AI-Based Smart Resource Manager for System Optimisation Using Machine Learning and Web Technologies

Aman Baheti[1], Dr. R. Savita[2]
[1]*Department of MCAB RV College of Engineering Bengaluru, India*
[2]*Professor, Department of MCA RV College of Engineering Bengaluru, India*

***Abstract*—Modern laptops and personal computers run dozens of applications at the same time, and users often have no idea why things start slowing down or when it is going to get worse. Standard tools like Windows Task Manager or Linux top only show a snapshot of what is happening right now — they do not predict what is coming or tell you what to do about it. This paper describes a system we built called the AI-Based Smart Resource Manager which monitors CPU, RAM, Disk, and Battery usage in real time, uses a Random Forest machine learning model to predict resource usage five seconds ahead, and generates smart suggestions through a four-engine recommendation system. We also built a web version using Flask so that any user can open a browser, register, and start seeing their own personalised dashboard without downloading any project code. A role-based access system keeps each user's data private while letting an admin see aggregated health scores across all connected machines. We tested the system with multiple users and found that the Random Forest model achieved a Mean Absolute Error of 4.3% and an R^2 score of 0.87. The web app was deployed on Render.com and worked correctly across three simultaneous users during our tests.**

***Index Terms*—anomaly detection, Flask, machine learning, predictive analytics, psutil, Random Forest, role-based access, system resource monitoring, web dashboard**

I. INTRODUCTION

Anyone who uses a laptop while running multiple applications at once has probably noticed the machine getting slow without any clear reason. The problem is that current monitoring tools are reactive — they only show what is happening at this exact moment. There is no built-in way to know whether your CPU is about to spike in the next few seconds, or which specific background application is quietly eating up your RAM. For most users, by the time they notice something is wrong, the performance damage has already happened.

We started working on this project because we personally faced this problem during our MCA coursework when running multiple development environments, browsers, and virtual machines at the same time. We wanted something that could watch the system continuously, predict where things were heading, and tell us what to do before the slowdown hit. That idea turned into the AI-Based Smart Resource Manager described in this paper.

The system has two main parts. The first is a desktop monitoring pipeline built in Python that collects CPU, RAM, Disk, and Battery readings every five seconds using the psutil library [9], stores them in a SQLite database, trains three machine learning models on the collected history, and uses the best one to predict future resource usage. The second part is a web application built with Flask [10] that extends everything to multiple users over the internet. Each user runs a lightweight agent script on their PC that pushes data to a central server, which runs the trained model on their data and serves a personalised dashboard through a browser.

The main contributions of this paper are: (1) a four-engine recommendation system combining rule-based alerts, ML-based predictive warnings, process-level identification, and Z-score anomaly detection; (2) a web architecture with role-based privacy design where admin can monitor aggregate health scores without accessing individual user data; and (3) experimental

validation with multiple simultaneous users on real hardware.

II. RELATED WORK

Resource monitoring and performance prediction have been studied extensively in cloud computing and edge system contexts. Alahmari et al. [1] showed that ensemble learning methods like Random Forest outperform single-model approaches for CPU utilisation forecasting in cloud environments, which directly influenced our algorithm choice. Hwang et al. [2] compared threshold-based, isolation forest, and Z-score methods for anomaly detection in system metrics and found that Z-score works well for low-frequency univariate sensor streams like CPU and RAM percentages.

Chahal et al. [3] used machine learning specifically for CPU workload prediction in cloud systems and highlighted the importance of lag features and rolling averages for capturing temporal patterns, which we adopted in our feature engineering. Torres and Patel [4] built a real-time system optimisation tool using AI-based predictive monitoring and found that combining rule-based and ML-based logic gives better user experience than either alone, which is the philosophy behind our four-engine recommendation design.

Roy and Banerjee [5] investigated LSTM networks for CPU and RAM prediction and achieved longer prediction horizons than traditional models, suggesting a clear upgrade path for our current approach. Verma and Joseph [6] addressed anomaly detection for smart computing and showed that statistical methods adapted to individual baselines perform better than universal thresholds, which is why we compute Z-scores against each user's personal rolling history.

What most of these systems have in common is that they run on a single machine or managed cloud infrastructure. None of them address the scenario of a personal or small-office environment where an administrator wants to monitor multiple user machines simultaneously without complex setup. That gap is what our web-based multi-user architecture fills.

III. SYSTEM ARCHITECTURE

A. Desktop Pipeline

The desktop version is structured as a five-stage pipeline. The monitoring module (monitor.py) uses psutil to read CPU percentage per core, RAM, disk usage, disk I/O speed, battery status, and running processes every five seconds. The collection module (collector.py) saves each reading to a SQLite database and a CSV file simultaneously. The ML module (ai_model.py) engineers features, trains three models, and saves the best to disk. The prediction module (predictor.py) loads the saved model and generates next-step predictions from recent database rows. The recommendation engine and Tkinter dashboard integrate everything and present results to the user.

B. Web Application Architecture

The web version follows a three-tier architecture. The agent tier (agent.py) runs on each user's machine, collects metrics using psutil, and sends them to the Flask server via HTTP POST every five seconds. The application tier is a Flask server on Render.com that handles session-based authentication, runs the Random Forest model server-side, generates recommendations, and serves HTML dashboards. The data tier is SQLite managed by Flask-SQLAlchemy.

The privacy boundary is enforced at the API layer. The /api/user/live endpoint returns full data to the logged-in user including process details and recommendations. The /api/admin/users endpoint returns only the derived health score and online status, deliberately excluding process lists and raw metric values. The health score formula is: health = (100−CPU)×0.35 + (100−RAM)×0.35 + (100−Disk)×0.20 + Battery×0.10.

IV. MACHINE LEARNING COMPONENT

A. Feature Engineering

Raw percentage values alone do not tell a model much about trends. We create three lag features for each target metric using pandas shift() to produce values from one, two, and three readings ago. We also compute rolling averages over windows of three and five readings. The hour and minute of day are included as features because CPU usage often follows daily patterns. The target variable is the value in the next reading created using shift(−1). The train-test split uses shuffle=False to preserve chronological order, preventing data leakage from future readings into the training set [7].

B. Model Selection and Evaluation

We trained three models: Linear Regression as a baseline, Decision Tree with max_depth=10, and Random Forest with 100 estimators. In our experiments the Random Forest consistently achieved the lowest MAE across both CPU and RAM prediction tasks. This is expected because the ensemble of 100 trees cancels out much of the noise present in the spiky irregular time-series data that CPU and RAM usage produces. The trained model is saved to disk using pickle and loaded into Flask server memory at startup. During testing we encountered one failure case where the model returned a predicted value above 100% during extreme CPU spikes caused by the model extrapolating beyond the training range. This was fixed by clipping predictions to [0, 100] using numpy clip.

V. RECOMMENDATION ENGINE

The recommendation engine runs four independent sub-engines whose outputs are merged and ranked. The rule-based engine generates immediate alerts when CPU, RAM, or Disk exceed configurable thresholds (CRITICAL above 90%, WARNING above 75%). These work from the first reading without needing a trained model. The ML-based engine compares the predicted next value against warning thresholds, generating proactive alerts before a spike happens. This is the key advantage over purely reactive tools.

The process engine scans the running process list and identifies specific applications consuming more than 20% CPU or more than 500 MB of RAM, naming the application and its process ID for direct action. The anomaly detection engine computes a Z-score for each reading against the user's rolling history of the last thirty readings. A Z-score above 2.0 indicates more than two standard deviations above the user's personal average and is flagged as a statistical anomaly [2][6]. All outputs are combined, deduplicated, and sorted with CRITICAL first, then WARNING, INFO, and TIP.

VI. RESULTS AND DISCUSSION

A. Comparison with Existing Tools

Table I compares the proposed system with commonly used monitoring tools. The main differentiator is the combination of ML prediction, proactive alerts, anomaly detection, and web-based multi-user access with privacy enforcement in a single free system.

TABLE I Comparison With Existing Monitoring Tools

Capability	Task Mgr	Glances	Netdata	Proposed
ML Prediction	No	No	No	Yes
Proactive Alerts	No	No	No	Yes
Anomaly Detection	No	No	Yes	Yes
Web Multi-user	No	No	Yes	Yes
Privacy Boundary	N/A	N/A	No	Yes
Free to Use	Yes	Yes	Yes	Yes

B. Model Evaluation

We collected 500 rows of system metrics from a laptop running Windows 11 during typical MCA coursework activity. The data was split 80/20 chronologically. Table II summarises the validation results. Random Forest achieved a MAE of 4.3% on CPU prediction, meaning on average the predicted next CPU value was within 4.3 percentage points of the actual value. The R^2 score of 0.87 indicates the model explains 87% of the variance in the test set, which is sufficient for the practical purpose of warning users when a spike is approaching.

TABLE II System Validation Results

Metric	Target	Result
RF MAE – CPU Prediction	< 8.0%	4.3%
RF R^2 Score	≥ 0.80	0.87
Dashboard Refresh Rate	≤ 3s	3s
Agent Push Latency (LAN)	≤ 2s	0.4s
Privacy Boundary	100%	100%
Test Case Pass Rate	≥ 95%	96.3%

C. Web Application Testing

We tested the web application by running three instances of agent.py simultaneously on three different machines on the same network, all pushing data to a single Flask server on Render.com. All three users appeared as online in the admin dashboard within 15 seconds of starting their agents. The process kill

feature worked correctly — clicking Kill in the browser caused the target process to terminate on the remote machine within 8 seconds.

Two test cases initially failed. First, on a desktop PC without battery hardware, the monitoring module crashed with an AttributeError because the code accessed battery.percent without first checking if psutil.sensors_battery() returned None. This was fixed by adding a None check before accessing any battery attribute. Second, the Excel export initially showed raw integer timestamps instead of formatted dates, fixed by correcting the strftime format string. These failures and their fixes give us confidence that the system has been genuinely tested rather than just theorised.

VII. CONCLUSION

We built and tested an AI-Based Smart Resource Manager that combines real-time monitoring, machine learning prediction, and smart recommendations in both desktop and web-based form. The Random Forest model performed well enough to be genuinely useful for early warning of CPU and RAM spikes. The four-engine recommendation system gives users specific, actionable information rather than generic alerts. The web version makes the whole system accessible from a browser without complex setup on the user's side, and the role-based privacy design ensures that administrative visibility does not compromise individual user data.

Future work would focus on replacing the Random Forest with an LSTM network for longer prediction horizons, adding GPU and network traffic monitoring, implementing email and SMS alerts, and migrating from SQLite to PostgreSQL for persistent cloud storage. The approach of combining rule-based and ML-based logic in a privacy-respecting multi-user web system is worth exploring further in subsequent work.

ACKNOWLEDGMENT

The authors would like to thank Dr. R. Savita, Assistant Professor, Department of MCA, RV College of Engineering, Bengaluru, for her guidance and support throughout this project. The Department of MCA, RVCE provided the resources and lab facilities used during testing.

REFERENCES

[1] A. Alahmari, B. Nassif, and F. Azzeh, “A Comparison of Machine Learning Methods for Predicting CPU Usage of Microservices,” IEEE Access, vol. 9, pp. 119705–119719, 2021.

[2] J. Hwang, Y. Kim, S. Oh, and B. Lee, “Anomaly Detection in Computer System Monitoring Using Statistical Approaches,” J. Network and Computer Applications, vol. 58, pp. 80–91, 2015.

[3] D. Chahal, S. Harous, and M. Guizani, “Machine Learning-Based CPU Workload Prediction in Cloud Systems,” IEEE Access, vol. 12, pp. 14560–14575, 2024.

[4] L. Torres and M. Patel, “Real-Time System Optimization Using AI-Based Predictive Monitoring,” IEEE Access, vol. 13, pp. 12561–12577, 2025.

[5] P. Roy and S. Banerjee, “Intelligent CPU and RAM Usage Prediction Using LSTM Networks,” J. Systems Architecture, vol. 145, pp. 102931, 2025.

[6] N. Verma and A. Joseph, “AI-Based Anomaly Detection and Resource Allocation for Smart Computing Systems,” Future Internet, vol. 17, no. 1, pp. 22, 2025.

[7] F. Pedregosa et al., “Scikit-learn: Machine Learning in Python,” J. Machine Learning Research, vol. 12, pp. 2825–2830, 2011.

[8] T. Chen and C. Guestrin, “XGBoost: A Scalable Tree Boosting System,” Proc. 22nd ACM SIGKDD, San Francisco, CA, 2016, pp. 785–794.

[9] psutil Development Team, “psutil Documentation,” 2024. [Online]. Available: https://psutil.readthedocs.io. [Accessed: May 2026].

[10] Flask Development Team, “Flask Documentation v3.1,” 2024. [Online]. Available: https://flask.palletsprojects.com. [Accessed: May 2026].

[11] S. Kaur and P. Singh, “Resource Prediction and Optimization Using Hybrid Machine Learning Models,” Expert Systems with Applications, vol. 245, pp. 122815, 2024.

[12] R. Gupta and N. Saxena, “AI-Enabled Process Scheduling for Smart Operating Systems,” Int. J. Intelligent Systems, vol. 39, no. 3, pp. 1450–1467, 2024.