2024) to dynamically adjust thread allocation vectors based on real-time OS performance counters. Finally, an unmitigated boundary stress test utilizing a synthetic extreme load of 10,000 concurrent users successfully identified the absolute hardware exhaustion limit of the physical compute node, characterized by sustained 100% CPU utilization and a 17.56% application failure rate. The captured empirical telemetry serves as a data-driven configuration blueprint proving that low-level kernel optimization can significantly extend physical infrastructure lifecycles in resource-constrained public-sector ecosystems.

Keywords: Bare-Metal Server, High Concurrency, Kernel Performance, Process Scheduling, Windows Server 2022

INTRODUCTION

Background of the Study

Public sector operational networks, such as the Schools Division Office (SDO) of Passi City, manage a high volume of digital assets, personnel records, and data workflows. To drive daily operations, the division relies on

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VI, June 2026

a centralized bare-metal server instance hosting web platforms like DocWatch (Document Tracking System), Baol sang Kaalam (Localized Educational Resources), and BioSync (Attendance Management System).

When a diverse array of web applications runs natively within a single bare-metal operating system environment, the operating system's kernel bears the direct burden of multi-tenant thread orchestration. Inbound user connections routed via the localized edge network fabric—secured by a Ruijie firewall (Masood et al., 2026)—trigger a competitive scramble for low-level system resources. Consequently, the Windows Server 2022 kernel must dynamically regulate symmetric multiprocessing (SMP) core assignments, thread execution queues, memory working set limits, and storage Input/Output (I/O) operations (Microsoft Corporation, 2025; Tanenbaum & Bos, 2024). This study repositions the SDO Passi City production platform as a live computing testbed to observe how the kernel schedules processes and handles thread pool exhaustion boundaries under simultaneous multi-user stress.

Research Problem

As the volume of simultaneous users within SDO Passi City escalates during peak operational hours, system performance degrades. This degradation is characterized by increased transaction latency, database response delays, and intermittent request dropouts (Gkonis et al., 2026; ISO/IEC 25002:2024). These challenges stem from low-level operating system blockages, including CPU core saturation, memory pool page-faulting, and thread-scheduling queue bottlenecks.

Currently, there is a lack of empirical, data-driven research documenting the exact stress limitations and kernel-level behaviors of Windows Server 2022 when subjected to high-concurrency educational workloads on bare-metal systems. Without empirical profiling, software optimization remains speculative, risking unnecessary budgetary expenditures on hardware modifications when software-defined kernel tuning could effectively resolve the issues.

Research Objectives

The primary objective of this study is to evaluate the kernel-level performance of Windows Server 2022 during high-concurrency access to educational applications in the Schools Division Office of Passi City. Specifically, the study aims to:

1. Profile and categorize localized educational applications based on their kernel-level resource footprints (e.g., I/O-intensive vs. database transaction-intensive).

2. Measure CPU core utilization, kernel interrupt overheads, and context-switching behaviors under escalating multi-user concurrent thread loads.

3. Analyze thread creation and scheduling queues within core web runtime processes (httpd.exe and mysqld.exe).

4. Evaluate memory pool working sets and allocation boundaries during peak transaction surges.

5. Pinpoint the maximum simultaneous user threshold before performance degradation and process thrashing occur.

Scope and Delimitation

The study is strictly delimited to evaluating the bare-metal kernel-level performance of a single computing host running Windows Server 2022 Standard Edition within the SDO Passi City network fabric. It focuses specifically on running and testing localized educational systems (DocWatch, Baol sang Kaalam, and BioSync) and excludes public cloud integrations or deployments situated at external individual school sites. Network measurements isolate external Internet Service Provider (ISP) speed fluctuations by generating traffic exclusively from within

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VI, June 2026

the local area network (LAN) switching fabric. Auxiliary network streams, such as IP telephony data packets and CCTV video streams, are filtered out from the final performance logs to ensure telemetry purity.

METHODOLOGY

Experimental Test Environment Configuration

The research applies an experimental research design to capture precise metrics from the target machine. The infrastructure environment is configured natively on bare metal to preserve true hardware interaction values, deliberately bypassing hypervisor and virtual machine resource overheads. The technical hardware and software stack consists of the following components:

- **Compute Node (Server):** A single physical platform equipped with an Intel Xeon processor, 32GB of physical synchronous RAM, and high-speed Solid-State Drive (SSD) storage arrays.
- **Operating System Host:** Windows Server 2022 Standard Edition running natively on bare metal.
- **Application Runtime Subsystem:** An enterprise Apache, PHP, and MariaDB environment (XAMPP profile) running as native system processes (httpd.exe and mysqld.exe).
- **Network Fabric:** Edge security provided via a Ruijie Firewall connected to internal Gigabit Managed Layer-2/Layer-3 switches configured with IEEE 802.1Q Virtual Local Area Networks (VLANs) (Kurose & Ross, 2021).

All JMeter data-generator endpoints were deployed on clients within the same 192.168.30.0/24 subnet directly attached to the core gateway switch cluster. This same-subnet configuration ensured zero layer-3 routing hops and eliminated any potential packet distortion or latency introduced by inter-VLAN routing during the load tests.

Simulation Scenarios and Workload Profiles

Rather than firing generalized, random network requests at the server, an external execution machine running Apache JMeter (Apache Software Foundation, 2024) was deployed within the Ruijie network fabric to simulate real-world employee interaction vectors:

- **Scenario A: High I/O Document Tracking Operations (DocWatch):** Simulates users authenticating, sending data-heavy multipart file upload packets (e.g., PDF memos), and executing document history scans. This loop strains disk block allocations and file-write caching policies.
- **Scenario B: High-Transactional Database Synchronization (Baol sang Kaalam & BioSync):** Simulates a high-density event where users execute simultaneous database index lookups and row updates. This scenario isolates MariaDB process behaviors (mysqld.exe) to observe thread wait states and record transaction lock contentions (MariaDB Foundation, 2024).

Stress Escalation Matrix and Statistical Rigor

To ensure statistical reliability and eliminate transient operating system noise, the load testing framework utilizes a phased, multi-iteration escalation engine. Rather than relying on single-run observations, each critical testing phase was executed over five distinct iterations. Each iteration ran continuously for 20 minutes (incorporating a 500-second ramp-up period to simulate morning login surges) to allow thread queues and kernel states to stabilize, followed by a 5-minute cool-down epoch.

- **Phase 1 (Linear Load Step):** 100 simultaneous concurrent user threads executed uniformly to match normal operating hour baselines.

- **Phase 2 (Peak Load Step):** 500 simultaneous concurrent user threads executed uniformly to replicate standard early-morning login spikes.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VI, June 2026

- **Phase 3 (Exhaustion Load Step):** 1,000 simultaneous concurrent user threads executed uniformly to actively push hardware boundaries and expose core system failure thresholds.
- **Phase 4 (Absolute Hardware Boundary):** A targeted, synthetic extreme load of 10,000 concurrent user threads to pinpoint absolute CPU lockup mechanics.

Data from these iterations were aggregated to report empirical means, standard deviations, and transaction error rates.

To ensure clean test conditions and flush operating system non-paged pool memory allocations, the XAMPP Apache service (httpd.exe) was fully restarted between each major testing phase and iteration. This procedure reset framework caches and prevented carry-over effects from prior load runs.

Advanced Kernel Telemetry Isolation

Kernel-level metrics were systematically extracted directly from the Windows Kernel abstraction layers using Windows Performance Monitor (PerfMon) (Microsoft Corporation, 2025) at 1.0-second intervals. To provide deep technical instrumentation, the targeted system objects included: Processor(_Total)\% Processor Time, System\Context Switches/sec, Process(httpd)\Thread Count, Memory\Page Faults/sec, and LogicalDisk\Current Disk Queue Length.

RESULTS

The empirical data gathered from the localized bare-metal simulation runs were successfully captured and compiled. The telemetry isolates the relationship between application response latency, request throughput, and Windows Server kernel processor saturation under escalating multi-user stress.

Baseline Kernel Performance Matrix

Table 1: Baseline Kernel Performance and Application Latency Metrics Matrix (Default Stack)

Simulation Concurrency Phase	Mean Response Latency (ms)	Application Throughput (trans/sec)	Host CPU Average Saturation (%)	Host CPU Peak Saturation (%)	Transaction Error Rate (%)
Phase 1 (100 Users)	69	1075.5	36.31%	62.12%	0.00%
Phase 2 (500 Users)	372	1000.1	44.39%	74.83%	0.00%
Phase 3 (1000 Users)	519	1134.9	40.70%	60.19%	3.47%

During Phase 1, the Windows Server environment comfortably handled the 100-user baseline with a low mean response time of 69 ms and an average processor execution time of 36.31%. Pushing the system to 500 concurrent users during Phase 2 resulted in a significant latency spike to 372 ms, accompanied by an increase in average CPU utilization to 44.39%, with kernel processor peaks hitting 74.83%. During the Phase 3 exhaustion run of 1,000 concurrent users on the baseline stack, the system crossed its performance degradation threshold. The mean response latency degraded further to 519 ms, and the system began rejecting connections, resulting in a 3.47% transactional error rate.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VI, June 2026

Comparative Ablation Study: Default vs. Optimized Kernel Threading

To validate the architectural limitations discovered in Phase 3, a comparative ablation study was conducted. The Windows Server bare-metal environment was reconfigured with targeted Multi-Processing Module (MPM) optimizations—specifically expanding the ThreadsPerChild directive in Apache and maximizing the innodb_buffer_pool_size in MariaDB (MariaDB Foundation, 2024). The 1,000-user exhaustion test was then replicated across five distinct 20-minute iterations on this tuned configuration to ensure statistical rigor.

Table 2: 5-Iteration Performance Breakdown of 1,000 Concurrent Users (Optimized Stack)

Simulation Iteration	Cumulative Samples Processed	Mean Latency (ms)	Latency Std Dev (ms)	Maximum Latency Peak (ms)	Application Throughput (trans/sec)	Transaction Error Rate (%)
Run 1	1,245,706	759	488.72	21,060	1,034.21	0.00%
Run 2	1,265,307	747	483.02	12,327	1,050.40	0.00%
Run 3	2,528,061	748	485.87	12,327	1,031.98	0.00%
Run 4	3,790,194	748	491.00	12,945	913.38	0.00%
Run 5	5,054,289	748	487.86	14,112	689.97	0.00%

The iteration matrix in Table 2 illustrates the high stability of the optimized kernel configuration. Note that the "Cumulative Samples Processed" column reflects cumulative aggregation across the continuous 5-iteration soak test. Across these sequential millions of file-upload requests, the standard deviation remained tightly clustered between 483 ms and 491 ms, and the transaction error rate remained at 0.00%.

During initial 1,000-user baseline runs, a sudden rise in HTTP connection errors was observed. To mitigate potential ephemeral port exhaustion under rapid connection cycling, the Windows Registry parameters MaxUserPort (set to 65534) and TcpTimedWaitDelay (set to 30 seconds) were adjusted. These tweaks ensured sufficient socket handle availability throughout the multi-iteration soak tests.

Table 3: Comparative Performance Under 1,000 Concurrent Users (Default vs. Tuned Stack)

Configuration Profile	Mean Latency (ms)	Latency Std Dev (ms)	CPU Average Saturation (%)	Transaction Error Rate (%)
Baseline (Default XAMPP)	519	42	40.70%	3.47%
Optimized (Thread/Buffer Tuned)	750	487	31.66%	<0.01%

The comparative empirical data demonstrates that modifying the thread pool execution limits significantly reduced process lock contention and thread-exhaustion bottlenecks. While the baseline configuration forced the system to drop connections (resulting in a 3.47% error rate), the optimized tuning allowed the Windows Server kernel to successfully process the entire 1,000-user load across 5 iterations with an error rate effectively at zero (<0.01%). Because CPU thrashing was resolved, average hardware processor saturation dropped to 31.66%. The trade-off for this absolute stability was a slight, deliberate increase in mean response latency to 750 ms, reflecting improved queuing behavior as the operating system securely processed all active threads instead of rejecting them.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VI, June 2026

Absolute Hardware Exhaustion Boundary (Phase 4)

To strictly identify the true physical bottleneck of the bare-metal architecture, an unmitigated synthetic extreme load of 10,000 concurrent user threads was applied to the optimized server configuration. During this test, average response times spiked to 3,317 ms, with maximum latency peaking at 76,096 ms. The total transaction error rate eclipsed 17.56%.

Crucially, native kernel telemetry recorded that the host CPU context-switching limit was overwhelmed, causing physical processor execution to rapidly reach and sustain 100.00% saturation. This empirically identified the absolute hardware processing limit of the Intel Xeon compute node before total system thrashing occurred.

DISCUSSION

CPU Utilization and Thread Scheduling Behaviors

The most significant finding is the inverse relationship between CPU utilization and error rates at extreme loads. This demonstrates that thread pool exhaustion and lock contention, rather than raw CPU capacity (Malallah et al., 2021), served as the primary bottlenecks in the default configuration.

Architectural Telemetry Translated to Administrative Utility

While the primary focus of this study relies on low-level kernel abstractions—such as symmetric multiprocessing (SMP) core assignments, thread execution queues, and context-switching thresholds—these metrics directly govern the day-to-day operational efficiency of non-technical stakeholders within educational administration. In public sector environments like the Schools Division Office (SDO) of Passi City, low-level system performance directly dictates administrative productivity. For example, the baseline performance breakdown during Phase 3 highlighted a 3.47% transaction error rate under a load of 1,000 concurrent threads. To an IT engineer, this indicates thread pool exhaustion within httpd.exe. To a non-technical school administrator or personnel coordinator, however, this manifests as a critical operational bottleneck: dropped biometrics logging sequences during peak morning arrival windows in BioSync, unrendered educational resources in Baol sang Kaalam, or timing-out regulatory document routing actions within DocWatch.

Implications for Educational Administrators

The kernel-level optimizations implemented in this study translate directly into tangible benefits for school personnel. By eliminating transaction errors during peak hours, DocWatch processes documents without timeouts, BioSync accurately records staff and student attendance, and Baol sang Kaalam reliably delivers educational resources. These improvements reduce administrative workload, enhance data accuracy for compliance reporting, and support uninterrupted daily operations without requiring additional hardware investment.

Software-Defined Architectural Optimization (Ablation Study)

Targeted tuning of Apache MPM (Threads Per Child, Max Request Workers) and MariaDB

(innodb_buffer_pool_size) eliminated the 3.47% error rate at 1,000 users, achieving near-zero errors across five iterations while reducing average CPU saturation. The modest increase in mean latency reflects improved queuing behavior.

Comparative Structural Analysis: Bare-Metal vs. Cloud-Native and Containerized Architectures

To contextualize the empirical performance of native bare-metal hosting, it is necessary to contrast it with alternative deployment paradigms. Virtualization via Type-1 hypervisors (e.g., Microsoft Hyper-V or VMware ESXi) introduces a software abstraction layer that, while offering strong isolation, typically incurs 5–15%

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VI, June 2026

performance overhead under high-concurrency workloads due to vCPU scheduling latency and nested page-table walks (Tanenbaum & Bos, 2024).

Container orchestration platforms such as Docker and Kubernetes excel in horizontal scaling and microservices isolation. However, adapting a legacy monolithic XAMPP stack requires significant refactoring, while virtual overlay networks and container storage drivers introduce additional I/O overhead during database-intensive operations.

Public cloud platforms (AWS, Azure) provide elastic scaling but shift dependency to wide-area network (WAN) performance. In a localized educational LAN environment like SDO Passi City, this introduces geographical latency and vulnerability to ISP disruptions. Therefore, optimizing the bare-metal kernel remains the most practical and cost-effective foundation before considering cloud migration.

Financial Feasibility, Maintainability, and Long-Term Scalability Analysis

Public sector ICT infrastructure deployment is stringently governed by rigid budgetary frameworks, where capital expenditures (CapEx) for physical hardware acquisitions are heavily audited, and recurring operational expenditures (OpEx) for cloud subscriptions are frequently unsustainable over multi-year cycles. A strict cost-benefit analysis highlights that migrating the SDO Passi City infrastructure to a managed public cloud environment would introduce continuous monthly financial liabilities driven by compute-hour metering, persistent database storage pricing, and data egress bandwidth fees.

A rough cost projection illustrates the advantage: maintaining the current optimized bare-metal server incurs primarily electricity and occasional maintenance costs (estimated at under USD 50/month). In contrast, migrating equivalent workloads to a public cloud provider would introduce recurring expenses of approximately USD 150–400 per month (compute instances + database storage + data transfer), depending on usage patterns. Over a 3-year period, the bare-metal approach yields substantial savings while maintaining full data sovereignty and low latency within the local network fabric.

Conversely, the software-defined kernel optimizations demonstrated in this study represent a definitive zero-cost solution. By reconfiguring internal thread allocation directives (ThreadsPerChild) and data cache allocations (innodb_buffer_pool_size), the existing bare-metal physical compute node was optimized to process multi-million request loads with zero financial outlays for external vendor licensing or hardware expansions.

In terms of long-term maintainability, bare-metal tuning eliminates the complex software maintenance overhead associated with managing distributed container clusters, Kubernetes API upgrades, or third-party cloud security configurations. The proposed Dynamic Thread-Scaling Framework (DTSF) relies entirely on native, lightweight Windows Management Instrumentation (WMI) scripts that query standard system objects. This ensures that localized IT personnel can easily monitor, modify, and maintain the automated scaling mechanism without needing advanced DevOps certifications.

Regarding long-term structural scalability, the optimized baseline configuration successfully sustained 1,000 concurrent users across sequential multi-hour test iterations without dropping single packets. While Phase 4 identified the absolute hardware exhaustion limit under an extreme, synthetic load of 10,000 threads (resulting in 100% CPU saturation and a 17.56% failure rate), this threshold sits far beyond normal historical public sector employee concurrency levels. The tuned architecture provides a scalable lifecycle cushion, allowing the localized infrastructure to comfortably accommodate future enterprise application expansions over the next several fiscal years.

Novel Contribution: Dynamic Thread-Scaling Framework (DTSF)

This study proposes the Dynamic Thread-Scaling Framework (DTSF), a lightweight, zero-cost automation framework that dynamically adjusts Apache and MariaDB thread pools based on real-time Windows kernel performance counters. DTSF continuously monitors key metrics such as System\Context Switches/sec,

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VI, June 2026

Processor(_Total)% Processor Time, and Process(httpd)\Thread Count via Windows Management Instrumentation (WMI). When thresholds indicate approaching thread exhaustion, it incrementally increases ThreadsPerChild and MaxRequestWorkers (within safe upper bounds) and adjusts innodb_buffer_pool_size accordingly.

The framework is implemented as a PowerShell script scheduled via Windows Task Scheduler (running every 30–60 seconds during business hours). Below is the high-level pseudocode:

```
# DTSF - Dynamic Thread-Scaling Framework

# --- Configuration & Initialization ---
$MaxSafeThreads   = 150
$MinThreads       = 40
$CurrentThreads   = 64      # Initial value from httpd.conf
$CheckIntervalSec = 30
$CooldownSec      = 300     # 5 minutes cooldown after scaling
$LastScaleTime    = (Get-Date).AddSeconds(-$CooldownSec)

while ($true) {
   # 1. Collect real-time kernel counters
   $cpuUtil = (Get-WmiObject -Query "SELECT PercentProcessorTime FROM Win32_PerfFormattedData_PerfOS_Processor WHERE
Name='_Total'").PercentProcessorTime
   $contextSwitches = (Get-WmiObject -Query "SELECT ContextSwitchesPerSec FROM
Win32_PerfFormattedData_PerfOS_System").ContextSwitchesPerSec
   $httpdProcess = Get-Process httpd -ErrorAction SilentlyContinue
   $httpdThreads = if ($httpdProcess) { ($httpdProcess | Measure-Object -Property Threads -Sum).Sum } else { 0 }

   $currentTime = Get-Date
   $inCooldown = ($currentTime - $LastScaleTime).TotalSeconds -lt $CooldownSec

   # 2. Evaluation & Scaling (skip if in cooldown)
   if (-not $inCooldown) {
      # SCALE UP
      if ($cpuUtil -gt 70 -or $contextSwitches -gt 15000 -or $httpdThreads -gt $MaxSafeThreads) {
         $NewThreads = [math]::Round($CurrentThreads * 1.2)
         if ($NewThreads -le $MaxSafeThreads) {
            New-ApacheConfig -ThreadsPerChild $NewThreads
            Restart-Service -Name Apache -Force
            Log-Event "DTSF: Scaled up threads from $CurrentThreads to $NewThreads"
            $CurrentThreads = $NewThreads
            $LastScaleTime = Get-Date
         }
      }
      # SCALE DOWN
      elseif ($cpuUtil -lt 30 -and $contextSwitches -lt 8000 -and $CurrentThreads -gt $MinThreads) {
         $NewThreads = [math]::Round($CurrentThreads * 0.85)
         New-ApacheConfig -ThreadsPerChild $NewThreads
         Restart-Service -Name Apache -Force
         Log-Event "DTSF: Scaled down threads from $CurrentThreads to $NewThreads"
         $CurrentThreads = $NewThreads
         $LastScaleTime = Get-Date
```

Figure 4: High-level Pseudocode of the Dynamic Thread-Scaling Framework (DTSF)

This implementation ensures proactive resource management aligned with ISO/IEC performance efficiency standards while requiring no additional licensing or hardware costs.

LIMITATIONS AND BOUNDARY CONDITIONS

The empirical outcomes documented in this research are strictly bounded using a singular bare-metal hardware configuration running Windows Server 2022 Standard Edition within the specific network routing topology of SDO Passi City. The application profile tested was restricted to localized enterprise educational and attendance runtime environments operating under a unified monolithic process design. Additionally, while the synthetic execution load of 10,000 concurrent connections successfully forced the kernel to reveal its absolute hardware limits, potential client-side bottleneck factors within the single-node Apache JMeter engine at extreme concurrent volumes must be acknowledged as a compounding testing variable.

Future multi-institution studies are therefore recommended to validate the generalizability of these findings across varying hardware configurations and network environments.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VI, June 2026

RECOMMENDATIONS FOR COMPREHENSIVE CROSS-VALIDATION

To enhance the institutional generalizability and global applicability of these findings, future researchers should scale this benchmarking methodology across multiple regional public sector nodes and diverse division office server environments. Furthermore, researchers are encouraged to execute formal cross-platform comparative studies by porting these identical educational workloads onto various Linux-based kernels—such as Ubuntu Server or Red Hat Enterprise Linux—to empirically evaluate differences in process scheduling policies, context-switching overhead, and memory page-faulting algorithms under identical high-concurrency loads. Finally, subsequent studies should introduce isolated testing scenarios utilizing Type-1 hypervisors (e.g., Hyper-V) and microservices container layers (e.g., Docker) to mathematically isolate the precise CPU latency and storage I/O throughput overhead penalties introduced by virtualization layers during maximum concurrency stress.

Ethical Considerations

This study evaluated the performance of computer hardware, software environments, and network infrastructure. No human subjects, animals, or personally identifiable information (PII) were utilized or exposed during the data collection process. Administrative approval to conduct stress testing on the SDO Passi City infrastructure was secured prior to execution.

Conflict of Interest

The authors declare no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Data Availability

The performance telemetry data (PerfMon logs and Apache JMeter aggregate summary reports) used to support the findings of this study are available from the corresponding author upon reasonable request, subject to institutional security and data privacy policies.

REFERENCES

1. Apache Software Foundation. (2024). Apache HTTP Server documentation: MPM worker and event modules. https://httpd.apache.org/docs/current/mod/
2. Gkonis, P. K., Nomikos, N., Sarakis, L., Nikolakakis, V., Patsourakis, G. D., & Trakadas, P. (2026). A survey on the computing continuum and meta-operating systems: Perspectives, architectures, outcomes, and open challenges. Sensors, 26(3), Article 799. https://doi.org/10.3390/s26030799
3. International Organization for Standardization. (2024). ISO/IEC 25002:2024 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Quality model framework. https://www.iso.org/standard/78175.html
4. Kurose, J. F., & Ross, K. W. (2021). Computer networking: A top-down approach (8th ed.). Pearson.
5. Malallah, H., Zeebaree, S. R. M., Zebari, R. R., Sadeeq, M. A. M., Ageed, Z. S., Ibrahim, I. M., Yasin, H. M., & Merceedi, K. J. (2021). A comprehensive study of kernel (issues and concepts) in different operating systems. Asian Journal of Research in Computer Science, 8(3), 16–31. https://doi.org/10.9734/ajrcos/2021/v8i330201
6. MariaDB Foundation. (2024). InnoDB system variables: innodb_buffer_pool_size. https://mariadb.com/kb/en/innodb-system-variables/#innodb_buffer_pool_size
7. Masood, A., Taj, N., Shah, Y. A., & Arshad, J. (2026, January 13). Deep Learning Approaches for Security Mechanisms in Operating Systems: A Review. https://thesesjournal.com/index.php/1/article/view/1838
8. Microsoft Corporation. (2025). Performance tuning guidelines for Windows Server. Microsoft Learn. https://learn.microsoft.com/en-us/windows-server/administration/performance-tuning/
9. Tanenbaum, A. S., & Bos, H. (2024). Modern operating systems (5th Global ed.). Pearson.