

Comparative Performance of Operating Systems in Handling Sentiment Analysis REST API Workloads

Jody^{1*} and Imelda²

^{1,2}Master of Computer Science, Information Technology Faculty, Budi Luhur University, Indonesia

ABSTRACT

This study evaluates the performance of a sentiment analysis REST API deployed across different operating system environments, with particular attention to tail latency and available operating-system-level execution indicators. The study examines how differences in platform-level execution conditions and scheduling-related behavior may be associated with variations in service performance under parallel workloads. Comparative experiments were conducted using Apache JMeter with incremental concurrency, while available system monitoring data were used to observe context switching and CPU utilization patterns. All recorded scenarios achieved a 0% error rate. In the available Windows Server and Ubuntu results, Ubuntu exhibited a lower P99 latency (179.40 ms) than Windows Server (416.53 ms), although Ubuntu experienced a single extreme maximum latency spike of 1792 ms. Ubuntu monitoring data also indicated uneven CPU utilization, with one core reaching 91.69%, suggesting a potential workload concentration issue. Windows Server exhibited a higher P99 latency together with context switching peaks of 2765 per second. The macOS results showed lower recorded latency values; however, because the available test conditions did not achieve equivalent workload parameters and synchronized monitoring across all platforms, these findings are interpreted descriptively rather than as evidence of superior operating-system or scheduler efficiency. Overall, the results demonstrate the value of interpreting application-level latency together with available operating-system-level indicators when evaluating REST API deployment performance. The findings further identify worker configuration, CPU affinity, and background process isolation as potential areas for future optimization and controlled experimental validation.

Keywords: Context Switching, CPU Scheduling, REST API, Sentiment Analysis, Tail Latency.

Open Access

Type: Research Paper

Article:

Received: January 19, 2026

Revised: May 16, 2026

Accepted: August 14, 2026

Issued: October 30, 2026

© Jody & Imelda. (2026).



This is an open-access article under the [CC BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) license

*Correspondence Address:

2511600310@student.budiluhur.ac.id

1. INTRODUCTION

Platform Operating system environments can influence the execution characteristics of application services through multiple interacting mechanisms, including process and thread management, CPU scheduling, memory management, and resource allocation. These mechanisms are particularly relevant for REST APIs that process parallel requests, such as sentiment analysis services, where increasing request concurrency may create execution queues and resource contention. Although CPU scheduling represents an important component of this environment, variations in application performance cannot be attributed exclusively to the scheduling algorithm because other operating-system-level subsystems may also contribute to the observed latency behavior. Windows Server and Ubuntu provide different operating system environments and employ different approaches to process and CPU scheduling. Windows uses a priority-based preemptive scheduling mechanism in which thread execution is influenced by dynamic priority and quantum management, whereas Ubuntu commonly employs the Completely Fair Scheduler (CFS), which aims to distribute CPU time fairly among runnable tasks. These architectural differences may influence execution patterns under concurrent workloads. However, their effects on application performance should be interpreted within the broader operating system environment rather than as isolated causal effects of the scheduler alone. Previous studies have examined CPU scheduling, service performance, and tail latency from different perspectives. Research on scheduling algorithms has demonstrated that scheduling policies can influence response time, throughput, and context switching behavior under competitive workloads [1], [2]. In parallel, studies of microservices and REST API performance have emphasized the importance of evaluating response time under increasing access intensity because workload contention may expose resource-management inefficiencies that are not visible under average load conditions [3], [4], [5], [6]. Other studies have further shown that workload interference, cache contention, and queuing strategies may contribute to high-percentile latency and Service Level Agreement (SLA) violations,

highlighting the importance of observing tail behavior rather than relying exclusively on average performance metrics [7], [8], [9]. Cross-platform and container-based evaluations also indicate that isolation mechanisms and resource contention can affect latency characteristics, reinforcing the importance of controlled performance evaluation when comparing execution environments [10].

Despite these developments, existing studies often focus on scheduling mechanisms, application-level benchmarking, or tail-latency management as separate research perspectives. There remains a need for performance evaluations that interpret application-level latency patterns together with available operating-system-level execution indicators within a specific deployment scenario. Such an integrated perspective may help distinguish observed application behavior from system-level symptoms, including CPU utilization imbalance and context switching intensity. Nevertheless, without direct scheduler-level instrumentation, these indicators should be interpreted as associated execution characteristics rather than as conclusive evidence of causal scheduler behavior.

Based on the background, the present study evaluates the performance of a sentiment analysis REST API across different operating system environments. The study focuses on three main concerns:

1. Variations in latency behavior under the recorded workload configurations.
2. The occurrence of tail latency, particularly P95, P99, and maximum latency, under increasing request concurrency.
3. The relationship between application-level performance patterns and the available operating-system-level indicators, including CPU utilization and context switching.

Accordingly, the objective of this study is to evaluate the observed performance characteristics of a sentiment analysis REST API across different operating system environments and to examine how application-level latency patterns can be interpreted alongside available OS-level execution indicators. The evaluation emphasizes service stability at high-percentile latency levels while considering throughput and observed resource utilization characteristics.

This study contributes a comparative performance evaluation that combines REST API latency measurements with available system monitoring indicators to provide an integrated interpretation of cross-platform execution behavior. Rather than treating these indicators as direct measurements of scheduler efficiency, the analysis identifies potential associations between application-level performance and observed system conditions. The findings further identify worker configuration, CPU affinity, and background process isolation as potential areas for future optimization and controlled experimental validation. Through this approach, the study provides deployment-oriented insights into factors that should be considered when evaluating the performance stability of sentiment analysis REST APIs across operating system environments.

2. METHODS

This study employs a comparative experimental approach to evaluate the performance of a sentiment analysis REST API across Windows Server, Ubuntu, and macOS environments. The evaluation focuses on application-level performance metrics, including throughput and latency, together with available operating-system-level execution indicators, such as per-core CPU utilization and context switching rates. Because the experiments evaluate complete operating system environments, the observed performance differences are interpreted as platform-level execution characteristics rather than as isolated effects of individual CPU scheduling algorithms. The experimental approach was selected because it enables structured measurement through the control of available experimental variables and consistent metric recording across platforms, aligning with established microservices benchmarking guidelines that emphasize test scenario equivalence and procedural consistency [11], [12].

The experimental design was intended to maintain consistency in the application implementation, REST API endpoint, runtime configuration, and request payload across the evaluated operating system environments. These controls were applied to reduce application-level variation during testing.

However, the available execution records indicate differences in achieved workload parameters and monitoring conditions across platforms. Therefore, the results are interpreted with consideration of the recorded test configurations rather than as measurements obtained under perfectly equivalent experimental conditions [13]. Observed performance differences may reflect multiple interacting operating-system-level factors, including scheduling behavior, process management, resource utilization, and other platform-specific execution mechanisms [12].

The study variables are defined as follows:

1. **Independent Variable:** The operating system environment in which the sentiment analysis REST API was deployed, namely Windows Server, Ubuntu, and macOS. Each environment has its own platform-specific execution and resource-management characteristics, including different CPU scheduling implementations.
2. **Dependent Variables:** Application-level performance metrics, including throughput, median latency, P95 latency, P99 latency, maximum latency, and error rate, together with available operating-system-level execution indicators, including per-core CPU utilization and context switching rates.

The analytical emphasis on P95 and P99 is based on literature findings that user-facing service quality is often influenced more strongly by tail latency than by average response time. Therefore, the analysis prioritizes high-percentile latency stability to evaluate the consistency of service performance under increasing workload and potential resource contention [14].

The testing environment for each operating system was standardized to the extent permitted by the available platform and hardware configurations. Efforts were made to maintain consistency in the application implementation, API endpoint, runtime configuration, request payload, and other controllable test parameters. Hardware-related factors, including available cores or logical processors, memory capacity, and application server configuration, were considered when interpreting the results. Where the tested environments were not fully identical,

differences in the available system specifications and execution configurations were considered as factors that may influence cross-platform performance. This consideration is important because differences in execution architecture, hardware configuration, and compatibility or virtualization layers may contribute to variations in system overhead and resource distribution [5]. Comparative studies of performance across virtualized and containerized environments similarly highlight the importance of experimental control when interpreting cross-platform performance differences [15].

The system under test is an HTTP-based sentiment analysis REST API that accepts text payloads and returns sentiment predictions. All dependencies and runtime versions are kept consistent to avoid variations caused by implementation. In the context of latency-sensitive services, inference requests are bursty and susceptible to interference and

CPU contention; therefore, control of background processes and non-essential activities is enforced to ensure the scheduling influence signal is easier to interpret. Additionally, studies comparing service protocols emphasize the importance of test scenario consistency and measurement validity using standardized tools to ensure unbiased conclusions [16].

Load generation was performed using Apache JMeter with staged scenarios involving ramp-up and increasing concurrency, where each scenario utilized the same endpoint and request payload. JMeter was selected due to its widespread adoption in API service performance evaluation and its capability to produce structured output (.jtl) that can be consistently re-analyzed [17]. The load parameters used in each scenario are summarized in **Table 1** to provide transparency regarding the experimental configuration.

Table 1. Experimental Load Parameters

Parameter Component	Description	Value/Content in Research
Load generation tool	Application to generate virtual users and record test results	Apache JMeter
Test target	REST API endpoint under test	Sentiment prediction endpoint (HTTP)
Number of threads/users	Number of virtual users running in parallel	Incrementally increased according to the recorded test scenario for each operating system
Ramp-up time	Time taken to increase users to the maximum count	Configured according to the recorded test scenario
Test duration	Duration for which each test scenario is executed	Determined according to the recorded test scenario
Loop/iteration	Number of request repetitions per thread	Configured according to the recorded test scenario
Think time (optional)	Pause between requests to simulate user behavior	Not used / Used (specify according to configuration)
Payload	Data sent in the request (content and size)	Identical across OS
HTTP Header/Configuration	Method, content-type, timeout, keep-alive, etc.	Identical across OS
Maximum concurrency level	Maximum number of parallel users in each scenario	Defined according to the recorded test scenario and available execution configuration
Replications per scenario	Number of times a scenario is repeated to reduce variability	≥ 3 replications per scenario
Test Output	Analyzed result format	JMeter .jtl result file containing request-level records

In the data processing phase, the JMeter test output recorded in .jtl files is utilized to compute key performance metrics. Throughput is calculated as the total number of successfully processed requests (N) during the test duration ($t_{end} - t_{start}$), as defined in Equation (1) [18].

$$T = \frac{N}{t_{end} - t_{start}} \quad (1)$$

Additionally, service reliability is measured via the error rate (ER), defined as the proportion of failed requests (N_{fail}) relative to the total requests (N) in the test scenario, as expressed in Equation (2) [18]. Both metrics are calculated for each load scenario and operating system, then used in conjunction with latency

metrics (median, P95, P99, maximum) to comprehensively compare service performance.

$$ER = \frac{N_{fail}}{N} \times 100\% \quad (2)$$

Data collection is conducted across two layers: the application layer and the OS layer. The application layer utilizes JMeter test output to calculate throughput, median, P95, P99, maximum latency, and error rate. The OS layer leverages monitoring logs to capture scheduling indicators, specifically per-core CPU utilization and context switching rates. This multi-layer approach is necessary because latency changes are often insufficiently explained by application metrics alone; for instance, incorrect CPU availability estimation on SMT processors can

trigger overcommitment and SLA violations in latency-sensitive workloads, necessitating OS-level indicators to explain the root causes of tail events [9]. Furthermore, the warm-up phase and observability/instrumentation overhead must be considered, as they can alter latency/throughput profiles if left uncontrolled [13].

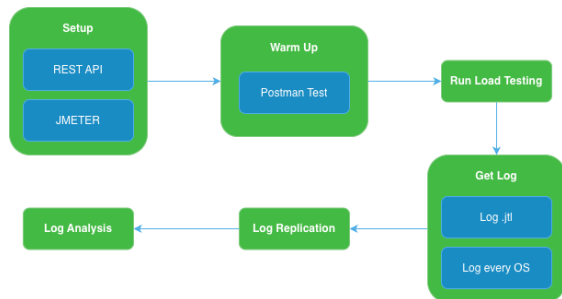


Figure 1. Cross-Operating System Comparative Experiment

As illustrated in the cross-operating system experiment flow in **Figure 1**, the experimental procedure for each scenario is performed sequentially:

1. setting up the REST API service on the target OS with identical configurations,
2. executing a warm-up phase until a stable runtime condition is reached,
3. executing the load test using JMeter and saving the results (.jtl),
4. recording OS metrics during testing, and
5. repeating the scenario multiple times to reduce variability.

The implementation of warm-up phases and rigorous repetition aligns with empirical findings that instrumentation and warm-up dynamics can create significant performance deviations if the initial execution phase is not separated from the main measurement phase [13].

Data analysis is conducted by presenting statistical summaries per scenario and per OS. The analytical focus is directed at tail latency stability (P95/P99/maximum) as a measure of service resilience under parallelism, as research on user-facing services indicates that queueing/scheduling policies explicitly considering tail latency SLOs are capable of yielding more consistent SLO compliance compared to strategies that solely pursue utilization or averages [8]. To strengthen conclusions, cross-OS results are interpreted through OS-level indicators (per-core imbalance and context switching), ensuring that

performance differences possess mechanistic explanations actionable for deployment configurations [2].

Because operating-system performance may also be influenced by execution architecture, resource management, and virtualization or compatibility layers, the findings were interpreted within the context of the tested deployment environments rather than as evidence of a universally superior operating system or scheduling policy. The comparative assessment therefore considered the observed balance among throughput, tail-latency stability, and the available system-level execution indicators for the evaluated deployment scenario [13], [2].

3. RESULTS AND DISCUSSION

This chapter presents the results of comparative experiments on three operating systems Windows Server, Ubuntu, and macOS in handling request loads on a sentiment analysis REST API service. The discussion focuses on two primary categories of findings:

1. Service performance observed at the application layer based on Apache JMeter output (.jtl files), and
2. Execution behavior indicators observed at the operating-system level, including per-core CPU utilization distribution and context switching rates.

Primary emphasis is placed on service stability under parallel loads, with particular attention to tail-latency behavior, which may reveal performance instability that is not captured by average values alone.

3.1 Data Description and Scope of Testing

Service performance data for each operating system were obtained from load-test results saved in .jtl format. Generally, a .jtl file contains per-request records, including response time (elapsed), success or failure status, and timestamps. During the analysis phase, these per-request data were summarized into aggregate metrics, including throughput (requests per second), median latency, high-percentile latency (P95 and P99), maximum latency, and error rate. These metrics were selected to represent both typical and high-latency performance: the median reflects the central tendency of response times under the

tested conditions, whereas P95, P99, and maximum latency describe the occurrence and magnitude of slower responses and extreme latency events that may affect user experience.

In addition to application metrics, this research collected operating system data during testing. Ubuntu provided per-core CPU utilization logs, enabling analysis of whether the workload was distributed evenly across cores or concentrated on a single core. Context switching logs were also available, indicating the intensity of execution transfer between threads/processes due to preemption, blocking, or contention. On Windows, monitoring metrics recorded total CPU utilization and the system context switching rate (per second), while macOS provided snapshot-based system monitoring logs. These OS-level indicators were used to provide additional context for interpreting differences in observed latency behavior, particularly when platforms exhibited similar median latency but different tail-latency characteristics.

It is important to note that the available data reveal differences in test duration, request sample counts, and throughput between platforms. Therefore, interpretations of relative performance are not based on a single metric but on the consistency of observed patterns, including whether latency remains stable or exhibits extreme spikes at particular load levels and whether the available system-level indicators suggest differences in CPU utilization distribution or execution activity.

3.2 Summary of Performance Results from JMeter Output (.jtl)

Based on the summary of calculations from the .jtl files presented in **Table 2**, all platforms recorded a 0% error rate across the tested scenarios. This indicates that no request errors were recorded under the applied test conditions. With a zero error rate across all platforms, the comparison is therefore directed toward performance dimensions, particularly throughput and latency profiles.

Table 2. Summary Of Throughput, Median, P95, P99, Maximum, And Error Rate Per Os

Operating System	N (Request)	Throughput (req/s)	Median (ms)	P95 (ms)	P99 (ms)	Max (ms)	Error Rate
Windows Server	500	16.68	74	92.05	416.53	1598	0%
Ubuntu	1500	9.62	73	92.00	179.40	1792	0%
MacOS	1000	4.27	1	3.00	29.05	514	0%

Three observations emerge from the results. First, Windows Server and Ubuntu show nearly identical median latencies (74 and 73 ms, respectively) and similar P95 values (92.05 and 92.00 ms). This indicates that their typical response times and upper-range latency up to the 95th percentile were similar under the tested scenarios. Second, a clearer difference appears at the higher end of the latency distribution: Windows records a substantially higher P99 value (416.53 ms), whereas Ubuntu records a lower P99 value (179.40 ms) but a slightly higher maximum latency (1792 ms compared with 1598 ms). These results indicate different patterns of high-latency observations, although the available aggregate metrics alone do not establish the underlying causes of those differences. Third, macOS records substantially lower latency values in the available data, with a median of 1 ms, P95 of 3 ms, and P99 of 29.05 ms. However, these figures should be interpreted cautiously because differences in

execution context and test conditions may influence the observed results, as discussed in the study's limitations

Differences in throughput should also be interpreted contextually. Windows Server records the highest throughput, followed by Ubuntu and macOS. However, throughput in these experiments may be influenced not only by operating-system characteristics but also by test duration, concurrency profiles, target endpoint conditions, and worker configurations. Therefore, the results are not interpreted on the basis of throughput alone but are considered together with the observed latency profiles and the available operating-system-level indicators.

3.3 Tail Latency Analysis and Outlier Patterns per Platform

Tail latency is an important aspect of service performance evaluation because it captures slower responses that may not be

reflected by average or median values. To further examine high-latency events, requests exceeding specific latency thresholds were counted, as summarized in **Table 3**. The cumulative latency distribution is presented in **Figure 2**.

Table 3. Counts of Requests Exceeding Selected Latency Thresholds by Operating System

Operating System	>200 ms	>500 ms	>1000 ms	>1500 ms
Windows Server	10	4	1	1
Ubuntu	15	4	1	1
MacOS	2	1	0	0

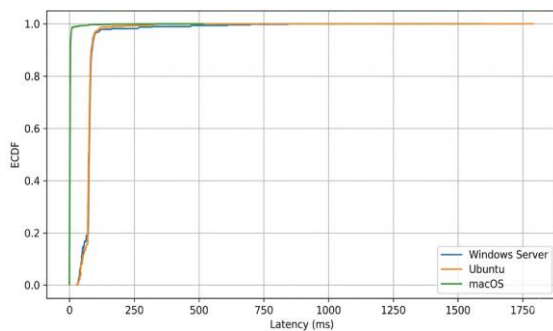


Figure 2. Empirical Cumulative Distribution of Request Latency by Operating System

The results show that, for Windows and Ubuntu, most recorded requests fall within the lower latency range, while a small number of requests extend into substantially higher latency values. Table 3 further shows that both platforms recorded requests exceeding the selected thresholds, although the counts differ across thresholds. These high-latency observations may arise under a range of execution conditions, including temporary request queuing, competition for computing resources, blocking activity, or uneven workload distribution. However, the available latency data alone do not establish which specific mechanism produced the observed spikes. The system-level indicators presented in the following sections are therefore considered as additional context rather than direct evidence of causation.

The differences between Windows' higher P99 value and Ubuntu's lower P99 value indicate different high-latency patterns under the tested scenarios. Windows records a higher P99, suggesting that slower response times extend further into the upper range of the observed latency distribution. Ubuntu records a lower P99

but a slightly higher maximum latency, indicating that the aggregate results include a smaller upper-percentile latency value alongside at least one more extreme observed response. Table 3 provides additional context by showing the number of requests exceeding the selected latency thresholds. Based on these observations, the comparison between platforms should consider both the consistency of upper-percentile latency and the occurrence of individual extreme latency values rather than relying on a single metric alone.

3.4 Ubuntu Discussion: Per-Core Load Distribution and Scheduling Context

On Ubuntu, the available per-core CPU utilization logs show a clear imbalance in observed CPU activity. Average core utilization shows one core (CPU2) operating at a substantially higher rate (approximately 91.69%), while the other cores remain considerably lower (CPU0 ~14.38%, CPU1 ~14.41%, CPU3 ~2.36%). This pattern indicates that CPU activity during the observed test period was unevenly distributed across the available cores, with one core showing substantially higher utilization. Figure 3 illustrates this imbalance in the recorded utilization pattern.

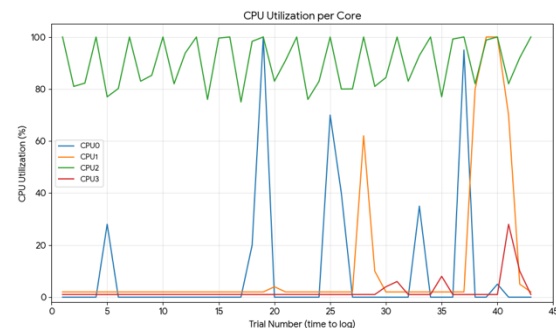


Figure 3. Ubuntu Per-Core CPU Utilization vs. Trials

This utilization pattern provides additional context for interpreting Ubuntu's latency results. The observed concentration of CPU activity on one core may coincide with periods in which execution capacity is unevenly utilized across the system. Such a pattern could contribute to localized execution pressure under particular workload conditions, although the available monitoring data do not establish a

direct causal relationship between per-core utilization and individual latency events. The observed latency profile—relatively similar median and P95 values, a lower P99 than Windows, and a higher maximum latency—should therefore be interpreted alongside the per-core utilization pattern rather than attributed to it directly.

Ubuntu context-switching logs provide additional information about execution activity during the test. The recorded context-switching rate for the Python process was higher than that of the Uvicorn server process. This difference suggests that the two processes experienced different levels of observed execution activity during the monitoring period. However, the available context-switching data alone do not establish whether the additional activity originated from inference computation, request handling, internal process behavior, or other execution mechanisms. From an analytical perspective, the result provides context for further investigation of the execution path but does not independently identify the source of latency or contention.

In a deployment context, these observations highlight the importance of examining application and process configuration when interpreting performance differences. The available data suggest that the recorded workload activity was not evenly distributed across cores during the tested Ubuntu scenario. However, further experiments would be required to determine whether changes in worker configuration, concurrency settings, or CPU affinity would improve the observed latency behavior. Accordingly, the findings should be interpreted as configuration-specific: under the tested setup, uneven per-core utilization coincided with the observed high-latency events, but the available data do not establish that this utilization pattern was their direct cause

3.5 Windows Server Discussion: Scheduling Indications from System Context Switching

The dynamics of scheduling activity during testing are demonstrated through the context switching rate in **Figure 4**.

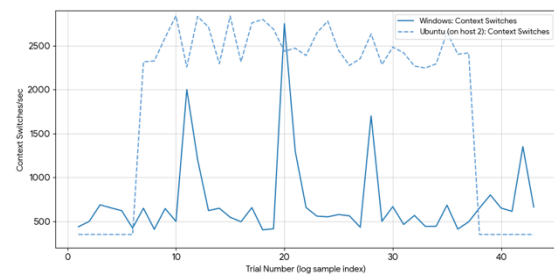


Figure 4. Context Switching Rate vs. Trials

On Windows Server, monitoring metrics show an average total CPU utilization of around 29.7% with peaks around 50%, while the system context switching rate averages around 1261 per second with peaks reaching approximately 2765 per second. Intuitively, higher context switching reflects more activity in transferring execution between threads/processes, which can be triggered by increased concurrency, blocking, or scheduling policies that perform preemption more frequently.

A direct correlation between context switching and tail latency cannot be concluded from a single aggregate metric alone, but it can serve as an indicator that the system experienced more intense scheduling dynamics during certain periods. Windows has a median and P95 comparable to Ubuntu, but a higher P99. This indicates that on Windows, a small fraction of requests experience delays that are more frequent or longer compared to Ubuntu. One explanation consistent with the data is that at specific moments, request threads must compete more strictly for CPU time slices or experience preemption that causes requests to be held up longer, entering the tail.

Since the available Windows metrics are system aggregates (not per-core or per-process), the discussion of Windows in this chapter is best stated as an indication of system-level scheduling behavior. To strengthen evidence in future experiments, it would be beneficial to capture per-core metrics, per-process CPU time, and per-process context switching, allowing for mapping of whether tail events coincide with switching spikes in the worker process or occur due to other system factors.

Nevertheless, the Windows results hold practical significance: if the production target prioritizes tail latency consistency, the higher P99 on Windows compared to Ubuntu suggests that further tuning is necessary (e.g., worker

configuration, process priority, or background load isolation) to stabilize the latency tail. In other words, Windows is strong in terms of throughput under these test conditions but requires special attention regarding tail stability.

3.6 MacOS Discussion: Interpretation of Results and Comparative Caution

Test results on macOS show very low latency values, with the median and P95 in the 1-3 ms range, P99 around 29 ms, and a maximum latency of approximately 514 ms. Numerically, this achievement appears superior to results on Windows Server and Ubuntu. However, in a cross-operating system comparative study, interpreting such large differences requires caution, as low latency figures cannot be directly attributed to differences in CPU scheduling mechanisms without sufficient equivalence in test parameters and supporting data. To complement the application-level reading, scheduler activity indicators on macOS (proxy context switching) are shown in **Figure 5**, depicting activity dynamics during the trial series.

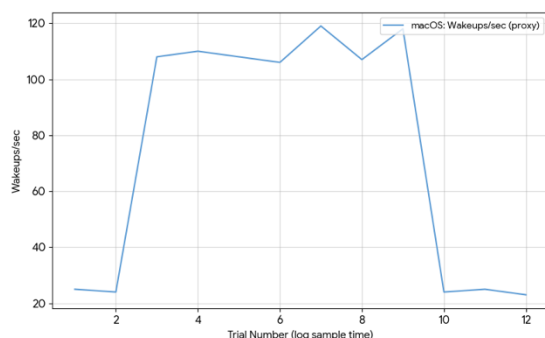


Figure 5. MacOS Scheduler Activity (Proxy Context Switching) vs. Trials

Based on the available test artifacts, the maximum concurrency profile in the macOS test was lower than those of the other two platforms. This suggests that the service may not have experienced an equivalent level of contention. Consequently, the lower latency distribution at high percentiles may partly reflect differences in workload intensity, as execution queues on workers/threads may not have formed as intensely as in scenarios with higher concurrency. Additionally, there is a lack of time synchronization between the test

periods in the .jtl result files and the attached macOS monitoring logs, meaning the available OS-level indicators cannot be used to explain latency behavior during the exact same test period or to link latency changes with CPU utilization dynamics and switching activity.

With these considerations, the macOS results at this stage are treated as descriptive findings of the executed scenario, and comparative reading is limited to the fact that recorded latency metrics were lower in the available data. To strengthen the cross-OS comparative interpretation, macOS testing needs to be aligned regarding load parameters (concurrency, ramp-up, duration, and number of replications) as well as the synchronization of OS monitoring log capture during testing. Once such alignment is performed, the analysis can re-evaluate whether latency differences especially tail latency remain consistent under equivalent load conditions and can be more clearly linked to CPU scheduling behavior indicators.

3.7 Interim Comparative Conclusions and Deployment Configuration Implications

Based on the available data, Windows Server and Ubuntu demonstrate similar central distribution performance (median and P95) but differ in tail characteristics. Windows has a higher P99, whereas Ubuntu has a lower P99 but exhibits a single occurrence of a very high maximum spike. This pattern suggests that, under the tested configuration, Ubuntu exhibits greater stability at high percentiles, whereas Windows displays a heavier latency tail. On the other hand, monitoring indicators on Ubuntu show a tendency for per-core bottlenecks; thus, tail latency stability on Ubuntu has the potential to be further improved through more even load distribution, for example, by adjusting worker counts and/or affinity settings.

On macOS, recorded latency metrics are at a much lower level (median, P95, and P99) with a smaller maximum latency compared to the other two platforms. In a comparative reading, this result indicates that under the executed test scenario, the service on macOS produced a lighter latency profile and smaller tail latency based on the available data. However, the test summary shows lower maximum concurrency, suggesting the service

likely did not enter a level of contention equivalent to Windows Server and Ubuntu. Consequently, macOS results at this stage are more accurately positioned as performance findings at the recorded load parameters, and a mechanistic interpretation regarding CPU scheduling remains dependent on parameter alignment in future testing to ensure cross-OS comparison is robust.

The practical implication of these findings is that the selection of an operating-system environment for production deployment should not be based solely on the lowest observed latency figure, but also on service stability as parallel load increases and on the consistency of tail-latency behavior (P95/P99/maximum). For production, the chosen platform should possess low and stable P95/P99 values and should not exhibit signs of extreme core utilization imbalance or context switching spikes indicating contention. The Ubuntu finding regarding load concentration on a single core suggests that worker and affinity configuration adjustments may be useful for improving load distribution. The higher P99 observed on Windows suggests that further investigation of background-process isolation, priority settings, and improved per-process observability may be useful for identifying potential sources of tail events.. Meanwhile, macOS results serve as an indication of excellent performance on the available data and provide a basis for re-running scenarios with aligned load parameters so that cross-OS comparisons truly reflect differences in CPU scheduling behavior.

CONCLUSION

Based on the research results, the comparison of cross-operating-system performance for sentiment analysis REST API deployment should not be assessed solely through average or central latency measures. Instead, service stability under parallel loads should also be considered through tail-latency indicators (P95/P99/maximum) and available OS-level execution indicators, such as per-core CPU utilization distribution and context switching rates. The controlled comparative experimental approach was applied using the "identical service, identical workload" principle, utilizing the same REST API implementation and sentiment model on

Windows Server, Ubuntu, and macOS, alongside staged load testing from Apache JMeter.

The results indicate that all platforms achieved a 0% error rate in the tested scenarios; thus, the discussion focuses on latency variations and system indicators. Windows Server and Ubuntu demonstrated similar central distribution performance with medians in the 73–74 ms range and P95 around 92 ms, but differed significantly in their tail characteristics. Windows exhibited a higher P99 (~416.53 ms), indicating a heavier tail latency, while Ubuntu showed a lower P99 (~179.40 ms) yet experienced a single extreme maximum latency spike reaching 1792 ms.

OS indicator analysis on Ubuntu revealed a tendency for uneven computational load across cores, with dominance on a single core (CPU2 reaching 91.69%), indicating a potential bottleneck in specific execution paths. This finding suggests that further stability improvements may be explored through worker configuration adjustments, concurrency settings, and affinity strategies intended to distribute work more evenly.

On Windows Server, this finding suggests that further investigation of background-process isolation, service process priorities, and enhanced per-process observability may be useful for identifying potential sources of tail events. Meanwhile, macOS showed significantly lower latency metrics in the available data; however, the testing summary revealed a lower maximum concurrency, suggesting the service likely did not enter a level of contention equivalent to Windows and Ubuntu. Consequently, the macOS results are treated as descriptive findings and serve as a basis for further testing with load scenario alignment.

Overall, the primary contribution of this research is demonstrating the value of interpreting application-level performance metrics together with available system-level indicators when evaluating cross-operating-system deployment scenarios. This combined perspective provides a more comprehensive basis for identifying deployment configurations that prioritize tail-latency stability under parallel loads.

DECLARATIONS

Author Contributions: The first author, Jody, conceptualized the study, designed the experimental methodology, configured the cross-operating-system testing environments, executed the Apache JMeter load testing, collected the system monitoring logs, performed the formal data analysis and visualization, and prepared the original manuscript draft. The second author, Imelda, supervised the research, validated the experimental design and analytical findings, critically reviewed and edited the manuscript, and approved the final version for publication.

Funding: This research received no external funding.

Conflict of Interest: The authors declare no conflict of interest.

Generative Artificial Intelligence Use Statement: During the preparation of this manuscript, the authors used Google Gemini as a supportive assistant solely for language refinement, grammatical editing, and improving the readability of the text. No Generative AI tool was used to generate research data, execute experiments, or formulate scientific conclusions. After using this tool, the authors thoroughly reviewed and edited the content as needed and take full responsibility for the originality, accuracy, integrity, analysis, and conclusions of the published work. Ethics

Ethics Approval and Informed Consent: Not applicable. This study evaluated operating system and application performance metrics and did not involve human participants, animal subjects, or personal sensitive data.

Data Availability: The experimental datasets (including Apache JMeter .jtl output files and operating-system-level monitoring logs) and testing artifacts supporting the findings of this study are not publicly archived due to privacy and internal system configuration considerations. However, the data are available from the corresponding author upon reasonable request.

REFERENCES

- [1] “Docker Performance Evaluation across Operating Systems,” *Applied Sciences*, vol. 14, no. 15, p. 6672, 2024, doi: 10.3390/app14156672.
- [2] M. González-Rodríguez, L. Otero-Cerdeira, E. González-Rufino, and F. J. Rodríguez-Martínez, “Study and evaluation of CPU scheduling algorithms,” *Heliyon*, vol. 10, no. 9, p. e29959, 2024, doi: 10.1016/j.heliyon.2024.e29959.
- [3] B. Cai *et al.*, “AutoMan: Resource-efficient provisioning with tail latency guarantees for microservices,” *Future Generation Computer Systems*, 2023, doi: 10.1016/j.future.2023.01.014.
- [4] H. Rodrigues, A. Rito Silva, and A. Avritzer, “Assessment of performance and its scalability in microservice architectures: Systematic literature review,” *Journal of Systems and Software*, vol. 230, p. 112500, 2025, doi: 10.1016/j.jss.2025.112500.
- [5] A. Lawi, E. Prasetyo, and A. Hajar, “Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System,” *Computers*, vol. 10, no. 11, p. 138, 2021, doi: 10.3390/computers10110138.
- [6] A. Golmohammadi, M. Zhang, and A. Arcuri, “Testing RESTful APIs: A Survey,” *ACM Transactions on Software Engineering and Methodology*, 2023, doi: 10.1145/3617175.
- [7] R. Meyer, E. P. da Silva, L. E. Kirchoff, and C. A. F. De Rose, “IADA: A dynamic interference-aware cloud scheduling architecture for latency-sensitive workloads,” *Journal of Systems and Software*, vol. 194, p. 111491, 2022, doi: 10.1016/j.jss.2022.111491.
- [8] Z. Wang, H. Li, L. Sun, S. Rosenkrantz, H. Che, and H. Jiang, “A Tail Latency SLO Guaranteed Task Scheduling Scheme for User-Facing Services,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 36, no. 4, pp. 759–774, Apr. 2025, doi: 10.1109/TPDS.2025.3542638.
- [9] H. Liao, T.-Y. Liu, J. Guo, B. Huang, D. Yang, and J. Ding, “Retrospecting Available CPU Resources: SMT-Aware

- Scheduling to Prevent SLA Violations in Data Centers,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 36, no. 1, pp. 67–83, Jan. 2025, doi: 10.1109/TPDS.2024.3494879.
- [10] F. Wiedner, M. Helm, A. Daichendt, J. Andre, and G. Carle, “Performance evaluation of containers for low-latency packet processing in virtualized network environments,” *Performance Evaluation*, vol. 166C, p. 102442, 2024.
- [11] L. Faber, S. L. Vadlamani, and A. v. Kistowski, “Benchmarking Microservices: A Systematic Literature Review of Current Approaches and Metrics,” *IEEE Access*, vol. 12, pp. 24560–24585, 2024, doi: 10.1109/ACCESS.2024.3364021.
- [12] J. Vargas, E. Guerra, and J. de Lara, “Automated Performance Testing of Web Applications: A Systematic Mapping Study,” *Automated Software Engineering*, vol. 30, no. 1, p. 8, 2023, doi: 10.1007/s10515-022-00373-w.
- [13] Y. Hammad, A. A. S. Ahmad, and P. Andras, “An empirical study on the performance overhead of code instrumentation in containerised microservices,” *Journal of Systems and Software*, vol. 230, p. 112573, 2025, doi: 10.1016/j.jss.2025.112573.
- [14] J. Li, Y. Liu, and K. Li, “Tail Latency in Cloud-Scale Services: A Comprehensive Survey of Causes and Solutions,” *ACM Computing Surveys*, vol. 56, no. 3, pp. 1–36, 2024, doi: 10.1145/3638291.
- [15] N. Zhao *et al.*, “Scheduling virtual machines and containers: A comparative review of techniques, performance, and future trends,” *Journal of Systems Architecture*, vol. 168, p. 103583, 2025, doi: 10.1016/j.sysarc.2025.103583.
- [16] M. Niswar *et al.*, “Performance Evaluation of Microservices Communication with REST, GraphQL and gRPC,” *Bulletin of the Polish Academy of Sciences: Technical Sciences*, vol. 72, no. 2, 2024.
- [17] T. Heer, “Comparative Analysis of Load Testing Tools: Apache JMeter vs. K6,” *Journal of Web Engineering*, vol. 21, no. 6, pp. 1823–1840, 2022, doi: 10.13052/jwe1540-9589.2165.
- [18] A. A. A. J. Al-Jumaili, “Performance Evaluation of RESTful Web Services using JMeter,” *International Journal of Advanced Computer Science and Applications*, vol. 14, no. 8, pp. 605–612, 2023, doi: 10.14569/IJACSA.2023.0140868.