CPU Scheduling Kernel Visualizer with Machine Learning–Based Algorithm Recommendation

Meesala Sree Sai Nath
Department of Computer Science
230103
Rishihood University
Sonipat, India
meesala.s23csai@nst.rishihood.edu.in

Pugazhendhi J
Department of Computer Science
230066
Rishihood University
Sonipat, India
pugazhendhi.j23csai@nst.rishihood.edu.in

Akula Jithendranath
Department of Computer Science
230120
Rishihood University
Sonipat, India
akula.j23csai@nst.rishihood.edu.in

***Abstract*—Understanding CPU scheduling algorithms is foundational to operating systems education, yet their abstract, dynamic nature is difficult to convey through static diagrams. This paper presents the CPU Scheduling Kernel Visualizer, a full-stack interactive platform that simulates, visualizes, and analytically compares eight CPU scheduling algorithms in real time. The system is built on two independent scheduler implementations: a lightweight `Scheduler` in `scheduler.py` powering a standalone Python CLI, and a full `SimulationEngine` in `kernel/engine.py` powering the web API with process state transitions, kernel event logging, and context switch tracking. A key practical contribution is the dual-mode terminal interface: (1) `terminal_ui.py`, a zero-dependency Python application rendering colored ASCII process tables, ready queues, and Gantt charts entirely in the shell, and (2) an embedded xterm.js browser terminal with five live API commands. The web platform provides a 3D animated landing page, a three-panel Kernel View with a 3D CPU chip and live event log, real-time analytics, and an ML-powered recommender using a Random Forest classifier trained on 5000 synthetic workloads across 14 statistical features, achieving 87–92% cross-validated accuracy.**

***Index Terms*—CPU scheduling, operating systems, kernel visualization, terminal UI, ANSI CLI, Gantt chart, FCFS, SJF, SRTF, MLFQ, Round Robin, machine learning, Random Forest, discrete-event simulation, React, FastAPI, xterm.js**

I. Introduction

CPU scheduling is one of the most critical functions of an OS kernel, directly influencing throughput, waiting time, response time, and CPU utilization [1]. Despite its importance, scheduling is commonly taught through static diagrams and pseudocode—methods that fail to convey real-time scheduler dynamics.

This work is motivated by three observations: (1) existing educational simulators support only one or two algorithms [5]; (2) practitioners lack tooling to empirically compare algorithms on specific workloads; (3) the combination of kernel-level tracing, CLI terminal simulation, and ML-guided recommendation has not been achieved in any existing open-source tool.

Contributions include: two independent scheduler implementations (`scheduler.py` for CLI, `kernel/engine.py` for the web API); a standalone Python terminal application (`terminal_ui.py`) operable entirely without a browser; an embedded xterm.js browser terminal with live API commands; eight scheduling algorithms under a pluggable policy architecture; a three-panel Kernel View with 3D CPU chip and kernel event log; real-time analytics and comparison dashboard; and a Random Forest ML recommender trained on 14 workload features.

II. Related Work

Process Scheduling Simulator [5] supports FCFS, SJF, and Round Robin but lacks preemptive variants, MLFQ, and ML capability. CPU Scheduling Algorithms Simulator [6] adds priority scheduling but omits a terminal interface and AI recommendation. OS-Sim [7] targets course labs with limited real-time streaming. Prior ML work on scheduling has focused on RL for cloud datacenters [8] and neural networks for real-time systems [9], but none combine interactive visualization, a CLI terminal, and workload-based ML recommendation in a single accessible tool.

III. Scheduling Algorithms

All eight algorithms share the tick-based simulation engine. Table I summarizes their key properties.

TABLE I
Scheduling Algorithm Properties

Algorithm	Preemptive	Selection Criterion
FCFS	No	Arrival order (FIFO)
SJF	No	Minimum burst time; optimal avg wait [3]
SRTF	Yes	Minimum remaining time; optimal preemptive avg wait
LJF	No	Maximum burst time
LRTF	Yes	Maximum remaining time
Priority	No	Lowest priority value
RR	Yes	FIFO with configurable time quantum q
MLFQ	Yes	3-level queues with quanta $[4, 8, \infty]$; demotion on expiry [2]

FCFS suffers from the convoy effect; SJF is provably optimal for non-preemptive policies but requires known burst times; SRTF achieves optimal preemptive average wait time at the cost of high context switches; MLFQ approximates oracle burst knowledge through progressive queue demotion, making

it the policy of choice for general-purpose kernels. LJF and LRTF are included for educational contrast. Average waiting time is $\bar{W} = \frac{1}{n}\sum_{i=1}^{n} W_i$ and turnaround time is $T_i = F_i - A_i$ where F_i and A_i are finish and arrival times respectively.

IV. System Architecture

Table II summarizes all system components. The React 18 SPA uses a shared `ProcessContext` for state persistence across seven pages. The FastAPI backend exposes `/tick`, `/run`, `/compare`, `/recommend`, `/train`, and `/state` endpoints under `/api/v2`.

TABLE II
System Architecture: Component Overview

Layer	Component	Role
Frontend	Landing Page	3D Spline background, algorithm cards
Frontend	Simulator	Gantt chart, ready queue, process table
Frontend	Kernel View	3D CPU chip, kernel log, state table
Frontend	Analytics	5 real-time streaming charts
Frontend	Comparison	8-algo bar charts + CSV export
Frontend	Recommender	ML recommendation with confidence scores
Frontend	Browser Terminal	xterm.js with 5 live API commands
Backend	kernel/engine.py	SimulationEngine: PCB, kernel log, context switches
Backend	scheduler.py	Lightweight Scheduler for terminal CLI
Backend	Algorithms	8 pluggable scheduling policies
Backend	AI Pipeline	Feature engineering, RF classifier
Standalone	terminal_ui.py	Python CLI: ANSI tables, ASCII Gantt

V. Simulation Engine

Each process is represented as a PCB tracking: PID, arrival/burst/remaining time, priority, state (NEW/READY/RUNNING/TERMINATED), start/finish times, wait time, response time, turnaround time, quantum used, and MLFQ queue level. At each clock tick the engine: (1) admits newly arrived processes (NEW→READY); (2) handles quantum expiry and policy-based preemption; (3) dispatches the next process via the policy's `select_next()`; (4) executes one tick and extends the Gantt chart; (5) increments wait times for all READY processes; (6) records per-tick metrics; (7) advances the clock; (8) checks for completion. This separates policy (selection) from mechanism (execution), enabling any of the eight algorithm classes to be swapped in without modifying the engine. Context switches are counted only on PID-to-PID CPU transitions. Consecutive same-process Gantt entries are merged into $\{pid, startTime, endTime\}$ to minimize payload.

VI. Platform Pages

A. Landing Page

The landing page (Fig. 1) is a scrollable full-page layout. The hero section overlays a fixed WebGL 3D Spline scene with

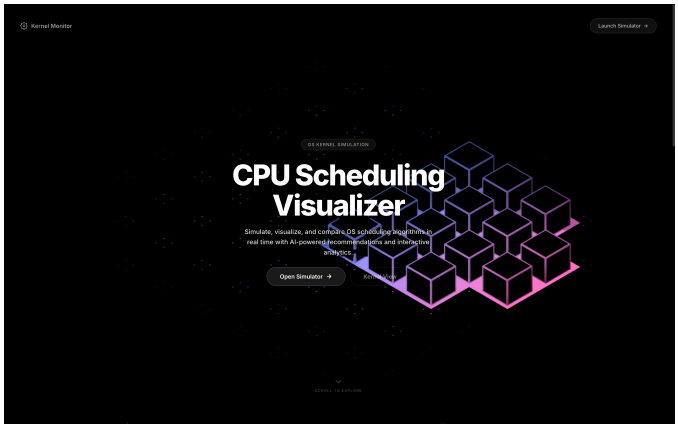

Fig. 1. Landing page: 3D Spline background, hero section, demo Gantt, features grid, and algorithms grid. [Add screenshot as screenshot_landing.png]

title, badge, subtitle, and CTA buttons. Below it, an animated demo Gantt card, a six-card features grid (glassmorphism cards for Kernel View, 8 Algorithms, Tick Simulation, ML Recommender, Live Analytics, Terminal), and eight SpotlightCard algorithm cards with hover spotlight effects complete the page.

B. Simulator Page

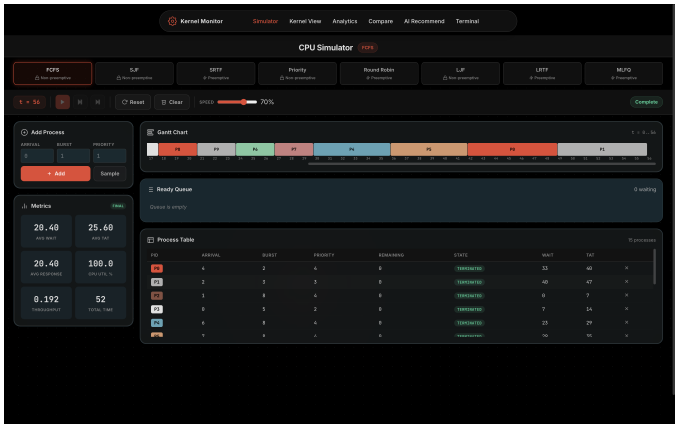

Fig. 2. Simulator page: algorithm selector, control panel, Gantt chart, animated ready queue, and process state table. [Add screenshot as screenshot_simulator.png]

The simulator (Fig. 2) is the primary interface. A scrollable pill selector covers all 8 algorithms with a time quantum field for RR/MLFQ. A control panel provides Start, Pause, Step, Reset, Run-to-End, and a 0.5x–8x speed slider. The sidebar hosts a process form and a 6-metric dashboard. The main area renders a live color-coded Gantt chart, animated ready queue cards, and a real-time process state table with state badges (NEW / READY / RUNNING / TERMINATED).

C. Kernel View Page

The Kernel View (Fig. 3) uses a three-panel layout. The left panel shows an animated tick counter, a 2×2 stats grid (Context Switches, Queue Size, Completed, Total), and active

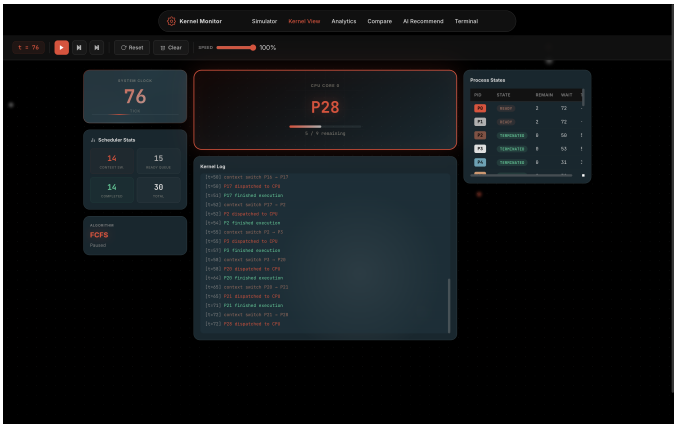

Fig. 3. Kernel View: system clock and stats (left), 3D CPU chip and kernel event log (center), process state table (right). [Add screenshot as screenshot_kernel.png]

algorithm status. The center panel features `CpuChip3D`—a pulsing 3D chip that glows when a process is running and shows a progress arc—and below it the `KernelLog` rendering the last 50 timestamped events color-coded by type: `arrive` (cyan), `dispatch` (green), `preempt/demote` (orange), `context_switch` (yellow), `complete` (green), `idle` (dim). For MLFQ, `MLFQDeepView` renders three labeled queue lanes. The right panel shows an animated process state table with the RUNNING row highlighted.

D. Analytics, Comparison, and Recommender Pages

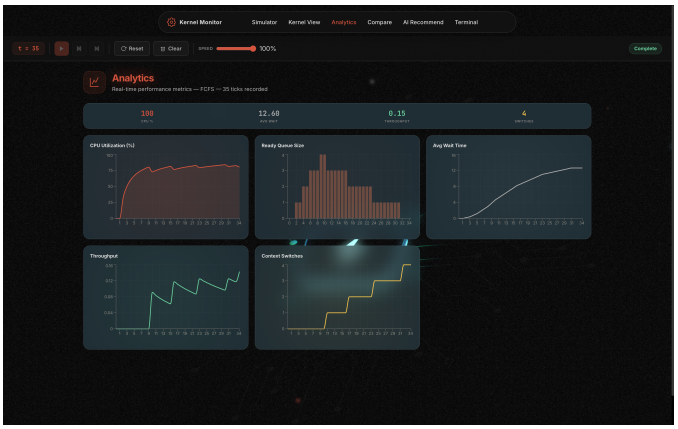

Fig. 4. Analytics page: five real-time streaming charts (CPU utilization, ready queue size, avg wait, throughput, context switches). [Add screenshot as screenshot_analytics.png]

The **Analytics page** (Fig. 4) renders five streaming Recharts time-series: CPU Utilization (area), Ready Queue Size (bar), Avg Wait Time (line), Throughput (line), and Context Switches (line), backed by the backend's per-tick `metricsHistory`. The **Comparison dashboard** (Fig. 5) runs all eight algorithms on the same workload via a single `/compare` call, renders grouped bar charts with best-performer highlighted at full opacity, and exports a `scheduler_comparison.csv`.

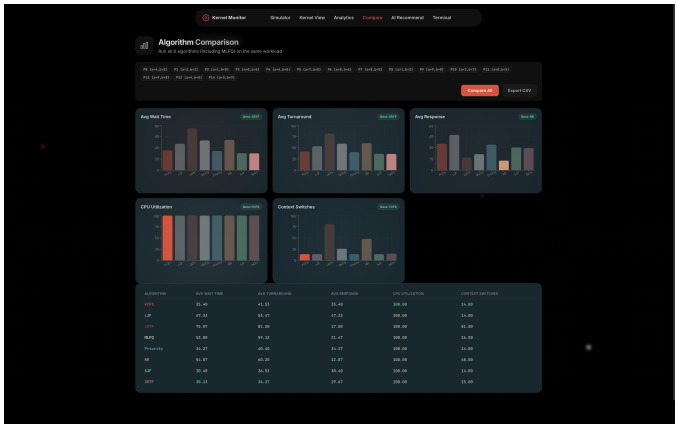

Fig. 5. Comparison page: all 8 algorithms ranked across 5 metrics; best algorithm per metric highlighted at full opacity. [Add screenshot as screenshot_comparison.png]

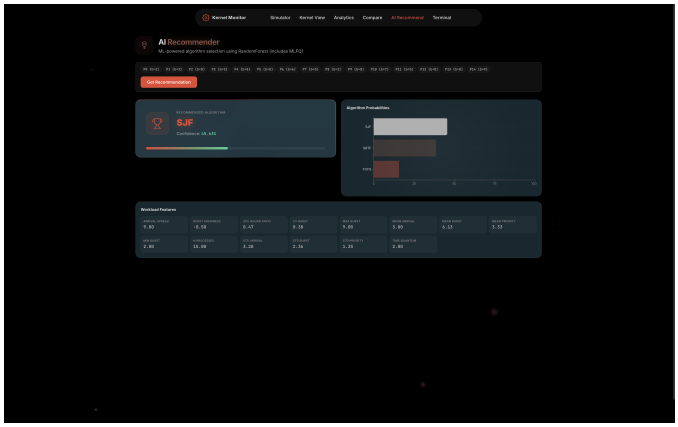

Fig. 6. AI Recommender: 14-feature workload vector, recommended algorithm, and probability distribution. [Add screenshot as screenshot_recommender.png]

The **Recommender page** (Fig. 6) displays the extracted 14-feature vector, top recommended algorithm with confidence score, and a sorted probability list across all eight algorithms.

VII. Terminal Interface: Dual-Mode CLI

In real operating systems, kernel scheduler interaction occurs through the terminal—`top`, `htop`, `perf sched`, and `/proc` reads. This platform ships a dual-mode terminal interface reflecting that practical reality.

A. Standalone Scheduler Core (scheduler.py)

`api/scheduler.py` is a self-contained scheduling module requiring only Python 3, with zero network or web dependencies. It provides a `_ALGORITHM_FN` dict mapping indices 0–7 to pure selector functions, `MLFQ_QUANTUMS = [4, 8, None]` (where `None` on Q_2 means FCFS—processes run to completion, matching the classic MLFQ rule [2]), and a `Scheduler` class using plain Python dicts for process state. The deliberate separation from `kernel/engine.py` keeps the terminal tool portable and zero-dependency while the web

engine carries the full complexity of PCB objects, kernel logging, and API serialization.

B. Standalone Terminal Application (terminal_ui.py)

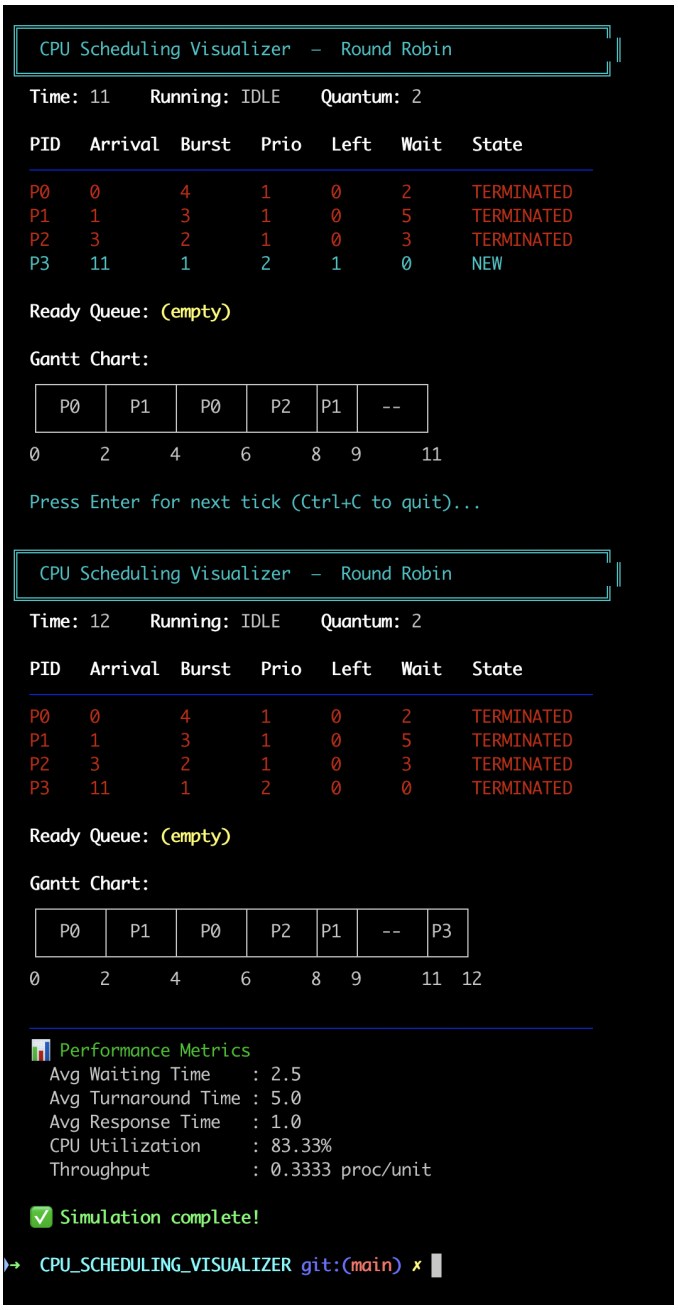

Fig. 7. Standalone terminal UI: ANSI-colored process table, ready queue, and ASCII Gantt chart during Round Robin simulation. [Add screenshot as screenshot_terminal_cli.png]

`api/terminal_ui.py` (Fig. 7) is a complete CLI CPU scheduling simulator running on UNIX, macOS, and Windows with no package installation beyond the Python standard library.

At each tick the terminal clears the screen and re-renders four panels: (1) a bordered header (cyan) showing algorithm, tick time, running process, and quantum; (2) a color-coded process table—cyan for READY, green for RUNNING, red for TERMINATED; (3) the ready queue as `P2 → P4 → P0` in yellow; (4) an ASCII Gantt chart built with box-drawing characters () scaled by execution duration with a timestamp ruler below. In **step mode** (default) the tool waits for Enter after each tick, mirroring `gdb` single-step. In **auto-run mode** (`-r`) ticks advance at the configured delay. On completion a metrics summary is printed (avg wait, turnaround, response time, CPU utilization, throughput). Example invocations:

TABLE III
TERMINAL_UI.PY COMMAND-LINE ARGUMENTS

Flag	Default	Description
`-a`	interactive	Algorithm (0=FCFS ...7=MLFQ)
`-q`	2	Time quantum for RR/MLFQ
`-s`	false	Use built-in 5-process sample workload
`-r`	false	Auto-run (no Enter key needed)
`-d`	0.5s	Delay between ticks

```
python3 api/terminal_ui.py -a 2 -s -r -d 0.3  # SR
python3 api/terminal_ui.py -a 7 -q 4 -s       # M
python3 api/terminal_ui.py -a 4 -q 3 -r -d 0.5 # R
```

C. Browser Terminal (xterm.js)

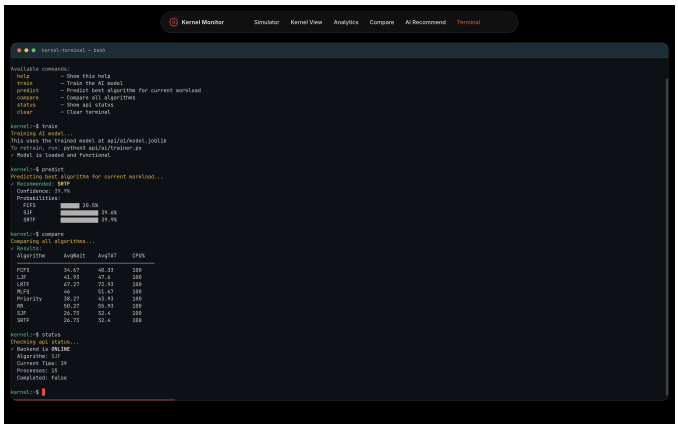

Fig. 8. Browser terminal: `predict` command with Unicode probability bar chart and `compare` ASCII table output. [Add screenshot as screenshot_terminal.png]

The seventh web page (Fig. 8) embeds xterm.js with JetBrains Mono font, a kernel-themed ANSI color scheme (red cursor, green prompt `kernel:~$`), and FitAddon for responsive resizing. Five commands make asynchronous `fetch()` calls to the backend: `help`, `status` (backend liveness + algorithm/time/process count), `train` (ML model validation), `predict` (ML recommendation with Unicode block-char probability bars), and `compare` (ASCII results table for all 8 algorithms). The browser terminal complements `terminal_ui.py` for users already in the web simulator who want CLI-style queries without a local Python installation.

VIII. ML-Powered Algorithm Recommendation

A. Feature Engineering

The recommendation task is formulated as multi-class classification: a 14-dimensional feature vector $\mathbf{x} \in \mathbb{R}^{14}$ maps to $\hat{y} \in \{\text{FCFS, SJF, SRTF, LJF, LRTF, Priority, RR, MLFQ}\}$. The feature vector (Table IV) captures workload statistics: process count, burst moments (mean, std, CV, min, max, skewness), arrival spread, priority statistics, CPU-bound ratio, and time quantum. The CV σ/μ is highly discriminative—low-variance workloads favor FCFS/SJF while high-variance favors SRTF/MLFQ. Burst skewness identifies long-tail distributions; CPU-bound ratio captures process mix.

TABLE IV
14-Dimensional Workload Feature Vector

Feature	Description
$n\_processes$	Number of processes
$mean\_burst$	Mean burst time
std_burst	Std. deviation of bursts
cv_burst	Coefficient of variation (σ/μ)
min/max_burst	Min and max burst times
$mean/std\_arrival$	Arrival time statistics
$arrival\_spread$	Range of arrival times
$mean/std\_priority$	Priority statistics
$burst\_skewness$	Third-moment skewness
cpu_bound_ratio	Fraction above median burst
$time\_quantum$	Configured quantum

B. Training and Inference

A `DatasetGenerator` produces 5000 synthetic workloads (3–20 processes, bursts 1–50, random arrivals and priorities). For each, all eight algorithms are run and the best is selected by composite metric rank (avg wait, turnaround, CPU utilization). A Random Forest [4] is trained with 200 trees, max depth 15, stratified 80/20 split, and 5-fold cross-validation (`n_jobs=-1`). At inference, the workload is serialized into the feature vector, the `joblib` model returns class probabilities, and the top recommendation is shown with confidence score and sorted probability list.

IX. Implementation

Frontend: React 18 + Vite; React Router v6; Framer Motion (spring-physics animations); Recharts (streaming charts); xterm.js + FitAddon; Spline (3D WebGL iframes). Custom Aceternity-inspired components (`Card3D`, `SpotlightCard`, `CpuChip3D`, `DotGrid`, `GlowText`, `BeamLine`, `AnimatedNumber`) provide a dark glassmorphism design language.

Backend: Python 3.11 + FastAPI + Uvicorn; scikit-learn [10] (Random Forest, cross-validation); joblib (model I/O); XGBoost (optional). The `SchedulerPolicy` abstract base class exposes `select_next()`, `should_preempt()`, `is_preemptive`, and `uses_quantum`, enabling fully polymorphic engine operation.

Deployment: Backend on Render (https://cpu-scheduling-visualizer-euxn.onrender.com); frontend as a static site with CORS. Terminal UI requires only Python 3 with no additional packages.

X. Experimental Evaluation

A. Algorithm Performance

Table V shows comparative results on an 8-process workload (P0–P7, arrivals 0–7, bursts 2–9, priorities 1–5; RR quantum=3, MLFQ base quantum=2).

TABLE V
Algorithm Performance Comparison

Algorithm	Avg Wait	Avg TAT	CPU%	C.S.
FCFS	14.25	19.75	100.0	7
SJF	9.50	15.00	100.0	7
SRTF	**7.13**	**12.63**	100.0	16
LJF	20.38	25.88	100.0	7
LRTF	23.75	29.25	100.0	23
Priority	11.88	17.38	100.0	7
RR (q=3)	12.63	18.13	100.0	18
MLFQ	8.25	13.75	100.0	14

SRTF achieves the lowest average wait (7.13) at the cost of the highest context switches (16). MLFQ achieves near-SRTF performance (8.25 avg wait) with fewer context switches (14) by approximating burst knowledge via demotion. LJF and LRTF perform worst as expected.

B. ML Accuracy and Terminal Performance

The Random Forest achieves ≈88–92% test accuracy and ≈0.87 ± 0.02 on 5-fold cross-validation. Top features by importance: `cv_burst`, `mean_burst`, `burst_skewness`, `std_burst`, `cpu_bound_ratio`. The browser terminal API round-trip averages 120–300ms. The standalone `terminal_ui.py` completes a 20-process simulation (800+ ticks) in under 1 second locally; tick-step API mode averages 120–180ms.

XI. Discussion and Conclusion

The platform addresses multiple learning modalities: Gantt chart for temporal intuition, kernel log for event-level tracing (analogous to `ftrace`/`perf sched`), 3D CPU chip for spatial intuition, and analytics charts for statistical properties. The standalone terminal is particularly significant for OS education—students run the simulation in the same shell environment used for kernel programming and systems debugging, with step-mode interaction mirroring `gdb` single-step. The dual terminal design embodies the Unix philosophy: *the scheduling core should be accessible regardless of interface*, portable to headless servers, SSH sessions, embedded systems labs, and classroom demonstrations.

Limitations include user-supplied burst times (no exponential averaging), single-core CPU model, no I/O bursts, and synthetic-only ML training. Future work: multi-core SMP simulation, I/O burst support, real workload trace import from `/proc`, RL-based adaptive online scheduling, and configurable MLFQ parameters.

This paper presented the CPU Scheduling Kernel Visualizer: a full-stack platform with two independent scheduler implementations, a dual-mode terminal interface (standalone Python CLI + browser xterm.js), eight scheduling algorithms, a three-panel Kernel View with live event log and 3D CPU chip, and a Random Forest recommender achieving 88–92% accuracy on 14 workload features. The platform is open-source at https://github.com/sainath2212/CPU_SCHEDULING_VISUALIZER and deployed live; the terminal UI requires only Python 3.

ACKNOWLEDGMENT

The authors acknowledge the open-source contributions of the React, FastAPI, Recharts, Framer Motion, xterm.js, and scikit-learn communities and thank the OS research community, particularly [1] and [2].

REFERENCES

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ: Wiley, 2018.

[2] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, v1.10. Madison, WI: Arpaci-Dusseau Books, 2023. [Online]. Available: https://pages.cs.wisc.edu/~remzi/OSTEP/

[3] E. G. Coffman Jr. and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[4] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001.

[5] T. Shinners-Kennedy and J. O'Sullivan, "A web-based CPU scheduling simulator for undergraduate OS courses," in *Proc. ITiCSE*, Bologna, Italy, 2006, pp. 296–300.

[6] A. Belal, "A visualisation tool for CPU scheduling algorithms," *Int. J. Comput. Sci. Network Security*, vol. 5, no. 3B, pp. 35–43, 2005.

[7] G. Back, W. Titch, and C. Van Riper, "Processes in educatOS: An educational operating system," in *Proc. ACM SIGCSE*, Dallas, TX, 2002, pp. 38–42.

[8] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. HotNets*, Atlanta, GA, 2016, pp. 50–56.

[9] G. Beccari, S. Bonura, G. Franceschinis, and C. Montangero, "Scheduling in real-time systems," *Real-Time Systems*, vol. 5, no. 4, pp. 367–389, Nov. 1993.

[10] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.