

BENCHMARKING LINUX DISTRIBUTIONS

FOR

AI TRAINING AND COMPUTATIONAL TASKS

DEEPAK SINGH



COURSE: **OPERATING SYSTEMS** | INSTRUCTOR: **DR. KANCHAN PANTA**

DATE OF CREATION: **02nd MARCH, 2026**

Table of Contents

Abstract	2
Introduction and background.....	2
Problem statement.....	3
Objectives of the project.....	4
Literature review	4
Methodology adopted.....	9
Results - project findings.....	10
Neural network training	10
Numerical computations.....	12
File I/O operation	13
Future development	18
Recommendations	18
Conclusion	19
Bibliography	20
Appendix A.....	22
Appendix B.....	28
Appendix C.....	32
Appendix D.....	34

Abstract

Linux distributions are a popular choice in the scientific and business communities because they offer a fast, secure, reliable, and cost-effective operating system. Most of these distributions focus on different aspects of user activity, which affects their performance in other tasks. Operating system selection is a vital process that affects the performance of the applications running on it. Benchmarking Linux distributions for machine learning, computational, and file I/O tasks eases the selection process for commercial or research use. This work provides benchmarking results for four Linux distributions focused on security, general use, or resource-constrained devices. The tests use Python, a popular programming language, and Bash, a native scripting language. Results show that Ubuntu excels at machine learning tasks compared to other distributions, including its predecessor, Debian, and its successor, Bodhi OS. Parrot OS provides excellent security by turning off containerized applications by default. It also provides good computational and file I/O performance while requiring higher hardware support. The tests also highlight that the choice of scripting language overshadows the choice of distribution for file access operations. Bodhi OS shows the best performance with Python, while Parrot OS performs better with a bash script. Debian and Ubuntu OS experience performance spikes during file access, suggesting that their table buffers may be insufficient for the number of files accessed. Overall, the tests suggest that benchmarking operations should be conducted before selecting a distribution for a task.

Introduction and background

Operating systems (OSs) are essential for providing a standard interface to hardware (Arpaci-Dusseau & Arpaci-Dusseau, 2014). It provides basic functionalities such as file handling, memory and process management, device management, and software application management (University of Wollongong, n.d.). Microsoft Windows, Apple macOS, and Linux are three of the most popular operating systems in the world. Among them, Linux is the only one that is free and open source. It is highly customizable and supported by a large community. Distributed under the GNU General Public License, it has been the basis for another popular operating system, Android (Wikipedia, n.d.).

During operating system development, its developers must balance performance in some areas at the expense of others (Arpaci-Dusseau & Arpaci-Dusseau, 2014). For example, when allocating CPU time between different processes, developers need to decide between fair allocation, time-to-finish, and time-to-start performance. As the foundation for other applications, these choices impact the overall performance that can be extracted from hardware while ensuring the system's stability and reliability. OS also needs to ensure that rogue applications can be isolated and prevented from bringing down the entire system. As a free and open-source alternative, Linux has served as the foundation for many OS distributions that use and modify the kernel to meet their needs (Distrowatch, n.d.).

A good OS is determined by its user interface, compatibility, software availability, performance, security, community support, and cost (PC Fixer, n.d.). Due to licensing requirements, most Linux distributions are free, while providing additional support or features through paid subscriptions or licenses. For example, Ubuntu is a free Linux distribution that also comes with a premium subscription, Ubuntu Pro (Lech, 2023). Ubuntu Pro promises a longer support schedule and legal compliance support. Having the same base kernel makes many Linux distributions quite similar in performance, with variations resulting from kernel tuning or modifications. The proliferation of similar distributions requires that their performance be benchmarked before any of them is used to build or run software applications.

Benchmarking is the process of evaluating a system's performance. This evaluation can be conducted quantitatively or qualitatively, comparing performance with industry leaders, internal alternatives, or external competitors (APQC, n.d.). In the case of an operating system, subjective user preferences may be hard to quantify, but system performance can be verified quantitatively. Although system performance can be vastly improved through user-tuning of the operating system (Ren et al., 2019), it might be worthwhile to compare out-of-the-box performance to assess developers' expectations.

This report aims to assess four Linux distributions by comparing their performance on specific tasks. After presenting the problem statement and objectives, the literature review examines the importance of operating systems, the history of Linux, and the benchmarking process. Afterwards, the methodology for the report is detailed, followed by a discussion on results, recommendations, and conclusions. The report uses multiple scripts to standardize the benchmarking process; most of them are documented and attached as an Appendix.

Problem statement

Compare the performance of different Linux distributions to determine which delivers the best performance out of the box, and whether distributions are tuned for better performance in specific tasks.

Objectives of the project

The objective of this work is to compare the performance of Bodhi OS, Parrot OS, Ubuntu, and Debian to assess which system provides the best performance out of the box for the following purposes:

1. Neural network training
2. Numerical computations
3. File access operations

Literature review

This section aims to establish a theoretical foundation for this study by examining the nature of operating systems and the challenges of their design and implementation. It further explores Linux kernel development and its diverse distributions, highlighting key distributions developed for specific tasks. Finally, it touches on the intricacies of a benchmarking test. Overall, it provides a structured review of the topics involved in benchmarking operating systems, thereby situating the research within its relevant scholarly context.

Operating systems – purpose and complexity

Operating systems are designed as a wrapper around associated hardware to allow application developers to interact with them. There are three objectives of an operating system (Arpaci-Dusseau & Arpaci-Dusseau, 2014):

1. **Virtualization:** It refers to the transparent illusion provided by the OS to application developers that they each have individual and independent access to the hardware. The operating system creates these illusions while tackling the complex task of ensuring fair use of system resources among the processes running on the system. Typically, the operating system manages these through scheduling policies and queues, virtual memories and paging, translation buffers, and interrupts. These can be summarized as follows:
 - a. Scheduling policies are basic policies that an operating system utilizes to share the hardware resources between competing processes and kernel executions. The challenge lies in ensuring a fair division when knowledge of a process's resource requirements and the total time required is limited. A typical compromise is to use multi-level priority queues, where each process starts at the top priority and drops to a lower one after consuming a set amount of time at that priority. Round-robin scheduling occurs based on priority, and processes are regularly moved to the top priority.
 - b. Virtual memory and paging: The operating system ensures that each application gets its own virtual memory isolated and hidden from other processes. This memory is typically divided into program code, stack, heap, and unallocated or free memory. Traditionally, static code is placed at the start of an application's memory, and heap memory follows

it. Heap memory grows sequentially forward. Stack memory, on the other hand, starts at the end of the allocated memory and grows backward. Applications can demand large amounts of memory at startup or increase their memory usage after initialization. So, maintaining all memory for a particular application in a single contiguous zone is not feasible. The operating system maintains multi-level page tables that contain the memory addresses used by an application.

- c. Translation buffers: Memory buffers are maintained to reduce the memory access time. They are built on the assumption that accessed memory and its spatial and temporal neighbors will be accessed again in the future. For this reason, operating systems typically maintain a buffer of recently accessed memory addresses to reduce access time on subsequent calls. The use of translation buffers becomes apparent when many files are accessed in a loop or when elements in a large array are accessed.
- d. Interrupts: These are signals that an operating system uses to interrupt the process currently running on the hardware and then utilizes the CPU for processing the interrupt. Interrupts can be signals from an I/O device that require immediate processing, or they can be timer-based interrupts that interrupt a process so the CPU can be given to another process running simultaneously. Interrupts allow operating systems to maintain fundamental control over the processor.

2. **Concurrency:** It is the operating system's ability to run multiple processes at the same time. On single-core, single-threaded processors, concurrency is illusory because the OS quickly jumps between them. However, true concurrency is demonstrated in a single multi-processor or multi-threaded system where two or more applications run simultaneously and attempt to access resources in parallel. Threads are like processes in that they contain a program counter for instructions and a private set of registers for computation. For concurrency, an OS must manage multiple issues, including thread management, resource sharing, deadlock, and livelock. These can be briefly summarized as follows:

- a. Thread management: The OS needs to provide a consistent manner for the creation and deletion of threads. New threads would need attributes to help execute methods, and they might return values from their computations that the OS should handle. Finally, safe lifecycle management might require a child thread to depend on its parent thread for its lifecycle.
- b. Resource sharing: A key issue with concurrency is that multiple processes might want to read, write, or execute at the same memory location simultaneously. This can create a race in which access and its outcomes might be inconsistent, leading to unpredictable, random results. OS provides locks, condition variables, and semaphores to ensure consistent resource sharing. Locks provide mutually exclusive access to resources, while condition variables can be used for signaling between threads. Semaphores are objects designed to carry out both tasks.
- c. Deadlock: When processes are competing for resources through mutually exclusive access to each resource, they can run into a deadlock. A deadlock is a situation in which

no process can proceed because all are waiting for a resource held by another process, also called a circular wait. There are four necessary conditions for a deadlock to occur: mutual exclusion, hold-and-wait, no preemption, and circular wait. Mutual exclusion refers to exclusive access to a resource that might be needed for consistency. Hold-and-wait is a behavior in which a thread holds a resource while waiting for access from others. No preemption prevents the OS from recovering a held resource. OS prevents deadlocks by ensuring that at least one of these conditions is false. It is also common for an OS to use timer interrupts with resource graphs to break deadlocks.

- d. **Livelock:** It refers to a condition where the threads are running without making any actual progress. Such a condition can occur when threads compete for multiple resources and relinquish held resources when they do not obtain the required access. In such scenarios, two or more threads can run continuously, but cannot access all the resources. One possible way to handle livelock is to add a random delay after each request to prevent request synchronization.

- 3. **Persistence:** It is the operating system's ability to store a value or maintain a persistent state. To maintain such states, the operating system interacts with storage devices through device drivers. Hard Disk drives (HDDs) are among the most common storage devices. It typically consists of multiple rotating disks, or platters, each divided into sectors. Each sector can be accessed via a head that reads or writes data to it. Although it is not explicitly stated, this arrangement also ensures that data reads from consecutive memory locations are faster than those from distant sectors. I/O operations are slower compared to normal computations, so one way to improve performance is to use Redundant Arrays of Inexpensive Disks (RAID). RAIDs can also enhance reliability through redundant mirroring that handles disk failures and increase capacity by providing access to multiple I/O devices. A key reason for the success of RAID technology is that a SCSI RAID device provides an interface similar to that of a SCSI disk drive.

Linux – history and development

Linux is a registered trademark of Linus Torvalds and is part of the Unix-like operating system family (Bovet & Cesati, 2005). It is open-source, and its kernel source code is available under the GPL license. It was initially developed in 1991 for IBM-compatible computers and has since been expanded to support other hardware. It is IEEE POSIX-compliant and is a monolithic, preemptible kernel. Like many modern OS, it is compiled and supports dynamic loading and unloading of kernel modules, such as device drivers. It supports multithreading and multiprocessing while allowing multiple file systems. The operating system is popular because it is free, customizable, low-performance-demanding, compact, stable, and has a large contributor community.

The driver directory contains most of the source code, accounting for nearly 57% of the space (Palix et al., 2011). It is also the largest source of kernel faults and bugs.

Still, in recent years, overall reliability has increased, as the rate of fault elimination has exceeded the rate of fault creation in kernel code. In 2011, a typical fault in Linux systems lasted less than 1.5 years, with the longest lifetimes of around 3 years. In Linux development, anyone can become an author and submit a patch to fix a bug. This patch is picked up by a maintainer, who integrates it into their repository and then posts it to Linus. So, the number of authors reflects the number of contributors to Linux, while the number of maintainers indicates the active workforce maintaining the code. Between 2004 and 2011, the number of authors increased from around 100 to 1400, while the number of maintainers rose from 20 to 120. Most of this increase is aimed towards improving the driver codebase.

Due to rapid development, kernel vulnerabilities have also grown over the years (Niu et al., 2014). It suffers from primarily three types of vulnerabilities: privilege escalation, denial-of-service, and IP spoofing. Although these are patched as they are discovered, the operating system's security posture benefits from OS hardening and extended access control. Overall, experts agree that Linux distributions are typically highly secure OS (Day, 2025). On the performance front, Ren et al. (2019) compared different Linux kernel versions and concluded that, surprisingly, performance has decreased with time. They found that between versions 3.1 and 4.2, around 75% of the analyzed functionalities became slower. The slowdown was substantive, ranging from 65% for at least 50% of the functionalities to over 100% for a few. The authors concluded that most of these could be attributed to 11 changes, carried out to fix misconfigurations and security issues, and to provide new features. The authors recommend kernel Linux tuning to mitigate the performance impact for users.

Linux – distribution

The open-source nature of the Linux kernel has led to a wide range of distributions tailored to specific tasks. These tasks range from education and general purposes to cybersecurity (MyHSTS, n.d.). Sometimes these distributions become so specialized that they recommend that users develop expertise in Linux systems before using them (Kali, 2025). Most Linux distributions include a graphical user interface (Day, 2025). DistroWatch (n.d.) provides a list of past and present Linux distributions. The website also hints at a particular distribution's popularity through the number of page views it receives. The website notes that CachyOs, Debian, Mint, and Ubuntu are among the top ten distributions.

Ubuntu is a leading Linux distribution developed by Canonical Ltd. for general usage (MyHSTS, n.d.). Although the operating system is regularly updated, it typically releases in two modes: a long-term service (LTS) version released every two years and a minor version update released every six months. The LTS version offers five years of security updates, making it suitable for users seeking a long-term solution

(Lech, 2023). Although the operating system is free to use, it also offers a pro version that meets compliance requirements for enterprises and organizations. It is a Debian-based operating system that provides a package manager for easy installation of applications. Debian itself is a popular general-purpose operating system (Debian, n.d.). It is free, stable, secure, and provides extensive hardware support and smooth upgrades. Its stability is one of the key reasons it serves as the base for many other Linux distributions. Like Ubuntu, Debian also provides 5 years of long-term support, with an additional 5 years for commercial users.

For security-conscious users and cybersecurity professionals, Parrot OS and Kali Linux have emerged as popular operating system choices (Day, 2025). Both distributions focus on cybersecurity and are pre-installed with penetration and sandboxing tools. The distributions are constantly updated via rolling releases, which allow faster dissemination of updates and features (Kali, n.d.; ParrotOS, n.d.). Kali Linux comes with three desktop environments: GNOME, which offers the most feature-rich desktop experience; KDE, which provides a Windows-like design and layout; and Xfce, which offers the fastest experience with minimal features (Kali, n.d.). Parrot OS provides a lightweight, modular installation that can be deployed in the cloud and virtual environments (ParrotOS, n.d.). It also provides prepackaged Docker images.

For users looking for a lightweight, minimal operating system, Bodhi OS offers a unique setup. It is built on top of Ubuntu 22.04 LTS (Bodhi, 2023). This distribution aims to balance the command-line interface and a bulky GUI. The operating system provides minimal pre-installed features, including a web browser, a text editor, an image viewer, a file manager, and a package manager. The operating system also includes an App Center that lets users install additional applications as needed.

Benchmarking operating systems

Benchmarking an operating system has become increasingly important, as most modern applications now rely on operating system code for multimedia, graphics, and data storage support (Brown & Seltzer, 1997). A typical web server can spend up to 85% of its time running the OS code, while a general application spends 20% to 90% of its time running OS code. So, a faster operating system will result in faster application performance. Operating system benchmarking can be done in three ways:

1. **Macro-benchmarking:** It involves running actual applications on the operating system and measuring their performance. It is useful for measuring end-to-end performance for a specific workload and provides a summarized overview of hardware and operating system integration.

2. **Profiling:** Kernel profiling involves measuring the performance of the operating system kernel. It helps identify performance bottlenecks for a specific workload, thereby improving the operating system's performance. Still, the limited availability of kernel code for some operating systems makes profiling difficult.
3. **Microbenchmark:** It involves isolating some of the kernel primitives and then testing their performance through custom tests. Osterhout developed one of the initial microbenchmarking test suites, which has been widely used in the industry. Although these tests provided some indication of how operating system performance depends on hardware, they were not sufficiently detailed for such characterizations.

Methodology adopted

This report compares Linux distributions, focusing on system performance for neural network training, general computation, and storage device interactions. Since hardware impacts system performance, ensuring a common hardware support is key to precise benchmarking. The operating systems were tested in a virtual environment created using VirtualBox, where these metrics can be controlled. The host system was only running VirtualBox during these tests. Figure 1 shows the system configurations for Ubuntu (the others were similar).

Figure 1: Virtual machine configuration with Ubuntu OS



To study neural network training performance, two Python scripts were developed to recognize handwritten numerals using the MNIST database (UCI Machine Learning Repository, 2010). Python was chosen for its widespread use in the ML research community. The first script uses a TensorFlow-based neural network, while the second script uses a PyTorch-based Convolutional Neural Network for identification. Both train on the database for 10 epochs. Appendix A presents the annotated code for the two scripts. These scripts were run inside Docker containers to provide a common interface for training. Docker provides commands to monitor memory usage and container performance. Appendix B presents the shell script used to run and monitor the containers. A Bash script automated container execution and polling, enabling a standardized methodology for measuring performance.

Linux is popular among scientists and researchers for computational work. So, a performance comparison was carried out using the container benchmarking work by Sobieraj & Kotyński (2024). The authors suggest that calculating pi using the Leibniz formula is a popular method for testing computational performance. Although the authors used C for their tests, here Python was used to calculate pi using the first 10 million terms of the Leibniz formula. Memory and computation performance were measured using “tracemalloc” and “DateTime” libraries (see Appendix C).

Finally, the typical interaction for most users is file access. However, file access can be tuned for performance by different compilers. So, file access was tested using both Bash and Python scripts. Following Arpaci-Dusseau & Arpaci-Dusseau (2014), the scripts were used to interact with an increasing number of files to evaluate operating system behavior. The script tests file creation, reading, access, writing, and finally file deletion. For file interaction, performance was measured as the time to complete the task (see Appendix D for the implementation).

To ensure uniformity in performance evaluation, the tests were repeated 3 times to obtain an average. For file creation, each test was run 10 times to get a clearer picture through averaging.

Results and findings

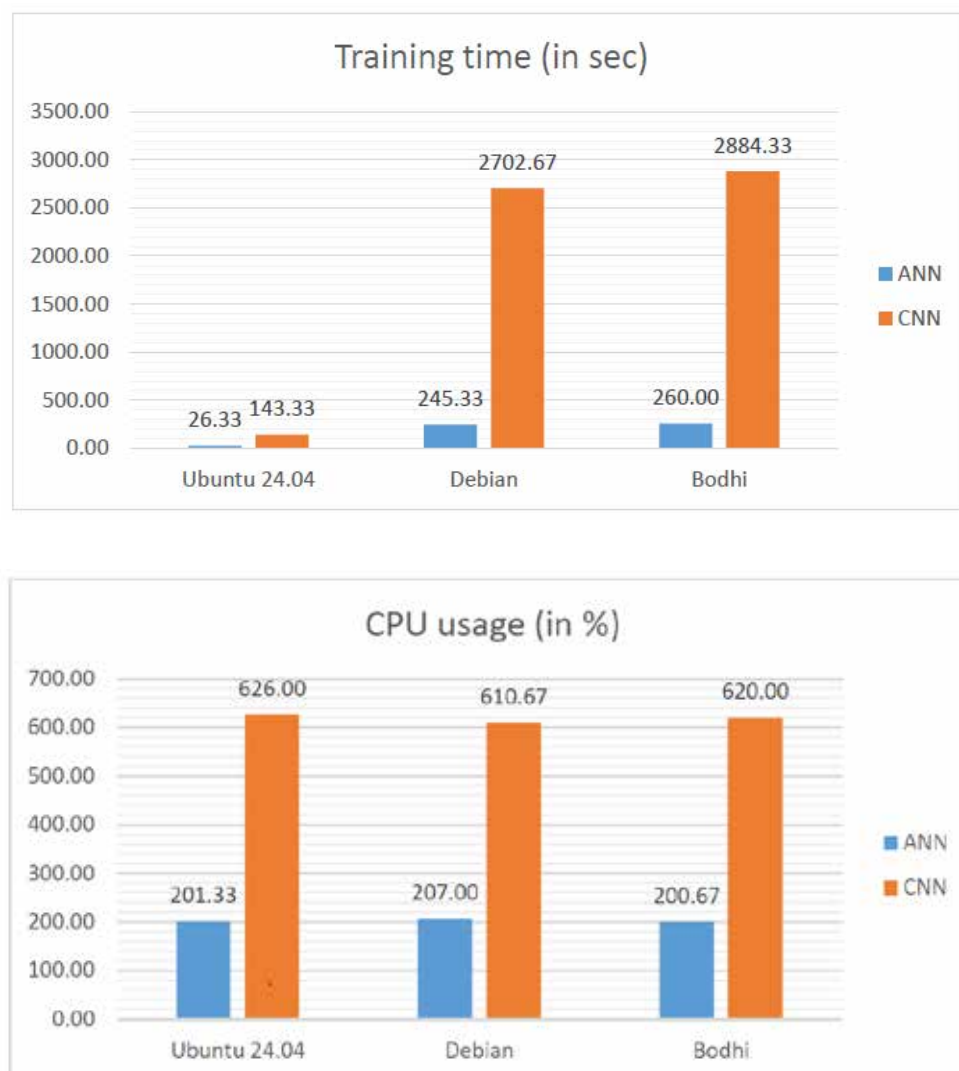
The aforementioned methodology was used to compare four Linux distributions: Parrot OS, Ubuntu, Debian, and Bodhi OS. The selection covers lightweight, security-focused, and popular distributions. This section has been divided into four subsections that detail the results for different tasks and introduce future development.

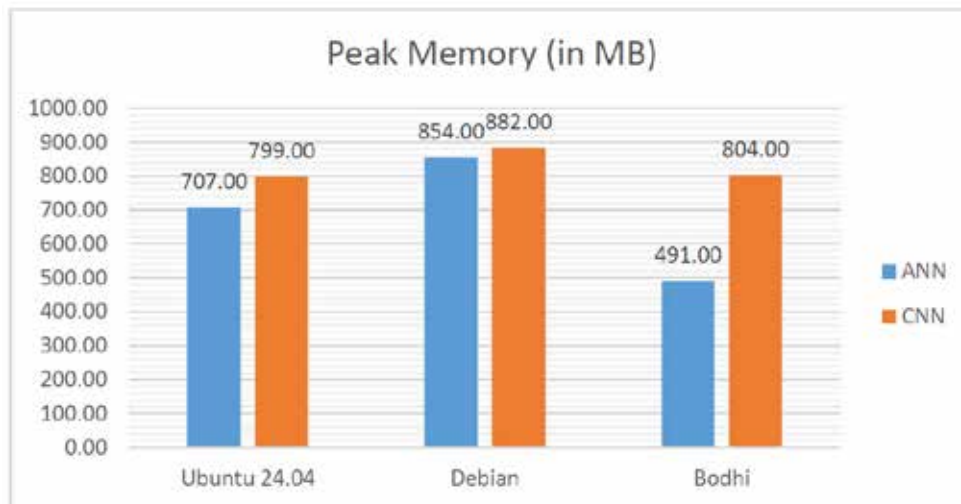
Neural network training

Parrot OS is a security-focused operating system. Out of the box, its SELinux security settings prevent the use of containers based on Other Distributions. This work compares out-of-the-box performance, which disqualified ParrotOS for this

test. Figure 2 compares the other three OSES based on training time, CPU usage, and peak memory usage. The training time plot shows that the Ubuntu OS is tuned for neural network training. It is 10-20 times faster during training. The performance growth is more substantial for CNN. This performance difference is more significant because, when comparing CPU and memory usage, all OS use similar amounts of both. Compared with other neural networks, CNNs require more memory and CPU resources. This is due to the network's larger size. Another possible reason could be the different Python libraries: TensorFlow for the ANN and PyTorch for the CNN. Another key point is that the performance difference between operating systems is independent of the underlying Python library. Finally, the lightweight Bodhi OS uses less memory during training, but its performance is slightly worse than Debian's. Note that, on average, Ubuntu OS used less peak memory than Bodhi OS. This makes Bodhi OS a good second-choice embedded OS for resource-limited devices.

Figure 2: Comparison of Ubuntu, Debian, and Bodhi in neural network training

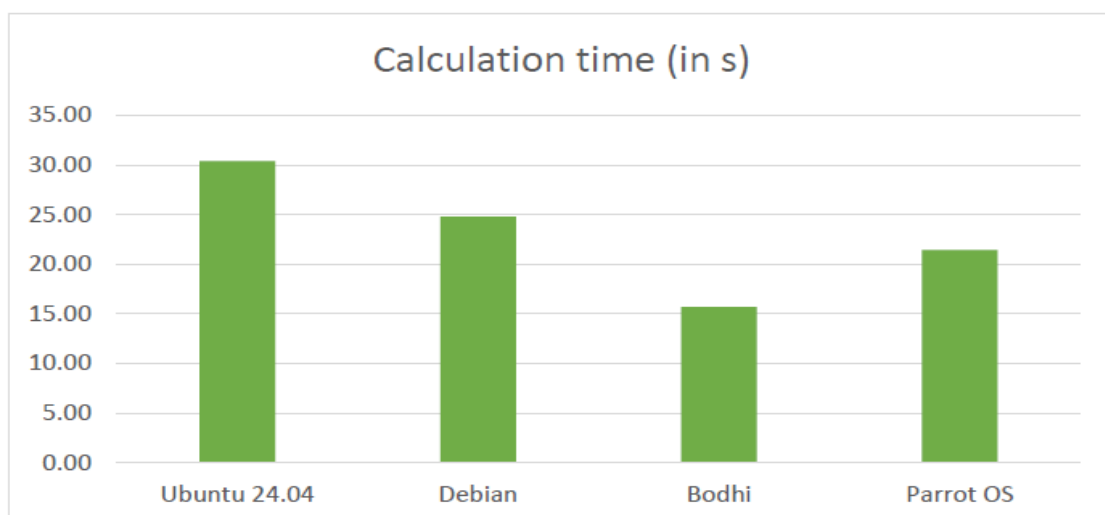


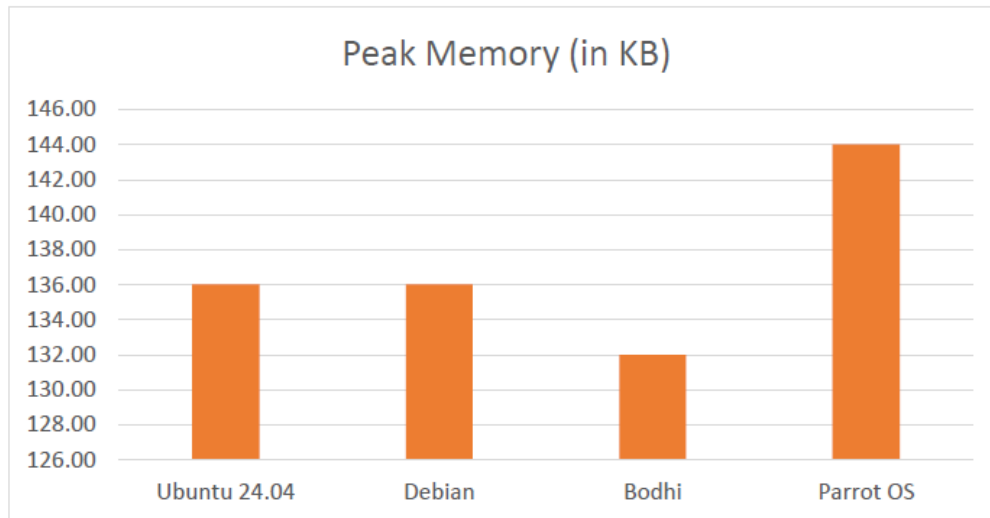


Numerical computation

Figure 3 compares the OS performance when calculating pi values using 10 million terms of Leibniz's series. The plots demonstrate that the lightweight operating system outperforms others. It calculates faster while using less peak memory. Surprisingly, Parrot OS outperforms both Ubuntu and Debian, but it also uses more memory. This behavior ensures that cryptographic calculations are quicker on Parrot OS, underscoring its security focus. Finally, Ubuntu seems to have added additional overhead after branching from Debian.

Figure 3: OS Comparison in pi calculation using 10 million terms in the Leibniz formula.

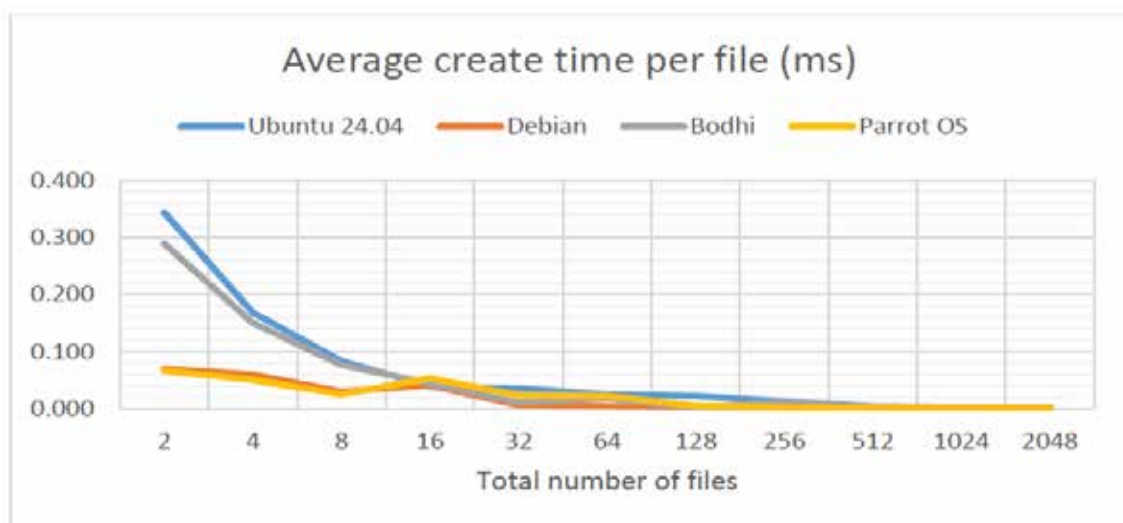




File I/O operation

File I/O is a key OS operation that affects a user's experience. Figures 4 and 5 compare this performance using a Bash script, while Figures 5 and 6 show the same using a Python script. The average time corresponds to 10 iterative operations, with files created and deleted in each iteration. Since these operations were not performed inside a container, the times reflect native operating system performance in a virtual machine.

Figure 4: OS Comparison in average time per file creation and deletion using Bash



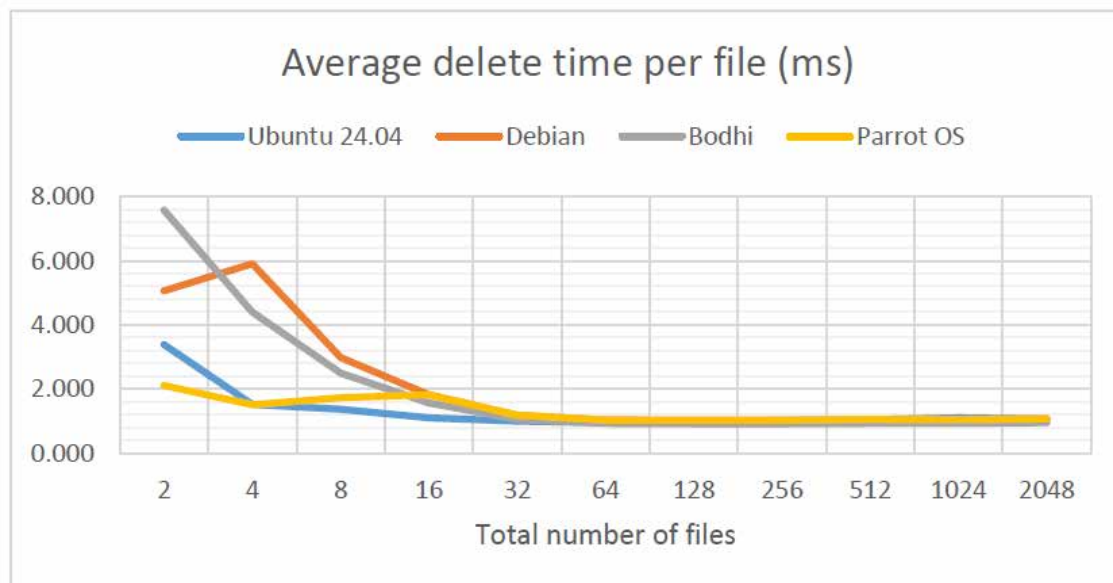
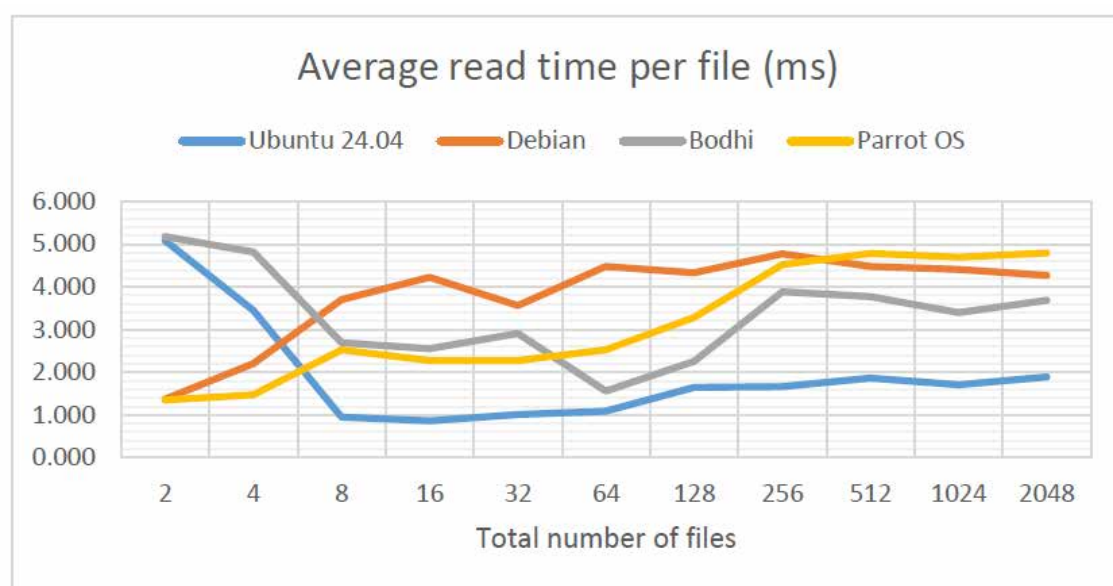
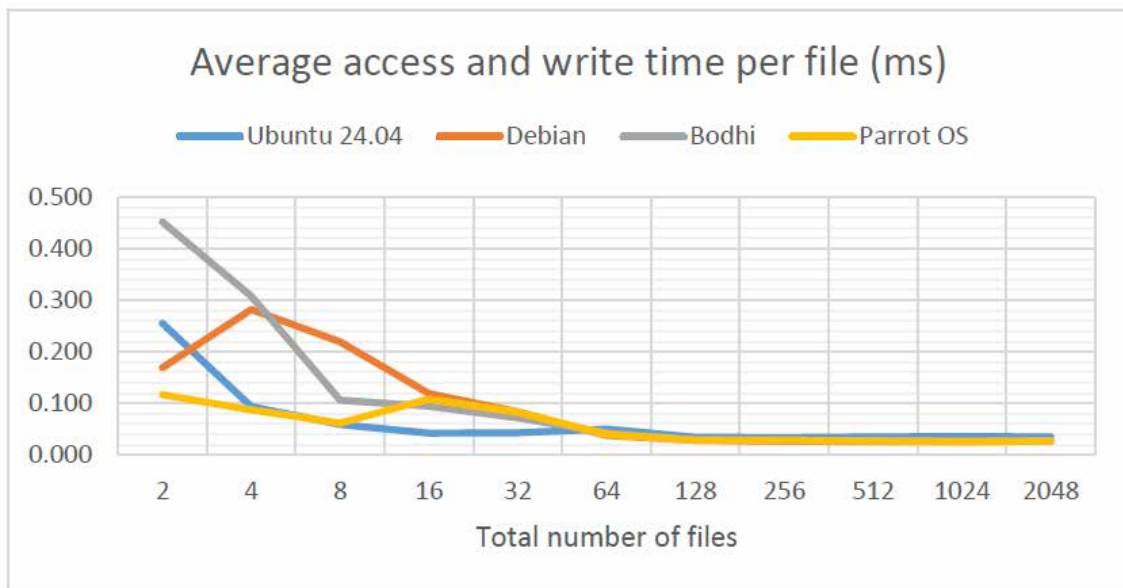
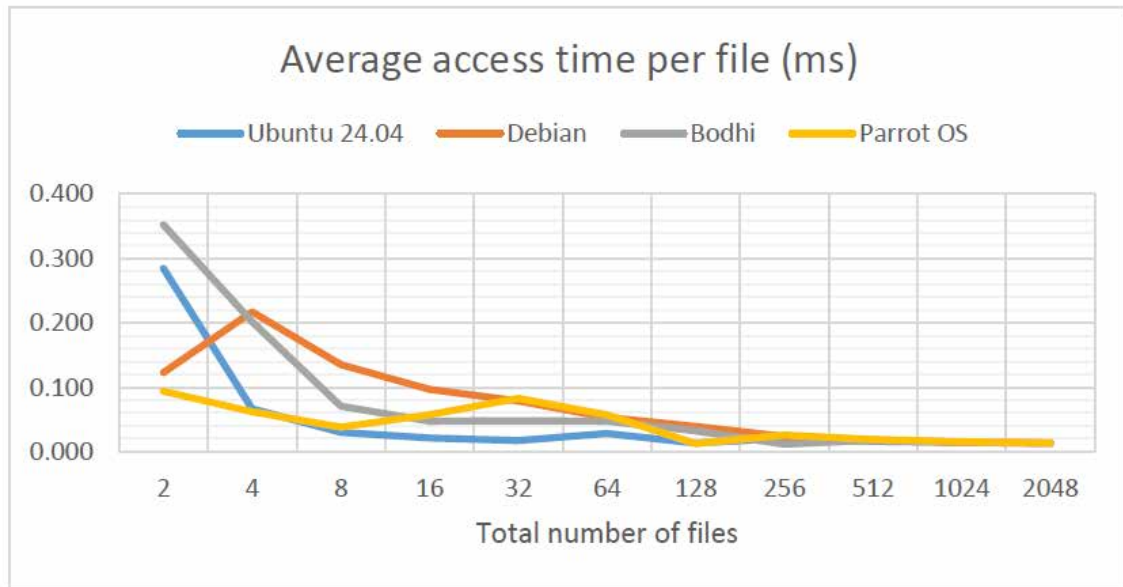


Figure 5: OS Comparison in average time per file reading, accessing, and writing using Bash





A key highlight from Bash script plots is that the time per file typically decreases as the number of files increases. Here, file reading is an exception. This could be due to two reasons: firstly, file reading is the first operation after creation, which could have buffered these access pointers for subsequent operations. Secondly, the OS might prioritize creation, writing, and deletion over reading to ensure file consistency across processes. The second reason is also evident in the lower average values for file creation and deletion compared to other operations.

Figure 6: OS Comparison in average time per file creation and deletion using Python

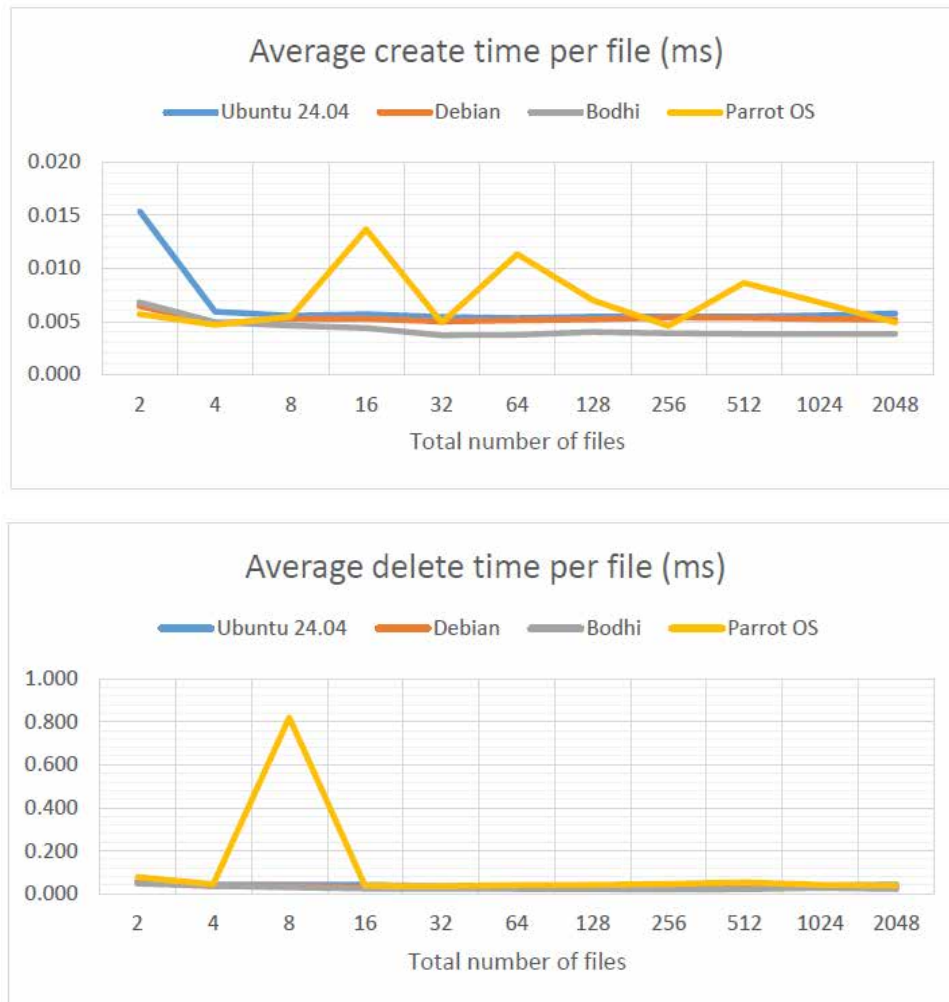
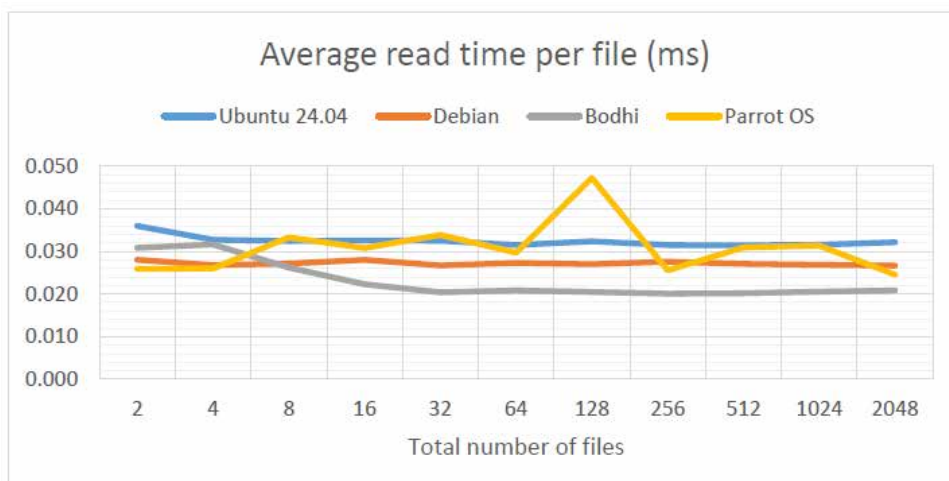


Figure 7: OS Comparison in average time per file reading, accessing, and writing using Python





In the case of a Bash script, file reading behavior differs between OS. Ubuntu and Bodhi OS initially show a decrease in average time per file, followed by an increase. On the other hand, Debian and Parrot OS show a step-wise increase. The stepwise behavior could be due to a smaller file access buffer table in these OS (Arpaci-Dusseau & Arpaci-Dusseau, 2014). Surprisingly, Parrot OS performs the fastest in typical file operations. However, it is substantially slower in file read operations, which could be due to security overheads and kernel tuning.

In Python, the file creation time typically decreases. Parrot OS typically shows peaks, suggesting a file-pointer buffer limit may be present for these combinations. File deletion, access, or writing is consistent across an increasing number of files, highlighting some internal tuning in Python libraries that makes it a constant-time operation. Average file reading time, on the other hand, increases with the number of files. Like Bash, it could be due to the order of file operations. In general, Python file operations are consistent across distributions, but Bodhi OS performs better in some cases. This improvement could be due to its lightweight operations. It uses around

149 MB of peak memory (like Parrot OS), compared to around 133 MB for Debian and Ubuntu.

Comparing Bash and Python, one can observe that Python file operations are typically faster than Bash. Bash outperforms by a large margin in file creation, but it lags in consistent performance and file writing. Bash is also better at reading many files sequentially, especially when the number of files exceeds 256. These behaviors are consistent across different distributions, underscoring that they are likely due to the compiler. This comparison highlights the suitability of these scripting languages in handling file operations.

Future development

The results highlight the importance of comparing different OS before using them for a particular purpose. Some important aspects still require further examination. One issue is confirming whether a particular ML library is suitable for a given distribution. For example, Bodhi OS uses the least memory and CPU with the TensorFlow Python library, which could indicate a synergistic effect. In their experiments, Arpaci-Dusseau & Arpaci-Dusseau (2014) observed a stepwise increase in the average number of file access operations as the number of files increased. This behavior was not typical in the current work. Current operating systems may maintain a larger buffer for the file access table. Finally, in all cases, file reading was the most expensive operation. This could be due to the script's order of operations or to the operating system's default order of operations. Future work can explore and clarify this matter.

Recommendations

This study demonstrates the potential to compare performance across different distributions. The study is limited to Linux distributions, yet the performance differences in computational tasks and neural network training underscore the importance of such comparisons. Looking at neural network training, the performance gains from choosing Ubuntu over other distributions make it the indisputable choice for the job. For general-purpose computing, the lightweight Bodhi OS is a clear choice, with lower memory usage and good performance. However, if community support is crucial, Debian is a better choice, as it offers reasonable performance with lower memory requirements than Parrot OS. Parrot OS can be used if security is the primary focus and memory requirements are not a concern. For file I/O operations, the choice of programming language seems to outweigh differences in operating systems. In the case of Bash scripts, Parrot OS demonstrates the best performance. While for Python scripts, Bodhi OS is the clear choice, and Parrot OS is the worst option. Finally, if containerized operations are required on a security-focused Linux distribution, it is better to build the container image from its own image for compatibility.

Conclusion

Linux distributions are safe and typically free. Depending on the distribution, each release tends to be stable, and some distributions offer long-term support. The plethora of Linux distributions focused on different specializations offers a wide range of choices for implementing a particular task. The current study highlights the importance of evaluating different distributions before choosing one of them for a given task. It shows that Ubuntu substantially outperforms other operating systems for machine learning tasks. Bodhi OS, a lightweight Ubuntu-based alternative, is better for Python computations. For file I/O, the scripting language becomes a more deterministic factor with Python removing discrepancies between different distributions. Finally, Parrot OS, with its security-focused approach, aims to provide faster computation and file I/O operations, but at the cost of more stringent hardware requirements.

Bibliography

APQC. (n.d.). What are the four types of benchmarking? APQC. Retrieved February 23, 2026, from <https://www.apqc.org/blog/what-are-four-types-benchmarking>

Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2018). Operating systems: Three easy pieces. Arpaci-Dusseau Books

Bodhi. (2023, May 1). What is Bodhi Linux? Bodhi Linux. Retrieved February 12, 2026, from <https://www.bodhilinux.com/w/what-is-bodhi-linux>

Bovet, D. P., & Cesati, M. (2005). Understanding the Linux kernel (3rd ed.). O'Reilly Media.

Brown, A. B., & Seltzer, M. I. (1997). Operating system benchmarking in the wake of LMBench. ACM SIGMETRICS Performance Evaluation Review, 25(1), 214–224. <https://doi.org/10.1145/258623.258690>

Day, B. (2025, December 22). Explore the top secure Linux distros for privacy & security in 2025. LinuxSecurity.com. Retrieved January 26, 2026, from <https://linuxsecurity.com/features/explore-the-top-secure-linux-distros-for-privacy-security-in-2025>

Debian. (n.d.). Reasons to use Debian. The Debian Project. Retrieved February 12, 2026, from https://www.debian.org/intro/why_debia

DistroWatch. (n.d.). DistroWatch: Put the fun back into computing. Use Linux, BSD. Retrieved February 12, 2026, from <https://distrowatch.com>

Kali. (n.d.). Frequently asked questions (FAQ). Kali Linux. Retrieved February 12, 2026, from <https://www.kali.org/faq>

Lech. (2023, February 20). Ubuntu Pro - FAQ. Ubuntu Community Hub. <https://discourse.ubuntu.com/t/ubuntu-pro-faq/34042>

MyHSTS. (n.d.). Linux distributions list – a complete guide. MyHSTS.org. Retrieved January 26, 2026, from <https://myhsts.org/tutorial-list-of-all-linux-operating-system-distributions.php>

Niu, S., Mo, J., Zhang, Z., & Lv, Z. (2014, May). Overview of Linux vulnerabilities. In Proceedings of the 2nd International Conference on Soft Computing in Information Communication Technology, 225–228. Atlantis Press. <https://doi.org/10.2991/scict-14.2014.5>

Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J. L., & Muller, G. (2011). Faults

in Linux: Ten years later. In Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, 305–318. Association for Computing Machinery. <https://doi.org/10.1145/1950365.195040>

ParrotOS. (n.d.). Parrot Security – The ultimate framework for your cyber security operations. Parrot Security. Retrieved February 12, 2026, from <https://www.parrotsec.org>

PC Fixer. (n.d.). How to choose the right operating system for your computer. PC Fixer. Retrieved February 23, 2026, from <https://pcfixercomputerrepairs.co.uk/how-to-choose-the-right-operating-system-for-your-computer>

Ren, X.J., Rodrigues, K., Chen, L., Vega, C., Stumm, M., & Yuan, D. (2019, October 27). An analysis of performance evolution of Linux's core operations. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19), 554–569. ACM. <https://doi.org/10.1145/3341301.335964>

Sobieraj, M., & Kotyński, D. (2024, July 31). Docker performance evaluation across operating systems. Applied Sciences, 14(15), Article 6672. <https://doi.org/10.3390/app1415667>

UCI Machine Learning Repository. (1998). MNIST database of handwritten digits [Dataset]. <https://doi.org/10.24432/C53K8>

University of Wollongong. (n.d.). Understanding operating systems. University of Wollongong. Retrieved February 23, 2026, from <https://www.uow.edu.au/student/support-services/academic-skills/online-resources/technology-and-software/operating-system>

Wikipedia. (n.d.). Operating system. Wikipedia. Retrieved February 23, 2026, from https://en.wikipedia.org/wiki/Operating_system

Appendix A

The Python code used to train the ANN for 100 epochs to identify handwritten numerals is given below:

```
import os

import tensorflow as tf

import pandas as pd

from tensorflow.keras import layers, models

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model_selection import GridSearchCV

from sklearn.metrics import mean_squared_error


# Step 1: Load MNIST dataset from the CSV files
train_file = os.environ.get("TRAIN_FILE", "data/mnist_train.csv")
train_df = pd.read_csv(train_file)

# Separate labels and pixel values
train_labels = train_df.iloc[:, 0].values.astype(np.uint8)
train_images = train_df.iloc[:, 1:].values.astype(np.uint8).reshape(-1, 28,
28)

# Preprocess the data
train_images = train_images / 255.0
train_images = train_images.reshape((train_images.shape[0], 28 * 28))

# One-hot encode labels
train_labels = tf.keras.utils.to_categorical(train_labels, 10)


# Build model
model = models.Sequential()

model.add(layers.InputLayer(input_shape=(28 * 28,)))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(16, activation='relu'))
```

```
model.add(layers.Dense(10, activation='softmax'))

# Compile
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

# Train
history = model.fit(
    train_images,
    train_labels,
    epochs=100,
    batch_size=64,
    validation_split=0.1
)
```

The Python code used to train the CNN for 100 epochs to identify handwritten numerals is given below:

```
import os
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms

# Dataset Class
class MNISTDataset(Dataset):
    def __init__(self, csv_file, transform=None):
        df = pd.read_csv(csv_file)
```

```
        self.labels = df.iloc[:, 0].values.astype(np.int64)

        self.images = df.iloc[:, 1:].values.astype(np.float32).
        reshape(-1, 28, 28)

        self.transform = transform

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]

        if self.transform:
            image = self.transform(image)

        return image, label

# CNN Model
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv_layers = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, padding=1), #
            28x28 -> 28x28
            nn.ReLU(),
            nn.MaxPool2d(2), # 28x28 -> 14x14
            nn.Conv2d(32, 64, kernel_size=3, padding=1), #
            14x14 -> 14x14
            nn.ReLU(),
            nn.MaxPool2d(2), # 14x14 -> 7x7
        )

        self.fc_layers = nn.Sequential(
            nn.Linear(64*7*7, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
```

```
        nn.Linear(64, 10)
    )

    def forward(self, x):
        x = self.conv_layers(x)
        x = x.view(x.size(0), -1) # Flatten
        x = self.fc_layers(x)
        return x

# Configuration
train_file = os.environ.get("TRAIN_FILE", "data/mnist_train.csv")

# Transform (normalize + to tensor)
transform = transforms.Compose([
    transforms.ToTensor(), # Converts to [C,H,W] tensor and scales 0-255 to
    0-1
])

# Load datasets
train_dataset = MNISTDataset(train_file, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

# Device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNN().to(device)

# Loss and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training Loop
num_epochs = 100
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
```

```
images = images.to(device)
labels = labels.to(device)

optimizer.zero_grad()
outputs = model(images)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()

running_loss += loss.item() * images.size(0)
epoch_loss = running_loss / len(train_loader.dataset)
print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss:.4f}")
```

Appendix B

The shell code used to analyse CPU performance and memory usage for Docker containers:

```
#!/usr/bin/env bash

set -eu

# -e : exit immediately if any command fails
# -u : error on use of unset variables

IMAGE_NAME="mnist-trainer-ann:v1"
# Name of the Docker image to run

CONTAINER_NAME="mnist-trainer-test-ann"
# Human-readable container name (useful for debugging)

TRAIN_FILE="/app/data/mnist_train.csv"
# Folder and file paths as arguments

STATS_INTERVAL=1
# Number of seconds between CPU/memory samples

STATS_FILE="resource_usage_ann.csv"
#File where resource usage statistics will be stored

if [ ! -f "$STATS_FILE" ]; then
    echo "time,duration_sec,peak_memory_mib,cpu_percent" > "$STATS_FILE"
fi

# Start the container
echo "Starting container..."

START_TIME=$(date +%s)
# Capture the start time as seconds since epoch
# Used later to compute total execution time

CONTAINER_ID=$(docker run -d --rm \
```

```
--name "$CONTAINER_NAME" \  
--cpus=6.0 \  
  
-v "$PWD/data":/app/data \  
-v "$PWD/outputs":/app/outputs \  
-e TRAIN_FILE="$TRAIN_FILE" \  
"$IMAGE_NAME")  
  
# docker run:  
# -d : run container in detached (background) mode  
# --rm : automatically remove container when it exits  
# --name : assign a friendly name  
# -v : map local folder into the container  
# -e : pass environment variables into the container  
# Returns the container ID, which we store for tracking  
# Resource tracking loop  
PEAK_MEM=0  
# Tracks highest memory usage seen (in MiB)  
PEAK_CPU=0  
# Tracks highest CPU usage seen (percentage)  
  
# Safely loop while the container is running  
while docker inspect -f '{{.State.Running}}' "$CONTAINER_ID" 2>/dev/null |  
grep -q true; do  
  
STATS=$(docker stats --no-stream --format "{{.MemUsage}} {{.CPUPerc}}"  
"$CONTAINER_ID" 2>/dev/null || true)  
  
# docker stats --no-stream:  
# fetches a single snapshot of container stats  
# --format:  
# outputs only memory usage and CPU percentage  
# 1. Use || true to prevent set -e from killing the script if container  
exits now
```

```
# 2. Redirect stderr to /dev/null to keep the console clean

# If STATS is empty, the container likely closed between the 'while' check
and now

    if [ -z "$STATS" ]; then
        break
    fi

    MEM_RAW=$(echo "$STATS" | awk '{print $1}')
    CPU_RAW=$(echo "$STATS" | awk '{print $4}')

# Split the two fields returned by docker stats and take the first and last
values

    MEM=$(echo "$MEM_RAW" | awk -F'/' '{print $1}' | sed 's/[^0-9\.]//g')
    # Example MEM_RAW: "123.4MiB / 2GiB"
    # awk extracts "123.4MiB"
    # sed removes units, leaving just the number
    MEM_INT=${MEM%. *}
    # Remove decimal portion for integer comparison

    CPU=$(echo "$CPU_RAW" | sed 's/%//')
    # Example CPU_RAW: "187.23%"
    # Remove percent sign

    CPU_INT=${CPU%. *}
    # Remove decimal portion for integer comparison

    # Update peak memory if current usage is higher
    if [[ -n "$MEM_INT" && "$MEM_INT" -gt "$PEAK_MEM" ]]; then
        PEAK_MEM=$MEM_INT
    fi

    # Update peak CPU if current usage is higher
```

```
if [[ -n "$CPU_INT" && "$CPU_INT" -gt "$PEAK_CPU" ]]; then
    PEAK_CPU=$CPU_INT
fi

sleep "$STATS_INTERVAL"

# Wait before next sample to reduce overhead
done

# Completion time calculation
END_TIME=$(date +%s)

# Capture time after container exits

DURATION=$((END_TIME - START_TIME))

# Total wall-clock execution time in seconds

# Final report

# Append stats to file
TIMESTAMP=$(date +"%Y-%m-%d %H:%M:%S")

echo "${TIMESTAMP},${DURATION},${PEAK_MEM},${PEAK_CPU}" >> "$STATS_FILE"
```

Appendix C

The Python code used to calculate Pi value using 10 million terms in Leibniz formula is given below:

```
#!/usr/bin/env python3

import time
import os
from datetime import datetime
import tracemalloc

def calculate_pi(n_terms):
    """
    Calculate an approximation of pi using the Leibniz series.
    """
    pi_approx = 0
    for n in range(n_terms):
        term = (-1)**n / (2*n + 1)
        pi_approx += term
    pi_approx *= 4
    return pi_approx

# Number of terms
num_terms = 10000000 # 10 million terms

# Start memory tracking
tracemalloc.start()

# Start timer
start_time = time.perf_counter()

# Calculate pi
pi_value = calculate_pi(num_terms)

# Stop timer
end_time = time.perf_counter()
```

```
# Get memory usage

current, peak = tracemalloc.get_traced_memory()

tracemalloc.stop()

duration = end_time - start_time

timestamp = datetime.now()

STATS_FILE = "performance.csv"

file_exists = os.path.isfile(STATS_FILE)

with open(STATS_FILE, "a") as f:

    if not file_exists:

        f.write("timestamp,num_terms,value,duration_
sec,current_mem_kb,peak_mem_kb\n") f.write(f"{timestamp},{num_
terms},{pi_value},{duration:.6f},{current:.6f},{peak:.6f}\n")
```

Appendix D

The Python code used to create, read, access, and delete files 10 times:

```
#!/usr/bin/env python3

import os
import time
import tracemalloc

#Create files and return a list of files
def write_files(n_files):

    # create file names
    files = [f"file_{i}.txt" for i in range(n_files)]

    # ensure files exist
    for fname in files:
        open(fname, "w").close()

    #return the new file names
    return files

#Delete a list of files
def delete_files(files):
    for fname in files:
        if os.path.exists(fname):
            os.remove(fname)

#Access a list of files in a loop
def access_files(files, n_loops):
    # access files repeatedly and append word
    for _ in range(n_loops):
        for fname in files:
            open(fname, "a").close()
```

```
#Reads a list of files in a loop
def read_files(files, n_loops):
    # access files repeatedly and append word
    for _ in range(n_loops):
        for fname in files:
            open(fname, "r").close()

#Access a list of files in a loop and add a word
def access_files_with_write(files, n_loops, word="a "):
    # access files repeatedly and append word
    for _ in range(n_loops):
        for fname in files:
            with open(fname, "a") as f:
                f.write(word + "\n")

def file_performance(n_files, n_loops, result_file):
    # Start memory tracking
    tracemalloc.start()

    start = time.perf_counter()
    files = write_files(n_files)
    end_file_write = time.perf_counter()
    read_files(files, n_loops)
    end_file_read = time.perf_counter()
    access_files(files, n_loops)
    end_file_access = time.perf_counter()
    access_files_with_write(files, n_loops)
    end_file_access_with_write = time.perf_counter()
    files = delete_files(files)
    end_file_delete = time.perf_counter()

    write_time = (end_file_write - start)
    read_time = (end_file_read - end_file_write)
    access_time = (end_file_access - end_file_read)
```

```
access_write_time = (end_file_access_with_write - end_file_access)
delete_time = (end_file_delete - end_file_access_with_write)

# Get memory usage
current, peak = tracemalloc.get_traced_memory()

# Start memory tracking
tracemalloc.stop()

with open(result_file, "a") as f:
    f.write(f"{n_files}, {n_loops}, {write_time:.6f}, {read_time:.6f},
    {access_time:.6f}, {access_write_time:.6f}, {delete_time:.6f},
    {current:.6f}, {peak:.6f}\n")

if __name__ == "__main__":
    n_files_list = [2,4,8,16,32,64,128,256,512,1024,2048]
    n_loop = 10
    result_file = "result_file_py.csv"
    with open(result_file, "w") as f:
        f.write(f"number of files, number of loops, write time, read
        time, access time, access and write time, delete time, current
        memory, peak memory\n")

    for n_files in n_files_list:
        file_performance(n_files,n_loop,result_file)
```

The shell code used to create, read, access, and delete files 10 times:

```
#!/usr/bin/env bash

set -eu

#Configuration
n_files_list=(2 4 8 16 32 64 128 256 512 1024 2048)
n_loops=10
result_file="result_file.csv"
```

```
#Helpers

now() {

# high-resolution timestamp (nanoseconds)

date +%s.%N

}

elapsed() {

awk "BEGIN {print $2 - $1}"

}

#Init result file

echo "number of files, number of loops, write time, read time, access time,
access and write time, delete time" \
> "$result_file"

#Main loop

for n_files in "${n_files_list[@]}"; do

    # Create filenames

    files=()

    for ((i=0; i<n_files; i++)); do

        files+=("file_${i}.txt")

    done

    #Write (create files)

    t0=$(now)

    for f in "${files[@]}"; do

        : > "$f"

    done

    t1=$(now)

    #Read files

    for ((l=0; l<n_loops; l++)); do

        for f in "${files[@]}"; do

            cat "$f" > /dev/null

        done

    done

done
```

```
t2=$(now)

#Access (open/close)
for ((l=0; l<n_loops; l++)); do
    for f in "${files[@]}"; do
        : < "$f"
    done
done

t3=$(now)

#Access + write
for ((l=0; l<n_loops; l++)); do
    for f in "${files[@]}"; do
        echo "Hello World. Test the performance of the OS by
        writing more words in file." >> "$f"
    done
done

t4=$(now)

#Delete files
for f in "${files[@]}"; do
    rm -f "$f"
done

t5=$(now)

#Timing
write_time=$(elapsed "$t0" "$t1")
read_time=$(elapsed "$t1" "$t2")
access_time=$(elapsed "$t2" "$t3")
access_write_time=$(elapsed "$t3" "$t4")
delete_time=$(elapsed "$t4" "$t5")

#Record results
```

```
printf "%d, %d, %.6f, %.6f, %.6f, %.6f, %.6f\n" \  
    "$n_files" "$n_loops" \  
    "$write_time" "$read_time" "$access_time" "$access_write_time" \  
    "$delete_time" \  
    >> "$result_file" \  
  
sleep 0.5 \  
  
done
```