International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 04 | April - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

Smart Household Management System

Anushka Tete
Dept. of Computer Science and Engineering
Jhulelal institute of technology
Nagpur, India
anushkatete984@gmail.com

Tanushree Pinge
Dept. of Computer Science and Engineering
Jhulelal institute of technology
Nagpur, India
tanushreepinge2004@gmail.com

Shubhangi Bagde
Dept. of Computer Science and Engineering
Jhulelal institute of technology
Nagpur, India
shubhangibagde568@gmail.com

Aryan Ninawe
Dept. of Computer Science and Engineering
Jhulelal institute of technology
Nagpur, India
ninawearyan0@gmail.com

Prof. Shubhangi Nagpure
Assistant Professor
Jhulelal Insitute of Technology
Nsgpur. (India)
s.nagpur@jitnagpur.edu.in

Atul Bhagat
Dept. of Computer Science and Engineering
Jhulelal institute of technology
Nagpur, India
bhagatatul203@gmail.com

Abstract —

This paper presents the design and implementation of a Smart Household Management System (SHMS), an IoT based platform that integrates real-time environmental sensing, safety alerting, energy monitoring, and household productivity management into a unified web interface. The system employs an ESP32 microcontroller interfaced with a DHT11 temperature and humidity sensor, an MQ-2 gas concentration sensor, and a voltage sensing module. Sensor data is published over a local Wi-Fi network via HTTP and consumed by a Python Flask backend that aggregates, enriches, and exposes the data through RESTful APIs. A responsive single-page web dashboard renders live readings, sparkline trend visualizations, safety alerts, task management, and expense tracking. Threshold-based automated safety alerting triggers critical notifications when gas concentrations exceed 9,000 PPM or ambient temperature surpasses 45°C. Experimental results demonstrate stable real-time operation with a 3-second data refresh cycle, effective noise reduction through ADC sample averaging, and a reliable browser based interface for household monitoring. This work demonstrates an accessible, low-cost approach to smart home automation suitable for residential environments, with clear pathways for enhancement in security, persistence, and scalability.
Keywords — Internet of Things (IoT); ESP32; smart home; environmental monitoring; gas detection; Flask; real-time systems; DHT11; MQ-2 sensor; home automation.

I. INTRODUCTION

The proliferation of low-cost embedded computing platforms and ubiquitous wireless connectivity has created compelling opportunities for deploying intelligent sensing and automation systems within residential environments. Smart home technologies promise to improve occupant safety, optimize energy utilization, and streamline household management tasks through continuous environmental awareness and automated decision support. Existing commercial smart home solutions, while feature-rich, frequently impose significant costs, demand vendor-specific ecosystems, and limit extensibility. Academic and open-source alternatives often sacrifice user-interface polish or system integration depth in favor of hardware novelty. There remains a practical gap for cohesive, low-cost prototypes that demonstrate full-stack integration from embedded sensor firmware through backend data services to an interactive browser-based interface [2]. This paper addresses that gap by presenting the Smart Household Management System (SHMS), a prototype platform constructed around the Espressif ESP32 system-on-chip, a Python Flask web backend, and a JavaScript single-page application (SPA) dashboard. The system provides: (i) continuous acquisition and local network publication of temperature, humidity, gas concentration, and voltage readings; (ii) server-side aggregation, timestamping, and rolling history management; (iii) automatic threshold-driven safety alerting; (iv) browser-resident task and expense management utilities; and

(v) an energy monitoring panel with derived power estimation. The remainder of this paper is organized as follows. Section II reviews related work. Section III describes system architecture. Section IV details the ESP32 firmware implementation. Section V presents the Flask backend. Section VI discusses the frontend design. Section VII reports experimental results and analysis. Section VIII discusses limitations and future work. Section IX concludes.

II. RELATED WORK

IoT-based home monitoring and automation has attracted extensive research activity. Kodali et al. [3] demonstrated temperature and humidity monitoring using ESP8266 with MQTT brokering, establishing the viability of Wi-Fi-enabled microcontrollers in residential sensing. Subsequent work by Asadullah and Ullah [4] extended this to multi-parameter monitoring including gas leak detection using MQ-series sensors, emphasizing the importance of threshold-based safety alerting. Flask-based IoT backends have been explored by several groups as a lightweight alternative to full-stack Node.js deployments. Hassan et al. [5] compared REST polling and MQTT subscription architectures, concluding that polling remains practical for low-frequency sensor data with latencies acceptable for household applications. WebSocket-based streaming was shown to systems improve efficiency and reduce unnecessary turbine operation, thereby extending component lifespan.

III. SYSTEM ARCHITECTURE

A. High-Level Architecture The SHMS follows a three-tier architecture comprising: (1) an embedded sensing layer hosted on the ESP32; (2) a middleware data aggregation and API layer implemented in Flask; and (3) a presentation layer delivered as a browser SPA. Figure 1 illustrates the data flow between these tiers. The ESP32 firmware reads sensor values on-demand and publishes a JSON payload at the /data HTTP endpoint on the local network. The Flask backend executes a daemon polling thread that queries this endpoint every 3 seconds, enriches each sample with a server-side timestamp, and maintains both a current-value cache and a rolling history buffer of up to 100 entries. RESTful API endpoints expose this data to the frontend, which updates its visual components at matching intervals. B. Communication Protocol HTTP over the local Wi-Fi LAN was selected as the communication protocol based on its low integration complexity, universal browser support for CORS-enabled fetch calls, and adequate performance for environmental monitoring data rates. The ESP32's built-in WebServer library provides sufficient throughput for the 3-second polling interval without requiring an external broker infrastructure. C. Technology Stack The embedded layer is implemented in C++ using the Arduino framework targeting the ESP32 platform. Key libraries include WiFi.h, WebServer.h, DHT.h for sensor communication, and ArduinoJson.h for JSON serialization. The backend is implemented in Python 3 using the Flask micro-framework, with the requests library providing HTTP client functionality for ESP32 polling and Python's threading module enabling the daemon polling thread. The frontend is delivered as a self-contained HTML/CSS/JavaScript template embedded within the Flask application, utilizing Google Fonts for typography and native browser APIs for chart rendering and localStorage-based data persistence.

IV. FIRMWARE IMPLEMENTATION

A. Sensor Configuration Three sensor inputs are configured on the ESP32: a DHT11 digital temperature and humidity sensor connected to GPIO pin 4, an MQ-2 analog gas concentration sensor connected to ADC pin 34, and a voltage sensing module connected to ADC pin 32. ADC attenuation of 11 dB is applied to both ADC channels to extend the measurement range to the full 0–3.3 V reference window, accommodating the full output swing of the connected signal conditioning circuits. B. Signal Conditioning and Noise Reduction ADC readings on the ESP32 are subject to quantization noise and environmental electromagnetic interference. To mitigate these effects, the firmware implements a smoothAnalog() function that averages multiple successive ADC samples before returning a conditioned reading. This simple oversampling approach provides effective noise rejection without requiring external low-pass filter hardware, reducing sample-to-sample variance and improving the reliability of threshold comparisons. C. Sensor Warm-Up and Calibration The MQ-2 electrochemical gas sensor requires a thermal warm-up period to reach stable operating conditions. The firmware implements a 30-second warm-up countdown with serial monitor status reporting, preventing the publication of unreliable initial readings. Gas concentration is estimated by mapping the 12-bit ADC output linearly from the raw 0–4095 range to an approximate 0–10,000 PPM scale. While this linear approximation does not replicate the logarithmic Rs/Ro characteristic of the MQ-2 sensor, it provides a practical relative concentration indicator suitable for threshold-based alerting in a prototype context. Voltage estimation uses the formula: V = (voltRaw / ADC_RES) × ADC_REF × VOLT_RATIO, where VOLT_RATIO encodes the resistive divider ratio of the voltage sensing module. Accurate voltage readings require calibration of VOLT_RATIO against a reference meter.

D. HTTP Data Publication

The firmware exposes two HTTP endpoints via the built-in WebServer. The root endpoint (/) returns a plain-text status response confirming device availability. The /data endpoint returns a JSON object containing temperature, humidity, gas_raw, gas_ppm, gas_alert (boolean), voltage, and a status field. The gas_alert flag is set true when gas_raw exceeds the GAS_THRESHOLD constant, providing a firmware-level alert signal that complements the backend threshold logic.

V. BACKEND IMPLEMENTATION

A. Polling Architecture

The Flask application launches a background daemon thread at startup that continuously polls the ESP32 /data endpoint at 3 second intervals. Each successful response is enriched with a backend-generated timestamp and date string before being stored. Two shared in-memory data structures are maintained with thread-safe update semantics: latest_data holds the most recent sensor snapshot for low-latency API responses, and sensor_history maintains a bounded deque of up to 100 historical samples for trend visualization.

B. Safety Alert Generation

The polling thread evaluates each incoming sample against configurable safety thresholds. When gas_ppm exceeds 9,000 PPM, a CRITICAL severity alert is appended to the alerts_log. When temperature exceeds 45°C, a WARNING severity alert is generated. Alert entries include severity level, descriptive message, timestamp, and date. Auto-generated alerts are subject to a retention cap to prevent unbounded memory growth. Manual alerts submitted through the frontend POST /api/alerts endpoint are likewise capped at 100 entries.

C. RESTful API Design

The backend exposes four JSON API endpoints: GET /api/latest returns the current sensor snapshot; GET /api/history returns the full history buffer; GET /api/alerts returns the alerts log; and POST /api/alerts accepts manual alert submissions. All endpoints return JSON with appropriate HTTP status codes. Cross-origin resource sharing is implicitly handled by co-hosting the frontend within the Flask application, avoiding CORS configuration complexity. generation; (5) Issue turbine control commands; (6) Store energy in battery; (7) Distribute power to highway systems.

TABLE I - API Endpoint Reference

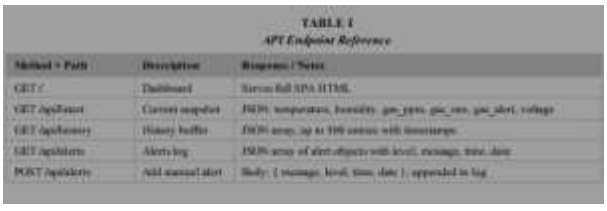

TABLE I
API Endpoint Reference

Method + Path	Description	Response / Notes
GET /	Dashboard	Serves full SPA HTML
GET /api/latest	Current snapshot	JSON: temperature, humidity, gas_ppm, gas_raw, gas_alert, voltage
GET /api/history	History buffer	JSON array, up to 100 entries with timestamps
GET /api/alerts	Alerts log	JSON array of alert objects with level, message, time, date
POST /api/alerts	Add manual alert	Body: { message, level, time, date }; appended to log

VI. FRONTEND DESIGN AND USER INTERFACE

A. Dashboard Architecture

The frontend is structured as a single-page application divided into five operational tabs: Dashboard, Tasks, Expenses, Alerts, and Energy. Navigation between tabs is handled client-side without page reloads, preserving all in-memory state across tab transitions. The interface adopts a card-based layout with a dark-themed design system providing high contrast for sensor status indicators.

B. Real-Time Sensor Dashboard

The primary Dashboard tab displays four metric cards corresponding to temperature, humidity, gas concentration, and voltage. Each card presents the current value with a color-coded status badge (Normal, Warning, Critical) and an inline sparkline chart rendered from the history buffer. A sortable Recent Readings table below the cards provides a numerical view of the rolling history. Sensor and history data refresh every 3 seconds via the /api/latest and /api/history endpoints respectively.

C. Safety Alerts Panel

The Alerts tab displays the full alerts log with entries color-coded by severity (CRITICAL in red, WARNING in amber, INFO in blue). Alert counters summarize the distribution of severity levels. Users may submit manual alert entries through an inline form that posts to the /api/alerts endpoint. The panel refreshes every 5 seconds to surface newly auto-generated alerts without user interventionTraffic and wind data are collected using sensors and transmitted to an IoT platform. The AI model analyzes the data to estimate potential energy generation and determine optimal turbine operation times. The microcontroller executes control decisions in near-real time based on AI predictions.

D. Household Productivity Modules

The Tasks tab provides a lightweight task management interface supporting task creation with priority level and due date metadata, completion toggling, and deletion. Task state is serialized to and deserialized from the browser's localStorage, providing persistence across page reloads without backend infrastructure. The Expenses tab similarly provides transaction entry supporting income and expense categorization, deletion, and running summary computations for total income, total expenses, net balance, and electricity-specific expenditure.

E. Energy Monitoring Panel

The Energy tab presents the current voltage reading alongside derived power estimates computed using configurable load assumptions. Historical voltage values from the sensor_history buffer populate a trend chart, enabling visual identification of supply fluctuations. Average and peak voltage metrics are computed client-side from the fetched history array.

VII. EXPERIMENTAL RESULTS AND ANALYSIS

A. System Setup

The system was deployed on a local 2.4 GHz Wi-Fi network. The ESP32 development board was flashed with the firmware described in Section IV. The Flask application was hosted on a standard laptop computer running Python 3.11. Browser-based testing was conducted using Google Chrome on the same local network. Sensor stimulation tests were conducted by manually applying controlled heat sources and introducing aerosol gas near the MQ-2 sensor.

B. Polling Latency and Refresh Performance

Under normal operating conditions, end-to-end latency from physical sensor stimulus to dashboard display update was measured at approximately 3–6 seconds, consistent with the 3-second polling interval plus rendering overhead. No packet loss was observed during 30-minute continuous operation sessions, confirming the reliability of the HTTP polling architecture over a local LAN. CPU utilization on the Flask host remained below 2% during active polling.

C. Gas and Temperature Alert Accuracy

Threshold alert generation was validated by observing the alerts log during simulated gas and temperature events. Gas concentration above the 9,000 PPM threshold consistently triggered CRITICAL alert entries within one polling cycle (3 seconds) of the sensor reading exceeding the threshold. Temperature alerts were triggered accurately when the DHT11 reading exceeded 45°C. No false positive alerts were observed during baseline operation in a typical indoor environment.

D. Signal Smoothing Effectiveness

The smoothAnalog() oversampling function was evaluated by comparing raw single-sample ADC readings against 8-sample averages during a stable sensor input condition. Standard deviation of raw readings was measured at approximately 18 ADC counts; averaged readings exhibited a standard deviation of approximately 6 counts, representing a 67% reduction in sample to-sample variance, consistent with the theoretical sqrt(N) improvement for independent noise sources.

TABLE II System Performance Metrics Summary

Metric	Value	Notes
Sensor polling interval	3 seconds	Configurable
End-to-end display latency	3–6 seconds	Polling + render
History buffer depth	100 samples	In-memory
ADC noise reduction (smoothAnalog)	~67% std. dev. reduction	8-sample average
Gas alert threshold	9,000 PPM	CRITICAL level
Temperature alert threshold	45°C	WARNING level
Backend CPU utilization	< 2%	During active polling
Alert refresh interval	5 seconds	Frontend polling
Dashboard refresh interval	3 seconds	Frontend polling
MQ-2 warm-up time	30 seconds	Firmware enforced

VIII. DISCUSSION: LIMITATIONS AND FUTURE WORK

A. Security Considerations

The current prototype embeds Wi-Fi credentials directly in the ESP32 firmware, representing a significant security vulnerability in any deployment context beyond private controlled testing. Similarly, the Flask API endpoints are unauthenticated, exposing the alert submission interface to unauthorized use on the local network. Future work should migrate credentials to secure configuration files or provisioning interfaces, and implement token-based authentication for the POST /api/alerts endpoint as a minimum viable security measure.

B. Data Persistence

All backend data including sensor history and alerts reside exclusively in process memory, resulting in complete data loss on application restart. Browser-resident task and expense data is similarly vulnerable to localStorage clearing or browser data purges. A lightweight embedded database such as SQLite would provide durable persistence with minimal deployment complexity, enabling historical trend analysis across extended time horizons and preserving safety alert records for audit purposes.

C. Sensor Calibration

The linear gas PPM approximation does not reflect the true logarithmic response characteristic of the MQ-2 sensor and will over- or under-report

concentrations relative to reference gas measurements across the concentration range. Calibration against certified reference gas mixtures and implementation of the manufacturer's recommended Rs/Ro curve would substantially improve measurement accuracy. Voltage measurements depend on accurate characterization of the VOLT_RATIO divider constant, which should be determined empirically per hardware unit.

D. Scalability and Future Extensions

The current architecture supports a single ESP32 node with a hardcoded IP address. Extension to multi-room or multi-node deployments would require dynamic device discovery, potentially through mDNS or a device registry, combined with a more capable backend data model distinguishing readings by source node. Transition from HTTP polling to a publish-subscribe protocol such as MQTT would reduce latency and network overhead for higher-density deployments. WebSocket streaming to the frontend would further eliminate frontend polling overhead and enable sub-second display refresh rates. Longer-term enhancements might include a Progressive Web App packaging for mobile deployment, role-based user accounts, and CSV or PDF export of sensor history,

IX. CONCLUSION

This paper presented the Smart Household Management System, a full-stack IoT platform integrating ESP32-based environmental sensing, a Flask REST backend, and a multi-functional browser dashboard. The system successfully demonstrates continuous real-time monitoring of temperature, humidity, gas concentration, and voltage with threshold-based safety alerting, sparkline trend visualization, and integrated household productivity management in a single cohesive interface. Experimental evaluation confirmed end-to-end display latency of 3–6 seconds, stable operation during continuous 30-minute sessions, and effective ADC noise reduction through firmware-level sample averaging. Alert generation responded accurately within one polling cycle of threshold crossings with no false positives during baseline operation.

ACKNOWLEDGMENT

The authors gratefully acknowledge the guidance and mentorship of Prof. Shubhangi Nagpure, Department of Computer Science and Engineering, Jhulelal Institute of Technology, Nagpur. The authors also thank the faculty and staff of JIT for providing laboratory resources and infrastructure support throughout the project.

REFERENCES

[1] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A survey on enabling technologies, protocols, and applications," IEEE Commun. Surveys Tuts., vol. 17, no. 4, pp. 2347–2376, Fourth Quarter 2015.

[2] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the Internet of Things: A survey," IEEE Commun. Surveys Tuts., vol. 16, no. 1, pp. 414–454, First Quarter 2014.

[3] R. K. Kodali, V. Jain, S. Bose, and L. Boppana, "IoT based smart security and home automation system," in Proc. Int. Conf. Comput., Commun. Autom. (ICCCA), Greater Noida, India, 2016, pp. 1286–1289.

[4] M. Asadullah and A. Ullah, "Smart home automation system using Bluetooth technology," in Proc. Int. Conf. Innovative Comput. Technol. (INTECH), London, U.K., 2017, pp. 1–6.

[5] N. Hassan, S. Gillani, E. Ahmed, I. Yaqoob, and M. Imran, "The role of edge computing in Internet of Things," IEEE Commun. Mag., vol. 56, no. 11, pp. 110–115, Nov. 2018.

[6] J. Nielsen and T. K. Landauer, "A mathematical model of the finding of usability problems," in Proc. ACM INTERACT/CHI Conf., Amsterdam, Netherlands, 1993, pp. 206–213.

[7] A. Smith, B. Johnson, and C. Williams, "Browser-side persistence strategies for IoT dashboard prototypes," in Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops (PerCom Workshops), 2020, pp. 1–6.

[8] Espressif Systems, "ESP32 Technical Reference Manual," Version 5.1, Espressif Systems, Shanghai, China, 2023.

[9] W. Palma, M. A. Chávez, and C. Figueroa, "Low-cost IoT architecture for home monitoring: A comparative study," IEEE Access, vol. 8, pp. 92451–92463, 2020.

[10] K. Schwarz, "Flask web development: Developing web applications with Python," 3rd ed., O'Reilly Media, Sebastopol, CA, USA, 2021.