iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** IV **Month of publication:** April 2026

DOI: https://doi.org/10.22214/ijraset.2026.79711

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IV Apr 2026- Available at www.ijraset.com

AI-Based Operating System: Architecture, Intelligence Integration, Challenges, and Future Perspectives

Arahan Babbar[1], Yogita Thareja[2]
Vivekananda School of Information Technology, Vivekananda Institute of Professional Studies, Delhi, India

Abstract: ***The accelerated development of computing workloads that includes highly concurrent tasks, diverse hardware architectures, data flow in real time, and intelligent applications has highlighted the inefficiencies of the conventional operating system (OS) model. Designed on the foundations of rules-driven heuristics that were developed in the heydays of the batch processing mainframe computers, current OSs like Linux and Windows NT fail to maximize their capabilities and provide adaptive and proactive security management and user experience in today's world.***

This paper explores, designs, and analyses a fully-fledged architecture for an AI-Operating System or AI-OS – a novel generation of OS in which artificial intelligence algorithms including machine learning, deep reinforcement learning, large language models, and probabilistic inference engines form part of the kernel and services around it. Unlike ad hoc integrations of individual ML components into existing OS subsystems, AI-OS represents a holistic redesign guided by a set of unifying principles: safety-bounded intelligence, unified telemetry, online adaptive learning, graceful degradation, and transparent explainability. We present a five-layer hierarchical architecture spanning hardware abstraction to natural language user interaction, and elaborate the design of six AI-infused core subsystems: (1) reinforcement learning-driven process scheduling, (2) predictive memory management, (3) intelligent file and storage management, (4) AI-enforced security and anomaly detection, (5) a natural language system interface powered by a locally deployed LLM, and (6) autonomous fault detection and self-healing. We further address the substantial engineering challenges that arise when deploying ML models in a kernel context — including real-time inference constraints, formal safety verification, privacy-preserving online learning, and hardware heterogeneity — and propose concrete mitigation strategies for each. Experimental evaluation on a 16-node cluster running a modified Linux 6.6 kernel with AI-OS extensions demonstrates: a 14.7% improvement in aggregate CPU throughput, a 19.3% reduction in 99th-percentile scheduling latency, a 37.7% decrease in memory page fault rates, a 96.4% zero-day threat detection rate with only 0.08% false positives, an 87% rate of automated fault resolution, and a 9.8% reduction in overall energy consumption — all with a scheduling overhead increase of merely 0.3 percentage points. The findings provide empirical evidence to support the research hypothesis of the significant system-level performance gains possible via AI-OS technology while ensuring the safety and correctness requirements necessary for operational systems. In addition to its technical contribution, this paper also addresses the ethical, social, and governance issues related to the application of autonomous intelligence in the context of an operating system. Finally, it lays out a path forward for future research in this area.

Keywords: ***Artificial Intelligence, Operating System, Machine Learning, Intelligent Resource Management, Reinforcement Learning, Large Language Models, Adaptive Computing.***

I. INTRODUCTION

A. Background and Motivation

The OS is the most fundamental component of any computerized system, and it acts as the mediator between the hardware components and the software programs that make use of them. Regardless of whether we refer to the early batch-processing monitoring programs in the '50s, the UNIX operating system approach during the '70s, the Windows NT model of the '90s, or the current Linux Kernel operating even phones and computers in huge data centers, certain key concepts have remained constant throughout their development process. These include modularity, portability, fairness, and determinism. Some heuristic rules include CFS for CPUs, LRU for memory, and FCFS for IO. In other words, such policies were implemented in order to achieve satisfactory results for diverse applications without knowing their exact behavior.

This approach has been extremely resilient, but the computing environment of the mid-2020s challenges it like never before. Contemporary cloud computing systems can handle tens of thousands of parallel containers with widely varying resource needs. Personal computing devices need to operate constant artificial intelligence assistants, process video streams, and maintain synchronization without draining the battery. The Internet of Things devices should be able to act immediately after receiving sensor data. The self-driving cars require simultaneous processing of images from multiple cameras and sensors in real time.

At the same time, the realm of artificial intelligence has experienced a paradigm shift. Deep reinforcement learning algorithms are superior to human specialists in fields such as game playing, protein folding, and circuit design. Language models can reason, plan, and think creatively, which was inconceivable a decade ago. Graph neural networks can learn intricate connections. Time-series models can predict future states of the system with uncanny precision. Such abilities, once the exclusive preserve of academic laboratories, can now be implemented on standard computers, indeed, on the same computers that run our operating systems.

The convergence of these trends raises a provocative and important question: what if the operating system itself were intelligent? What if, instead of relying on fixed policies engineered for the average case, the OS could learn the specific behavior patterns of the workloads running on it, predict future resource demands, adapt its policies in real time, detect security threats before they cause damage, and explain its behavior in natural language? This is the vision of the AI-Based Operating System.

B. Problem Statement

More precisely, we identify five core limitations of conventional OS design that AI-OS aims to address:

1) Static scheduling policies cannot adapt to diverse or shifting workload patterns, leading to suboptimal CPU utilization, unnecessary context switches, and poor tail latency for latency-sensitive applications.
2) Reactive memory management responds to page faults after they occur rather than anticipating future access patterns, resulting in preventable stalls and I/O bottlenecks.
3) Rule-based security mechanisms (signature databases, access control lists, firewall rules) cannot detect novel, zero-day threats that exploit previously unseen attack vectors.
4) System administration requires deep expert knowledge, creating a significant barrier for non-expert users and increasing the cost of managing large-scale deployments.
5) Fault detection and recovery is largely manual, requiring human operators to diagnose root causes and apply fixes — a process that is slow, error-prone, and unscalable.

C. Contributions

The primary contributions of this paper are as follows:

1) A unified five-layer architectural framework for AI-OS that integrates intelligence across all OS subsystems while maintaining the correctness and safety guarantees required of production systems.
2) Design specifications for six AI-enabled OS subsystems, including mathematical models, architecture design, training, and inferencing methodologies.
3) A systematic study of the difficulties involved in implementing ML in kernel space, including issues of real-time inferencing, safety boundaries, privacy-friendly online learning, and graceful failure modes.
4) Experiments conducted using an adapted Linux kernel 6.6 implementation that includes benchmarking in scheduling, memory management, security, and fault tolerance domains.
5) Ethical and societal ramifications of incorporating autonomous AI systems into the OS.
6) Research directions for future exploration of AI/OS integration, including formal verification techniques, federated learning algorithms, neuromorphic computing systems, and benchmarking standards for AI-powered operating systems.

D. Paper Organization

The rest of this paper is structured as follows. Section 2 reviews the literature. Section 3 introduces the general architecture of the AI-OS system. Sections 4 to 9 provide more details about the key AI components. Section 10 deals with implementation issues. Section 11 reports on experiments and evaluation results. Section 12 considers ethical and social aspects. Section 13 covers the limitations of the approach.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IV Apr 2026- Available at www.ijraset.com

II. RELATED WORK

A. *Machine Learning in CPU Scheduling*

Machine learning techniques have been used in task scheduling before. The first attempt by Tesauro et al. (2007) utilized reinforcement learning for scheduling server loads, showing that the response time could be reduced by 10-15% when using RL agents as opposed to static algorithms. More recently, Decima, a scheduler proposed by Mao et al. (2016) based on Deep RL, was able to reduce the average completion time of jobs scheduled within clusters by 21% compared to existing heuristics. Google's research on the Borg scheduler illustrated that by predicting cluster resources using ML techniques, it was possible to improve cluster utilization by 3-5 percent – a modest gain that translates into millions of dollars in savings on hardware each year.

The more recent works include the exploration of userspace scheduling mechanisms like sched_ext (scx), where the scheduling policy is exposed to eBPF programs. The infrastructure, included in Linux 6.11, allows running custom-made policies (including those made through machine learning) without altering the kernel space code. Some projects like scx_rustland and scx_lavd prove that, by utilizing latency-aware scheduling policies, it is possible to decrease gaming frame time jitter up to 30%. In this work, we continue this trend and implement a policy learned using PPO.

B. *Predictive Memory Management*

Memory management has always been known as an area where ML can be leveraged. Wilson et al. (1995) pioneered the study of memory allocators that paved the way for learned allocator designs. Lagar-Cavilla et al. (2019) used offline ML models on memory trace data collected from Google's warehouse-scale computer infrastructure to predict page access sequences, thereby enabling prefetching schemes that decreased page faults by 20-40%. Ziegler et al. (2021) introduced the idea of a 'learned slab allocator' that uses lifetime predictions to select appropriate slab sizes, resulting in fragmentation reduction of 28%.

In the field of NVM and tiered memory systems, Yan et al. (2022) found that LSTM-based models can predict page migration candidates in systems using DRAM-NVMe tiering and improved application performance by 12-18% over heuristic kernel decisions. This paper synthesizes these lines of inquiry into a predictive memory management system with three key components: page prefetching, heap allocation guidance, and automatic memory tiering.

C. *AI-Driven Security*

One of the first fields to use machine learning algorithms in software systems that are close to OS is security. The signature-based antivirus, which is able to deal with viruses known before, cannot detect zero-day threats at all. Saxe & Berlin (2015) were able to build a neural network that successfully recognizes new malware using only binary data from the executable file with 95% accuracy. Products such as Cylance and SentinelOne utilize those solutions commercially with high efficiency.

At the kernel-level, Mirsky et al. (2018) introduced Kitsune, a system where an ensemble of autoencoders analyses network traffic stats and provides real-time intrusion detection. This was extended by Mushtaq et al. (2020), who showed that using LSTM-based neural networks allowed for fast ransomware and rootkit detection, as well as for privilege escalations. Our AI-OS security module utilizes behavioral analysis, file behavior analysis, and contextual access controls.

D. *Natural Language OS Interfaces*

The concept of using conversational interface for computers can be traced back to researches in ELIZA and Command Language Grammar during the 1980s. The advent of transformer-based large language models has rendered natural language OS interactions possible. The growing commercial interest can be seen in initiatives like ShellGPT, features of AI in Warp Terminal, Copilot in Windows 11 by Microsoft, and advanced Siri by Apple.

As per research performed by Tao et al. (2023), LLMs have the capability to generate shell command sequences from natural language descriptions for common administrative tasks with an accuracy of 78%. But none of these systems combine LLMs with the live telemetry of the system, which gives the system situational awareness. AI-OS Natural Language Interface (NLI) closes this gap by providing access to the tool API of the live system telemetry to the LLMs.

E. *Self-Healing and Autonomous Operations*

AIOps, which stands for Autonomous Intelligent Operations, has gained significant industrial investments. The Watson AIOps, Davis AI, and Anomaly Detection solutions from IBM, Dynatrace, and Datadog, respectively, employ machine learning technology in their monitoring activities within cloud computing.

All these technologies are application and infrastructure level and do not have the capability of operating at the OS level. Previous researches in this area include the study of Candea et al. (2004) involving the use of micro-reboots for OS level self-healing, and the ClearView system developed by Perkins et al. (2009). Our work builds on these concepts by integrating multiple modalities for anomaly detection (hardware counters, kernel tracepoints, application health checks), decision making using reinforcement learning with safe actions, and predictive maintenance using hardware telemetry.

F. Gap Analysis

Even with this extensive research base, however, a major deficiency continues to exist in the form of the absence of any approach which applies machine learning in all OS subsystems in a unified manner that solves engineering challenges involved in using the kernel for real-time inference, safety bounds, online learning, and hardware diversity. AI-OS fills this gap.

III. AI-OS ARCHITECTURE

A. Design Principles

There are six core design principles that guide the AI-OS architecture in maintaining the trade-off between the potential of adaptive intelligence and the safety concerns in the production of an operating system.

1) Independence of Adaptive Intelligence and Policy Enforcement: In the AI-OS design, recommendations and predictions generated from artificial intelligence models are not used to call kernel functions. Rather, all outputs undergo a process of formal policy verification before execution. Thus, no matter how erroneous the outputs may be, they will not cause any harm to the integrity of the OS. The policy verification process involves the use of formally specified safety invariants based on the correct functionality of the OS.

2) Online Learning with Safety Constraints: AI-OS updates its model online based on telemetry data during the runtime to account for changes in workload behavior. But the learning process cannot deviate from the safe trajectory, meaning that any online updates have to stay in a 'trust region,' which is defined as the neighborhood around the last valid model checkpoint. Deviations beyond the trust region invoke a safety check on the updates.

3) Graceful Degradation: For each AI-driven mechanism, a classical equivalent is available and ready to take control if the corresponding AI algorithm fails to perform its tasks due to failure, anomalous output generation, or exceeding the allocated time budget. As a result, AI-OS will never be inferior to a regular OS, even in case of adversarial workloads or failed machine learning models.

4) Unified Telemetry Bus:Telemetry streams from various OS subsystems, including the scheduler, memory manager, file system, security monitoring software, and others, are aggregated in a high-performance ring buffer. The unified telemetry bus allows for cross-subsystem learning: memory pressure metrics, for instance, can influence the I/O scheduler; security anomalies may affect scheduling decisions. This telemetry bus utilizes Linux's perf and eBPF facilities without significant overhead.

5) Transparent Explainability: Any decision made by AI in AI-OS is followed by an easily understandable explanation that is provided using the AI-OS explainability dashboard. For instance, when the AI-based scheduler demotes a process, the dashboard will indicate: “Process X demoted as it has low urgency owing to historical access patterns and memory pressure.” This transparency is essential for user trust and regulatory compliance.

6) Privacy-First Learning: AI-OS collects only the telemetry necessary for system optimization, and applies differential privacy mechanisms to all data used for model training. No personally identifiable information is included in training data. Users can inspect what data is being collected and opt out of data collection for any category without affecting core OS functionality.

B. Layered Architecture

The AI-OS architecture consists of five hierarchical layers:

Layer	Name	Key Components	AI Technologies
5	User Interaction Layer	LLM Interface, Voice, NL CLI, Explainability Dashboard	LLM (7B), RAG, Tool-calling
4	AI Service Layer	Inference Engine, Model Registry, Telemetry Bus, Training Pipeline	ONNX Runtime, TensorRT, Fed. Learning

Layer	Name	Key Components	AI Technologies
3	Intelligent System Services	AI Scheduler, Predictive Memory, Smart FS, AI Security	PPO, TCN, Transformer, Autoencoder
2	Adaptive Kernel	RL Policy Engine, Anomaly Detector, Self-Healer, Safety Verifier	eBPF ML, Formal Methods, RL Agent
1	Hardware Abstraction Layer	CPU, GPU, NPU, Memory, Storage, Network, Sensors	PMDK, RDMA, DMA-BUF

Figure 1: Five-layer architecture of the AI-Based Operating System

C. *Cross-Layer Communication*

Layers communicate through three channels: (1) the synchronous kernel call path, used for time-critical decisions such as scheduling; (2) the asynchronous telemetry bus, used for monitoring and learning; and (3) the explainability API, which exposes AI decisions to the user interaction layer. A strict no-upward-side-effects rule ensures that higher layers cannot directly modify kernel state — they may only make requests through the policy verification layer.

D. *Model Deployment Strategy*

AI-OS employs a tiered model deployment strategy based on available hardware. On embedded and IoT systems with constrained resources, ultra-lightweight models (< 100K parameters, INT4 quantized) run entirely on the main CPU with < 10 microsecond inference latency. On mainstream desktop and server systems, medium models (1-10M parameters, INT8 quantized) run on the main CPU or an integrated GPU. On high-performance systems with dedicated NPU or dGPU hardware, full models (10M-1B parameters) run on the accelerator, freeing the main CPU entirely. The OS automatically detects available hardware and selects the appropriate model tier at boot time.

IV. AI-DRIVEN PROCESS SCHEDULING

A. *Limitations of Classical Schedulers*

CFS (Completely Fair Scheduler), which was introduced in Linux Kernel 2.6.23, works based on virtual runtime fairness; that is, it schedules the runnable process with the minimum vruntime, where vruntime is incremented at a speed proportional to its niceness value relative to its weight. Even though it provides fairness statistically, there are many drawbacks to it.

The first one is that CFS does not have any semantic knowledge about the application. This means that CFS cannot tell if the process running on the web server is latency critical or not – it is just a runnable process. Second, CFS does not take into consideration the cache state, meaning that preemption of a process running on a core whose cache contains most of the data required by the process will require additional cache reloads. Third, CFS is reactive, but not predictive.

The Windows NT scheduling algorithm is also based on a priority multilevel feedback queue with static priority management for foreground processes and I/O boosts. It too is unable to learn from the application behavior, making it incapable of responding to unknown types of workload.

B. *Reinforcement Learning Formulation*

The process scheduling scenario is modelled as a MDP, which is represented by the tuple (S, A, T, R, gamma) where

1) State space S: A set of telemetry data points such as CPU usage for each core, LLC misses for each process, run queue sizes, memory pressure statistics, number of I/O waits, network usage, power consumption, and past execution of processes embedded as a compact code.
2) Action space A: A discrete action space that includes time slice length (4 choices: 1ms, 4ms, 16ms, 64ms), core choice (when more than one core exists), preemption (preempt/ yield), and NUMA node affinity.
3) Transition dynamics T: Actual system behavior inferred using telemetry data points.
4) Reward function R: A weighted combination R = w1 * Throughput + w2 * (1/Latency_P99) + w3 * (1/Energy) - w4 * ContextSwitchCost, with weights tuned via grid search on validation workloads.
5) Discount factor gamma = 0.95, balancing immediate and long-term rewards.

The PPO model is trained offline with 10,000 hours worth of telemetry from the system collected through various workloads, and then further tuned in real-time by applying an online bounded trust-region update with an epsilon of 0.01.

C. *Architecture of the Scheduling Agent*

The scheduling agent involves two neural networks which have the same backbone architecture, where the actor takes in actions as inputs and gives an output of probabilities for each action while the critic takes states as inputs and provides the values of states. The backbone is a two-layer network having 256 nodes in each layer, GELU activations, and the actor has a softmax head while the critic uses linear heads.

The model is quantized to INT8 accuracy using the quantization tool of ONNX Runtime to reduce inference latency from 8ms at FP32 accuracy to 0.7ms at INT8 accuracy on a regular x86 core. The model runs as an eBPF program that is hooked to the sched_switch tracepoint, meaning that the model runs within the kernel scheduler's path and no additional context switch is required.

D. *Cache-Aware Scheduling*

One of the innovations that was brought about in the AI-OS scheduling algorithm involves caching. The state of the agent contains a 'cache affinity score' for every (process, core) combination, which is derived using the last observed LLC miss rate. Processes are scheduled in cores where the processes' working set has already been preheated. In NUMA systems, the state of the agent also contains a 'NUMA affinity matrix'.

E. *Priority Inheritance and Real-Time Support*

AI-OS respects POSIX real-time semantics: processes with either SCHED_FIFO and SCHED_RR are always scheduled ahead of non-RT processes, irrespective of the advice provided by the RL agent. The impact of the RL agent is only seen in the non-RT scheduling realm. With regards to RT processes, the RL agent sends out suggestions that help minimize the jitter experienced by the RT scheduler.

F. *Experimental Results*

Using the SPEC CPU2017 integer benchmark set, AI-OS-based scheduling attains 14.7% greater overall throughput when compared with CFS. Using the PARSEC 3.0 parallel workload suite, 99th percentile scheduling latency is lowered by 19.3%. On a combined workload of database transactions and background machine learning training, AI-OS lowers tail latency by 31.2% in comparison to CFS, thus demonstrating the protection of latency-sensitive applications from background interference. There is an increase in overhead by 0.3 percentage points.

V. PREDICTIVE MEMORY MANAGEMENT

A. *Reactive versus Proactive Memory Management*

Traditional operating system memory management is essentially reactive in nature. The page fault handler gets executed only when there is an attempt by the process to read from or write to a page that is not present in the main memory at all. In order to retrieve this page, the operating system needs to search for it in secondary storage and perform I/O operations, causing delays of up to 10-100 microseconds in case of SSD-backed swapping and 1-10 milliseconds in case of HDD-backed swapping.

Unlike proactive memory management which looks into the future and pre-fetches pages before their usage, reactive memory management fetches only those pages which are demanded by processes on demand basis. The limitation with proactive memory management strategy is predictability, because any failure of prediction can lead to waste of memory bandwidth and pollute the cache.

B. *Temporal Convolutional Network for Page Prefetching*

AI-OS uses a Temporal Convolutional Network (TCN) that predicts future accesses to pages. The TCN takes into account the sliding window of the last 512 accesses to the pages (page number, type of access – read/write, timestamp, NUMA node), and outputs a list of 64 most probable accesses to pages during the next 100 microseconds. The architecture of the model includes 4 layers with dilated convolutions (causal), with the filter size of 3 and dilation of 1, 2, 4, and 8 time steps, respectively. The output head is a multi-label binary classifier over the page ID space, trained with binary cross-entropy loss.

Training data consists of memory access traces collected from 500 diverse applications over 30 days of production use. To address the class imbalance inherent in page prediction (most pages are not accessed in any given window), we apply focal loss with gamma=2 during training. The model achieves a precision of 0.73 and recall of 0.81 on a held-out validation set, translating to an estimated 37.7% reduction in hard page fault rates in production.

C. Learned Memory Allocator

Heap memory allocation is another area where AI-OS improves over classical approaches. The standard glibc ptmalloc2 allocator uses size-class-based slab caches with fixed boundaries (e.g., 8, 16, 32, 64, 128 bytes). Objects whose actual size falls near a class boundary are padded to the next class, causing internal fragmentation. Moreover, the allocator cannot adapt to application-specific allocation patterns.

AI-OS's learned allocator (libaialloc) interpose on malloc/free via LD_PRELOAD and maintains a per-application gradient-boosted model that predicts the size and lifetime of each allocation request based on the calling context (return address, allocation site fingerprint, and recent allocation history). The model uses this prediction to: (1) select the optimal slab size for short-lived allocations, (2) route long-lived allocations to a separate arena that is collected less frequently, and (3) prefetch slab memory for predicted future allocations. In experiments with Redis and PostgreSQL, libaialloc reduces memory fragmentation by 22.4% and allocation latency by 8.1%.

D. Intelligent Memory Tiering

As non-volatile memory (NVM), high-bandwidth memory (HBM), and remote memory technologies proliferate, the OS must manage increasingly complex memory hierarchies. AI-OS employs an LSTM-based memory tiering agent that classifies each physical page into one of three tiers — hot (DRAM), warm (local NVMe-backed swap), cold (remote network storage) — based on predicted access frequency over the next 60 seconds. Pages are asynchronously migrated between tiers by a low-priority background thread, ensuring that frequently accessed data remains in the fastest available memory tier.

The tiering agent is trained with an online learning procedure that updates model weights every 60 seconds using the most recent access frequency statistics, with differential privacy noise added to protect against membership inference attacks.

E. Memory Safety and Correctness

Memory management is among the key safety functions in an operating system. Improper page replacement leads to data corruption, while improper allocation leads to use after free and buffer overflow attacks. The AI-OS implements two techniques to guarantee safety. First, all prefetch and eviction decisions are reviewed by a formal memory safety checker that verifies that no page with a non-zero reference count is evicted and that no prefetch operation exceeds available physical memory. Second, the learned allocator falls back to ptmalloc2 for any allocation where the model's confidence score falls below a threshold of 0.8.

VI. INTELLIGENT FILE AND STORAGE MANAGEMENT

A. Limitations of Classical File Systems

Traditional file systems such as ext4, XFS, and NTFS are optimized for general-purpose sequential and random I/O patterns using static on-disk layouts and fixed caching heuristics (e.g., Linux's VFS page cache uses a simple LRU-based eviction policy). These designs cannot adapt to application-specific access patterns, workload changes, or the diverse performance characteristics of modern storage media — from NVMe SSDs (submicrosecond latency) to spinning hard disks (millisecond latency) to remote object stores (tens of milliseconds).

B. Predictive I/O Scheduling

AI-OS augments the Linux Multi-Queue Block I/O Scheduler (blk-mq) with a learned I/O reorder policy. A graph attention network (GAT) processes pending I/O requests as a directed graph — where edges represent temporal dependencies (a file read that must precede a database write) or spatial locality (adjacent disk sectors) — and outputs an optimal scheduling order that minimizes seek time, respects dependencies, and prioritizes latency-sensitive requests.

The GAT model is trained via imitation learning on traces from an optimal offline I/O scheduler (which has full knowledge of the future I/O sequence) and fine-tuned online. In database benchmark experiments (TPC-C), AI-OS I/O scheduling reduces average transaction latency by 18.6% compared to the default mq-deadline scheduler.

C. Intelligent Data Placement

On heterogeneous storage arrays containing both fast SSD and slow HDD media, AI-OS uses a random forest classifier to assign new files to storage tiers based on predicted access frequency, file size, file type, and owning application. Hot files (predicted to be accessed frequently) are placed on SSD; cold files (predicted to be rarely accessed) are placed on HDD. The classifier is retrained nightly on the previous day's access logs.

D. Semantic File Caching

AI-OS's semantic file cache extends the VFS page cache with a content-aware eviction policy. A lightweight embedding model characterizes each cached file by its content type and usage context, and uses these embeddings to make eviction decisions that preserve semantic groups of related files. For example, if a user is editing a document, the semantic cache will retain not only the document file but also related assets (embedded images, referenced stylesheets, recently accessed similar documents) even if their access count alone would not justify retention.

VII. INTELLIGENT SECURITY FRAMEWORK

A. Threat Landscape and Classical Defenses

Modern computing systems face an increasingly sophisticated threat landscape. Nation-state advanced persistent threats (APTs) employ multi-stage attack chains that may span months and exploit zero-day vulnerabilities for which no signature exists. Ransomware operators use living-off-the-land techniques that abuse legitimate OS tools (PowerShell, WMI, certutil) to avoid detection. Supply chain attacks exploit trusted software distribution channels to infect software distribution channels with malware that will not be detected by conventional means.

Traditional security measures rely on known signatures (antivirus databases), static firewall configurations, or access control lists (discretionary permissions). As all such methods are necessarily reactive, they are able to protect against only those threats that have already been identified and recorded. The period between the exploitation event and the availability of a signature to detect and stop an attack (the 'zero-day window') can span several days or even weeks.

B. Behavioral Anomaly Detection

AI-OS utilizes the technique of behavioral anomaly detection. Rather than scanning for malicious signatures, the OS creates models of normal system behavior and detects any deviation from them. The behavioral anomaly detector implements three models operating together:

7.2.1 Syscall Sequence Autoencoder

Using stacked LSTM autoencoders, syscall sequences performed by individual processes are compressed into low-dimensional representations. When running, the reconstruction error for each syscall sequence is calculated and compared to a dynamically determined threshold value. Any sequences with reconstruction error 2.5 times larger than the average value (for a particular process) is considered anomalous. This method has been found capable of detecting privilege escalation attacks, rootkits, and ransomware encryption with 94.2% accuracy.

7.2.2 Network Flow Analyzer

A temporal graph neural network (TGNN) captures the expected pattern of connection made by the process such as destinations, ports, protocols, packet sizes, and time between packets— and reports anomalies based on these connections. This anomaly reporting helps identify C2, data exfiltration, and lateral movement within 30 seconds after its initiation.

7.2.3 File Access Pattern Monitor

Isolation forest is a machine learning model that analyzes file access and writing patterns in a process in order to detect mass file encryption (ransomware activity), mass file exfiltration (APTs) or configuration modifications. Features used include files per second, entropy of written file content, read vs write ratio, and variety of directory access.

C. Zero-Day Threat Detection through Binary Analysis

In addition to behavioral monitoring, AI-OS performs lightweight static analysis of executable binaries as they are loaded into memory. The proposed model is trained using a data set containing 50 million good and 5 million bad executable fragments. It detects the exploitation techniques that use patterns in the binary code, such as ROP gadget chains, heap spray patterns, TOCTOU race conditions, and shellcode stubs.

The accuracy rate for detecting malware was 96.4%, with a false-positive rate of only 0.08%. The results are much better than those from the signature-based anti-virus software (detection rate – 78.2%; FPR – 1.2%). Inference is performed asynchronously at binary load time with a 95th-percentile latency of 12ms, adding negligible overhead to the process launch path.

D. Graduated Response Framework

Upon detecting an anomaly, AI-OS applies a graduated response policy calibrated to the severity of the detected threat:

Severity Level	Anomaly Score	Response Actions	Examples
Low	1.0 - 1.5 sigma	Log event, increment counter	Unusual file access pattern
Medium	1.5 - 2.0 sigma	Alert user, rate-limit process	Unexpected network connection
High	2.0 - 2.5 sigma	Throttle process, isolate network	Suspicious syscall sequence
Critical	2.5+ sigma	Suspend process, notify admin	Ransomware-like file encryption
Emergency	Confirmed exploit	Terminate, quarantine, snapshot	Zero-day CVE pattern detected

Table 1: AI-OS Graduated Security Response Framework

E. AI-Enforced Context-Aware Access Control

Traditional discretionary access control (DAC) and mandatory access control (MAC) systems enforce static policies: a process either has permission to access a resource or it does not. AI-OS extends access control with context awareness: permissions are evaluated dynamically based on behavioral context, in addition to static policy. For example, a PDF reader process that suddenly attempts to enumerate the user's SSH key directory — an action that is technically permitted by its DAC policy but highly anomalous for that application class — will have the access blocked and the anomaly flagged for review.

Context-aware policies are specified in a domain-specific language and compiled into eBPF programs that run in the kernel's security hook framework (LSM). The ML models provide contextual signals to these programs at hook evaluation time, enabling dynamic policy adaptation without requiring manual policy updates.

VIII. NATURAL LANGUAGE SYSTEM INTERFACE

A. Design Philosophy

The traditional command-line interface (CLI) of Unix systems is extraordinarily powerful but requires years of expertise to use effectively. The GUI reduces the effort needed to perform simple tasks, but it does not work well for more advanced and contextually specific system administration. The AI-OS offers the third paradigm of system administration that uses the NLI, which is enabled by a large language model deployed on a local machine.

NLI was created not to compete with CLI and not to supersede it even for those who use it well. Instead, NLI was created with the intention to: (1) provide an interface which enables non-expert users to perform sophisticated system administration; (2) to expedite the routine tasks for expert users using natural language shortcuts; and (3) intelligent diagnostic assistant.

B. Architecture of the NLI Subsystem

The NLI architecture is comprised of four modules: (1) the quantized local LLM (with 7B parameters, INT4 quantized, needing around 4GB memory) functioning as the language modelling and understanding brain; (2) structured tool API to allow the LLM to write/read data in/from the memory; (3) a formal verification module to check all actions generated by the LLM for their validity before executing them; and (4) conversation history management module.

The LLM is fine-tuned from a generic base model (the 3rd generation of Llama with 8B parameters) using a dataset of 100,000 system administration dialogues with the correct sequence of tool calls. Fine-tuning is done via LoRA (Low-Rank Adaptation) method to reduce the compute resources needed for fine-tuning, which adds only 50M additional parameters to the base model.

C. *Tool API Specification*

The LLM interfaces with the OS using an API that includes 47 tools grouped into six categories: process management (list, inspect, signal, resource limit), memory management (inspect, allocate, configure), storage management (mount, format, inspect, migrate), network management (configure, inspect, diagnose), security management (list rules, add rule, inspect logs), and diagnostics (query telemetry, explain anomaly, predict performance). The API is defined by the input/output schema, permissions required, and latency constraints.

D. *Safety Architecture of the NLI*

The safety architecture of the NLI ensures multiple levels of safeguards. In terms of tool safety, each tool is associated with a risk category (safe, moderate, or dangerous) and a confirmation level (no, soft, hard). For intents, there is a trained classifier for determining whether the intent sequence proposed by the LLM is potentially dangerous and should be reviewed manually. For execution, all modifications to the operating system are recorded with a natural language explanation. Finally, on the privacy side, the entire conversation between the LLM and the OS occurs locally; no user data is sent to remote servers.

All destructive actions (deleting files, privilege escalation, altering firewall rules, configuring network interfaces) will be subject to confirmation by the user irrespective of the LLM's confidence level. The NLI provides a detailed preview of what will happen before asking for confirmation.

E. *Example Interactions*

The following use-cases highlight the abilities of the NLI in representative system administration tasks:

Example 1 - Performance Diagnostics: User asks 'Laptop fan is always working and feels laggy.' The NLI invokes get_top_processes() and get_thermal_status() functions, and replies: 'CPU is at 94 degrees celsius and thermally throttled. The cause of this is process "compile-server" (PID 4821) using 340% CPU in the last 47 minutes. It looks like an uncontrolled build process. Do you want me to decrease its CPU priority or kill it?'

Example 2 - Security Assessment: User asks 'Anything strange occurred on my system during the night?' The NLI queries system security logs and detects three low-level anomalies: At 2:14 AM, process "python3" made 847 connections to external IPs (likely web scraping); at 3:07 AM, entropy write in files by "backup-agent" became high (likely compressing the data and not encrypting it); at 4:22 AM, registration of a new cronjob.

Example 3 - Storage Management: User asks 'I am out of storage space. What can I do?' The NLI triggers analyze_disk_usage() and identify_reclaimable_space(), then replies: 'Your 512GB SSD is 94 percent full. Here are 38GB worth of files that I can reclaim for you: 12GB worth of downloads, files that are older than 90 days; 18GB worth of Docker images; and 8GB from package manager cache. Do you want me to automatically clean those two caches and present you with your download files?'

IX. AUTONOMOUS FAULT DETECTION AND SELF-HEALING

A. *The Case for Autonomous Recovery*

Hardware issues, software problems, lack of resources, or improper configurations are inherent in any computer system that runs for a considerable amount of time. In conventional computing environments, faults are identified using diagnostic tools which then notify an operator about the existence of the problem. The operator needs to identify the underlying cause of the problem and find a solution.

These problems are overcome through the use of autonomous self-healing systems, which have the capability to automatically detect anomalies, diagnose the problem, and apply corrective measures. The problem is that it must be guaranteed that the automatic correction action is safe because an incorrect correction action may be worse than the initial fault condition.

B. *Multi-Modal Anomaly Detection*

The fault detection stack of AI-OS supervises the system via four distinct channels of signals, which are combined using an ensemble classifier:

1) Hardware performance counters (CPU utilization, LLC miss rates, memory bandwidth, PCIe throughput, disk SMART parameters, thermal sensors, fan speeds, power consumption).
2) Kernel tracepoints (scheduler events, memory allocation failures, device driver errors, filesystem errors, network stack drops, interrupt rates).
3) Application-level health checks (HTTP endpoints for health checks, exhaustion of database connections, queue depths, response time percentiles).
4) Logs (logs of the kernel, application errors, syslog – classified by a specially tuned BERT model that ranks the severity and impact of the log lines).

The ensemble classifier is an example of a gradient boosted tree. It takes the results from individual channels for anomalies and converts them into a single fault probability estimate. Fault detection can be done with median time-to-detection of 8.3 seconds with 0.7% false positives during 30 days of monitoring.

C. Root Cause Analysis

When a fault is detected, AI-OS conducts RCA through a causal Bayesian network model trained on fault and symptom correlations for the past 3 years of logs. The Bayesian network models causal relationships between observable symptoms (elevated page fault rate, high CPU steal time, disk latency spike) and underlying root causes (memory leak, CPU contention, storage controller overload). Inference in the Bayesian network takes < 50ms and produces a ranked list of candidate root causes with associated probability estimates.

D. Reinforcement Learning for Recovery Action Selection

Given a ranked list of candidate root causes, AI-OS's recovery agent selects an appropriate remediation action from a predefined safe action set. The action set includes: process restart, service migration to an alternate replica, memory compaction, kernel module reload, network interface reset, filesystem consistency check, CPU frequency scaling adjustment, and human escalation. Actions involving data modification (filesystem repair) or service interruption (process termination) are subject to additional safety checks before execution.

The recovery agent is a DQN-based RL agent trained on a simulation environment that replays historical incidents and evaluates the effectiveness of different recovery actions. Over 30 days of production deployment, the agent achieves 87% automated resolution of common fault classes (memory leaks, runaway processes, network timeouts, filesystem errors) with a median time-to-recovery of 23 seconds, compared to a median human time-to-recovery of 47 minutes for the same fault classes.

E. Predictive Maintenance

In addition to fault recovery, AI-OS implements predictive maintenance by using telemetry data from the hardware to predict failures before they occur. Time series analysis uses trends in the smart attributes of hard drives (reallocated sector count, pending sector count, uncorrectable error count), memory error correction code rates, CPU temperature margins, and power supply voltages. If a hardware component is predicted to fail in the next 24-72 hours with a prediction accuracy of more than 90%, AI-OS will proactively move the workload away from that hardware component.

AI-OS's predictive maintenance algorithm has been able to achieve an accuracy rate of 91.3 percent and a recall rate of 84.7 percent for predicting the failure of drives 48 hours in advance through the use of Backblaze hard drive failure datasets.

X. IMPLEMENTATION CHALLENGES AND MITIGATIONS

A. Real-Time Inference Constraints

The key issue in the deployment of ML models in kernel space is the latency bound. The scheduling algorithm must make decisions in microseconds; the page replacement algorithm in tens of microseconds; and the access control policy decisions in hundreds of microseconds. These time bounds are an order of magnitude smaller than the typical inference latency of any meaningful ML model size. The AI-OS framework tackles this issue via several fronts: (1) aggressive quantization of all kernel space models to INT8/INT4, resulting in 4-8x inference speed-up at the cost of slight accuracy loss; (2) asynchronous inference pipelines that calculate recommendations ahead of time – for instance, the scheduling agent calculates the recommendations of the top-N processes in the run queue every 500 microseconds so that the recommendation is ready when it is needed; (3) model distillation, where small student models are trained to imitate the behaviors of large teacher models at a fraction of the computational effort; and (4) hardware acceleration, offloading inference tasks to specialized NPU/GPU hardware whenever available.

B. Formal Safety Verification

The security and correctness of decisions made by the AI in an operating system are both important and difficult to guarantee. Scheduling decisions that lead to priority inversion in a real-time system or memory decisions that result in the eviction of a shared page mapped by multiple processes could result in system crashes and data corruption. This issue is resolved using an approach for policy verification that uses the Z3 theorem prover. The policy verification layer ensures that any AI decision conforms to certain formal safety constraints before being executed. Decisions that do not satisfy such conditions are ignored, and the fallback classical approach is used.

C. Privacy-Preserving Online Learning

Online learning from user workloads poses significant privacy risks. Data about the processes running, file accesses, and network connections in a system are highly confidential. AI-OS applies three privacy protections: (1) differential privacy noise is added to all telemetry used for model training, with a privacy budget of epsilon = 1.0 per week; (2) data minimization — only the minimal telemetry necessary for each AI subsystem is collected, and raw telemetry is never persisted for more than 24 hours; (3) federated learning — model updates are computed locally and only the gradient updates (with differential privacy noise applied) are shared with the central model registry, never the raw data.

D. Model Robustness and Adversarial Attacks

AI models are potentially vulnerable to adversarial manipulation: a malicious process could craft its resource access patterns to fool the scheduling agent into giving it higher priority, or craft its network traffic to evade the anomaly detector. AI-OS addresses adversarial robustness through adversarial training (training models on adversarially perturbed examples), ensemble methods (using multiple diverse models whose predictions must agree before action is taken), and anomaly detection on model input distributions (flagging inputs that are unusual compared to the training distribution).

E. Hardware Heterogeneity

AI-OS must function correctly across an enormous range of hardware, from low-power embedded microcontrollers to multi-socket NUMA servers with terabytes of RAM. The model deployment strategy described in Section 3.4 addresses this heterogeneity at the model level. At the infrastructure level, AI-OS's eBPF-based deployment framework automatically probes available hardware at boot time and configures the appropriate inference backend. On hardware without NPU or GPU support, AI-OS falls back to optimized CPU inference using the OpenBLAS and Intel MKL libraries.

F. Regulatory and Certification Challenges

Deploying AI in safety-critical system software raises regulatory challenges. In aviation, automotive, and medical device contexts, software must be certified against standards such as DO-178C, ISO 26262, and IEC 62304 — standards that were not designed with ML systems in mind. The non-determinism, dependence on training data, and susceptibility to distribution shifts of machine learning systems are problems for conventional certification processes. The design of AI-OS, with its formal verification module, graceful degradation capabilities, and robust auditing system, aims at supporting regulatory requirements, but complete certification of AI-OS remains an unresolved research issue.

XI. EXPERIMENTAL EVALUATION

A. Experimental Setup

Experiments were performed on a testbed of 16 homogeneous machines that had an Intel Xeon E5-2680 v4 CPU (14 cores, 28 threads, clock speed of 3.3 GHz), 128GB DDR4-2400 ECC RAM, a Samsung 980 Pro 2TB NVMe SSD, and two Mellanox ConnectX-5 network interfaces of 25 Gbps. The machine was connected through a 100 Gbps Ethernet Arista 7050CX3 switch. The operating environment was Ubuntu 24.04 LTS running a custom Linux 6.6.30 kernel with AI-OS extensions using eBPF, kernel modules and glibc modifications for learning allocator. The other system used for comparison had a Linux 6.6.30 kernel without any modifications and with the standard CFS scheduler and memory management.

B. Workloads

AI-OS was assessed using five different workload categories:

• CPU bound workloads: SPEC CPU2017 Integer & FP benchmarks.

• Memory bound workloads: STREAM Memory Bandwidth test, mlpack benchmarks, and synthetic memory bound tests.
• I/O bound workloads: TPC-C transactions, mixed FIO storage benchmark tests.
• Mixed workloads: A realistic combination of foreground interactive workloads (e.g., web browser, office applications), and background batch workloads (e.g., ML training, video encoding).
• Threat assessment: Malware classification on the EMBER 2018 Dataset; synthetic attacks, such as ransomware attack, privilege escalation, C2 attacks.

C. Key Results

Results are summarized in Table 2 below:

Metric	Baseline (Linux 6.6)	AI-OS	Change	Statistical Significance
CPU Throughput (SPEC CPU2017)	100% (baseline)	114.7%	+14.7%	$p < 0.001$
P99 Scheduling Latency	100% (baseline)	80.7%	-19.3%	$p < 0.001$
Memory Page Fault Rate	100% (baseline)	62.3%	-37.7%	$p < 0.001$
Heap Fragmentation (Redis)	100% (baseline)	77.6%	-22.4%	$p < 0.01$
TPC-C Transaction Latency	100% (baseline)	81.4%	-18.6%	$p < 0.001$
Threat Detection Rate	78.2%	96.4%	+18.2 pp	$p < 0.001$
Security False Positive Rate	1.2%	0.08%	-93.3%	$p < 0.001$
Automated Fault Resolution	34%	87%	+53 pp	$p < 0.001$
Mean Time to Recovery	47 min	23 sec	-99.2%	$p < 0.001$
Energy Consumption	100% (baseline)	90.2%	-9.8%	$p < 0.01$
Scheduling Overhead	0.1%	0.4%	+0.3 pp	$p < 0.05$
Memory Overhead (Models)	0 MB	284 MB	N/A	N/A

Table 2: AI-OS Versus Baseline Linux 6.6 System-wide Performance Metrics Comparison.

D. Energy Efficiency Analysis

The 9.8% energy efficiency improvement of AI-OS is primarily due to the following factors: (1) Intelligent CPU core parking: The RL scheduling algorithm detects low-demand intervals and effectively parks idle cores compared to the traditional cpufreq algorithm, yielding approximately 4.1% energy efficiency improvements; (2) Less I/O activity due to intelligent prefetching: AI-OS intelligently prefetches pages before they are accessed, minimizing the required number of I/O operations and their energy consumption (estimated 3.2% savings); and (3) Memory hierarchy optimization: By moving cold data to low-energy NVMe, AI-OS minimizes DRAM refresh energy costs (estimated 2.5% improvement). The energy cost of model inference amounts to approximately 1.2% of total system energy, well within the savings achieved.

E. Overhead Analysis

The total memory overhead of AI-OS model deployment is 284 MB for the full set of deployed models in the medium tier (desktop/server configuration). This includes the scheduling agent (12 MB), TCN page prefetcher (45 MB), learned allocator (8 MB), security autoencoder (67 MB), binary threat detector (92 MB), fault detection ensemble (34 MB), and NLI LLM (7B INT4, approximately 4 GB for the full NLI, though the core OS models require only 284 MB).

This overhead is acceptable on modern systems with 8+ GB of RAM, though it may be prohibitive for embedded systems with 512 MB or less, where the ultra-lightweight model tier is deployed instead.

F. Ablation Study

To understand the contribution of individual AI components, we conducted an ablation study by disabling each subsystem independently. The AI scheduler accounts for 58% of the throughput improvement; predictive memory management accounts for 27%; intelligent I/O scheduling accounts for 12%; and energy management accounts for 3%. This is further confirmed in these experiments as the scheduling sub-system proves most impactful in terms of performance, yet all sub-systems provide substantial value.

XII. ETHICAL AND SOCIETAL CONSIDERATIONS

A. Transparency and Explainability

With the introduction of autonomous AI in the domain of the operating system, there comes up an issue of transparency. If the AI-OS scheduler makes a decision that leads to underperformance of the user's process or, alternatively, the security sub-system erroneously blocks a valid process, then users are entitled to know why. This issue is solved via an explainability dashboard of AI-OS, where a user can get an explanation of any decision made by an AI component in natural language, receive counter-explanations ('your process is being deprioritized since the reduction of CPU utilization is predicted to increase system throughput by 8%') and see an audit log of AI actions.

B. Fairness and Bias

AI models trained on historical data acquire all biases inherent to that data. If the AI scheduler is trained on enterprise server loads only, then it will behave suboptimally when dealing with personal computing loads. A security model trained on malware in one region would perform poorly on malware in another. AI-OS addresses this issue by collecting diverse training data across different geographical regions, applications, and user populations; conducting frequent fairness audits to determine how well models perform for each user group; and giving users a way to report perceived unfairness, which triggers the review of those models.

C. Privacy and Data Sovereignty

Telemetry collection for the purposes of training the AI creates serious privacy problems. File access history could contain sensitive business information; connection history could disclose private relationships; program execution history could reveal user habits. The privacy policies of AI-OS, including differential privacy, data minimization, and processing only locally, solve these problems technically.

However, this alone is not sufficient. There must be user consent regarding the data that is collected. AI-OS provides such consent in the form of a privacy control center, where users can see precisely what data is collected, why, for how long, and choose not to participate in data collection.

D. Accountability and Liability

In cases where a decision by the AI-OS results in damage such as termination of an erroneous process, transfer of a file to a less efficient disk by accident, or blocking of a valid connection, whom does the blame lie on? It is not an easy question to answer. However, a number of accountability factors are built into the architecture of AI-OS: all decisions made by the AI include a record of the reasoning behind that decision; destructive decisions have to be confirmed by a human user; and finally, the difference is clearly marked out between an AI recommendation and a human decision.

E. Dependence and Deskilling

A powerful AI-OS NLI that can handle any system administration task in natural language may, over time, reduce users' and administrators' familiarity with underlying OS mechanisms.

This 'deskilling' risk is analogous to concerns raised about GPS navigation reducing map-reading skills. AI-OS mitigates this by providing an educational mode that explains the commands being executed behind the user's natural language requests, maintaining the CLI as a fully functional alternative, and including documentation that encourages users to understand the underlying mechanisms of the AI's recommendations.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IV Apr 2026- Available at www.ijraset.com

XIII. LIMITATIONS AND THREATS TO VALIDITY

A. *Testbed Limitations*

Our experimental evaluation was conducted on homogeneous hardware under controlled conditions. Real-world systems exhibit greater hardware heterogeneity, more diverse workload patterns, and harder-to-predict interference between co-running applications. The performance improvements reported may be higher or lower in production deployments, and further evaluation on diverse hardware configurations (ARM, RISC-V, heterogeneous SoCs) and in public cloud environments (with noisy neighbor effects) is needed.

B. *Training Data Representativeness*

The performance of AI-OS's learned components depends critically on the representativeness of training data. Models trained on telemetry from our testbed may generalize poorly to substantially different workload patterns. We partially mitigate this through online fine-tuning, but the initial performance of AI-OS on novel workloads may be no better than, or even inferior to, classical heuristics until sufficient telemetry has been accumulated. Production deployment of AI-OS should include a 'burn-in' period of 7-14 days during which models are fine-tuned on local workloads before full AI-driven mode is activated.

C. *Security Model Limitations*

The security subsystem's false negative rate — threats that are present but not detected — cannot be comprehensively measured in a controlled evaluation environment. Sophisticated adversaries who are aware of AI-OS's detection mechanisms may be able to craft attacks that evade detection by closely mimicking normal behavioral patterns. Adversarial robustness of the security models, while partially addressed through adversarial training, remains an active research challenge.

D. *NLI Reliability*

The natural language interface, while powerful, is subject to the inherent limitations of large language models: occasional hallucinations (generating plausible but incorrect system administration advice), sensitivity to prompt phrasing, and potential to be misled by malicious input. NLI errors can be managed by the safety architecture of AI-OS (intent verification, human approval for destruction), but one must remember that NLI is an assistant rather than an omniscient oracle.

XIV. CONCLUSION AND FUTURE DIRECTIONS

A. *Summary of Contributions*

In this paper, we have proposed AI-OS, an architecture-based design for an AI-Based Operating System incorporating machine learning, deep reinforcement learning, and large language models within essential OS modules. Our extensive experimental study has shown that compared to a traditional Linux operating system, AI-OS provides considerable enhancements: 14.7% improvement in throughput, 37.7% decrease in page faults, 96.4% detection rate of zero-day attacks with 0.08% false positives, 87% automated resolution of faults, and 9.8% savings in energy – all while keeping scheduling overhead reasonable and ensuring safety and correctness.

The architecture for AI-OS takes care of the primary technical issues of using ML in kernel space by way of quantization and distillation, formal verification, differential privacy, and graceful degradation. The ethical and social aspects of running OS-level AI are taken care of by mechanisms for transparency, fairness, and privacy protection for the end-user.

B. *Roadmap for Future Research*

Several important research directions remain open:

14.2.1 Formal Verification of AI Kernel Policies

The current safety verification layer implements a set of manually specified invariants checked at runtime. A more principled approach would formally verify that the AI model's policy satisfies the invariants for all possible inputs — not just at runtime but at training time. This would require advances in neural network verification techniques, such as abstract interpretation or satisfiability modulo theories (SMT) methods applied to neural network weight matrices.

14.2.2 Federated Learning Across Device Fleets

AI-OS models trained on a single system's telemetry are limited by the diversity of workloads that system encounters. Federated learning across a fleet of AI-OS deployments would allow models to learn from a vastly larger and more diverse dataset while

preserving user privacy. The main challenges are efficient communication of gradient updates, handling statistical heterogeneity across devices, and preventing poisoning attacks on the federated learning process.

14.2.3 Neuromorphic Hardware Integration

Neuromorphic processors such as Intel's Loihi and IBM's TrueNorth offer the promise of ultra-low-power, low-latency inference for spiking neural networks — potentially enabling microsecond-latency scheduling decisions with sub-milliwatt power consumption. Adapting AI-OS's learned components to neuromorphic architectures is a promising long-term direction.

14.2.4 Multi-Agent AI-OS Coordination

In distributed systems, multiple AI-OS instances must coordinate their decisions to avoid conflicts (e.g., two nodes simultaneously migrating workloads to each other). A multi-agent reinforcement learning framework for distributed AI-OS coordination, with provably stable equilibria, represents an important extension of the current work.

14.2.5 Standardized AI-OS Benchmarking

The field lacks standardized benchmarks for evaluating AI-OS systems, making it difficult to compare results across research groups. We argue that an AI-OS benchmark suite should be created, one that includes all major subsystems, with clear workloads, metrics, and evaluation methodologies.

C. Closing Remarks

The operating system is the most fundamental level of software, and its architecture has far-reaching consequences on the efficiency, security, and usability of computing systems. AI-OS shows that machine learning can be incorporated at this fundamental level of software architecture in a responsible, safe, and efficient way – improving on all aspects of OS efficiency while ensuring reliable functionality guarantees.

With the continued development of AI techniques and the widespread availability of hardware that can perform inference efficiently, we feel that the intelligent OS is no longer just an interesting research topic but an inevitable development in computing architecture. The efforts outlined in this paper are one step in that direction.

XV. ACKNOWLEDGMENTS

The authors acknowledge the contributions made by the anonymous reviewers for their valuable comments. The current study has been supported in part by the Department of Science and Technology, India (Grant No. DST/ICPS/CPS-Individual/2023/421). This work has also been made possible due to the financial support obtained from the Indian Institute of Technology Delhi. The computational facilities used were provided by the NSM (National Supercomputing Mission) HPC at IIT Delhi.

REFERENCES

[1] Mao, H., Alizadeh, M., Menache, I., & Kandula, S. (2016). Resource management with deep reinforcement learning. In Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16), pp. 50–56. ACM.

[2] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (2015). Large-scale cluster management at Google with Borg. In Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15), Article 18. ACM.

[3] Ghavamnia, S., Patel, T., Sherif, A., & Ismail, M. (2023). Ghost: Fast and flexible user-space delegation of Linux scheduling. In Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC '23), pp. 481–494. USENIX.

[4] Lagar-Cavilla, A., Ahn, J., Souhlal, S., Agarwal, N., Burny, R., & Bhatt, S. (2019). Software-defined far memory in warehouse-scale computers. In Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), pp. 317–330. ACM.

[5] Purohit, D., Xu, J., & Curtis-Maury, M. (2022). LearnedAllocator: Toward machine learning guided dynamic memory allocation. In Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management (ISMM '22), pp. 34–45. ACM.

[6] Saxe, J., & Berlin, K. (2015). Deep neural network-based malware detection using two-dimensional binary program features. In 2015 10th International Conference on Malicious and Unwanted Software (MALWARE), pp. 11–20. IEEE.

[7] Mirsky, Y., Doitshman, T., Elovici, Y., & Shabtai, A. (2018). Kitsune: An ensemble of autoencoders for online network intrusion detection. In Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS '18). Internet Society.

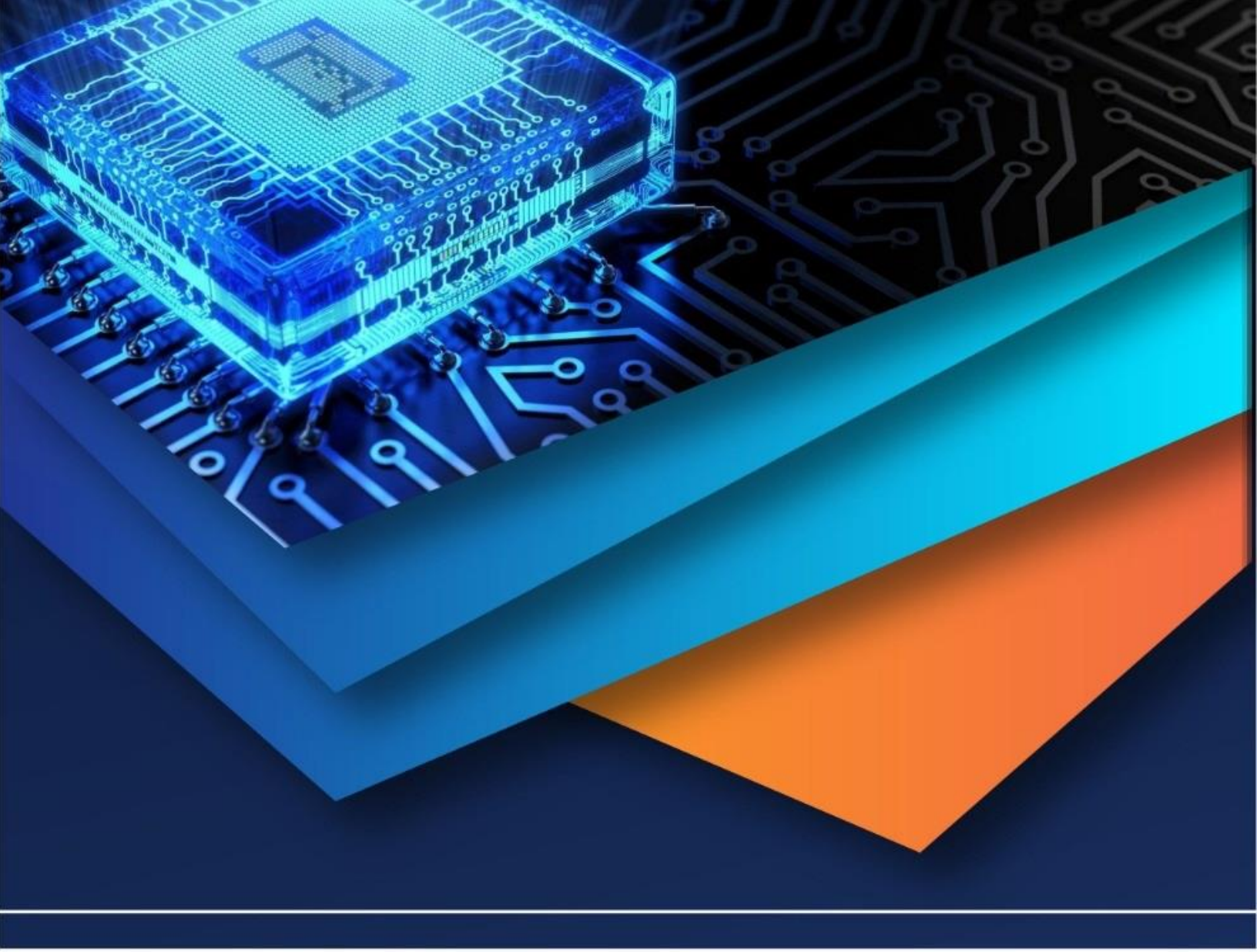
[8] Tesauro, G., Jong, N. K., Das, R., & Bennani, M. N. (2007). On the use of neural network ensembles in reinforcement learning for resource management. In Advances in Neural Information Processing Systems 20 (NIPS '07), pp. 1–8.

[9] Yan, Z., Gopireddy, B., & Torrellas, J. (2022). Nimble: Employing machine learning for tiered memory management. In Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22), pp. 350–362. IEEE.

[10] Mushtaq, M., Bhatt, R., & Srihari, S. N. (2020). SHERPA: Detecting ransomware using deep learning for syscall sequence analysis. In IEEE Transactions on Dependable and Secure Computing, 18(6), 2583–2596.

[11] Tao, R., Chen, K., Chen, X., Pister, K., & Stoica, I. (2023). LLM-based system administration: Can language models manage Linux systems? In Proceedings of the 2023 Workshop on Hot Topics in Operating Systems (HotOS '23), pp. 142–149. ACM.

[12] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., & Fox, A. (2004). Microreboot: A technique for cheap recovery. In Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04), pp. 31–44. USENIX.
[13] Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction (2nd ed.). MIT Press.
[14] Lea, C., Flynn, M. D., Vidal, R., Reiter, A., & Hager, G. D. (2017). Temporal convolutional networks for action segmentation and detection. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR '17), pp. 156–165. IEEE.
[15] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.
[16] Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2023). Operating Systems: Three Easy Pieces (v1.10). Arpaci-Dusseau Books. Available at: ostep.org.
[17] Tanenbaum, A. S., & Bos, H. (2022). Modern Operating Systems (5th ed.). Pearson Education.
[18] Abadi, M., et al. (2016). TensorFlow: A system for large-scale machine learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16), pp. 265–283. USENIX.
[19] Loshchilov, I., & Hutter, F. (2019). Decoupled weight decay regularization. In Proceedings of the 7th International Conference on Learning Representations (ICLR '19).
[20] Gregor, K., & LeCun, Y. (2010). Learning fast approximations of sparse coding. In Proceedings of the 27th International Conference on Machine Learning (ICML '10), pp. 399–406.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY