A WEB-BASED IOT MONITORING FRAMEWORK SUPPORTING INDUSTRY 5.0

LINA SULSKA[1], ARTEM KOLIESNIK[1], MICHAL DUHANCIK[1], JAN PITEL[1], MAREK KOCISKO[1]

[1]Technical University of Kosice, Faculty of Manufacturing Technologies with the seat in Presov, Slovakia

DOI: 10.17973/MMSJ.2026_11_2026042

lina.sulska@tuke.sk

This article presents the design and implementation of a software-based IoT monitoring system with interactive data visualization. The system simulates a pneumatic piston controlled by a virtual PLC server, transmits process variables through the OPC UA communication protocol, and displays real-time and historical data through a web interface. The backend is implemented in Python using FastAPI library; the frontend uses HTML, CSS, and JavaScript. The architecture follows a three-layer IoT model: a data-generation layer (OPC UA server), a collection and storage layer (OPC UA client, JSON), and a visualization layer (web application). The system requires no physical hardware and validates the full communication chain from process simulation to user display. Results confirm data transfer at 200 ms polling intervals, correct piston position derivation from chamber volumes, and usable historical querying across custom time ranges. The solution provides a foundation for deploying remote human-process interfaces aligned with Industry 5.0 principles.

KEYWORDS

IoT, process monitoring, OPC UA, PLC simulation, data visualization, web interface, Industry 5.0.

1 INTRODUCTION

Manufacturing plants operating under the Industry 5.0 paradigm place human–machine collaboration at the centre of production strategy, framing the industrial environment around human-centric, sustainable, and resilient principles [Alves 2023, Ghobakhloo 2023]. Reaching those goals depends directly on the communication layer between automation systems and operators — it must carry enough semantic context for operators to make informed decisions in real time [Yang 2024, Kant 2025].

OPC Unified Architecture (OPC UA) addresses the interoperability and semantic modelling requirements that this communication layer demands. Performance benchmarks confirm that OPC UA client–server exchanges meet latency and throughput requirements for factory-floor monitoring on both general-purpose and resource-constrained hardware [Ladegourdie 2022, Lee 2021, Cavalieri 2013]. The protocol carries typed, discoverable data structures that give higher-level applications enough semantic context to reason about process state without vendor-specific adaptors, and with time-sensitive networking extensions it supports both soft real-time control and data logging on a shared physical network [Trifonov 2023]. The importance of reliable and accurate monitoring is further demonstrated in pneumatic actuator research, where dedicated measurement systems integrating sensors, PLC-based control, and automated data acquisition are required to obtain repeatable process data for actuator characterisation and validation [Kondrat 2025].

SCADA systems and HMIs form the supervisory and visualisation layer above OPC UA field communication. Usability evaluations of deployed SCADA displays find that most score poorly against standard heuristics — interfaces carry excessive visual clutter and fail to prioritise operator needs [Pham 2023]. Web-accessible SCADA implementations built on HTML5 demonstrate that browser-rendered displays can distribute monitoring functions across different hardware architectures without dedicated operator stations [Maccarana 2016], and multilayer web-based SCADA architectures applied in refinery environments confirm the feasibility of integrating distributed PLC systems with web-accessible supervisory functions across the full control hierarchy [Salihbegovic 2009]. The growing role of web-based visualisation also extends beyond conventional industrial automation, as augmented reality platforms increasingly employ browser-accessible and interactive visualisation technologies to support engineering model presentation and user interaction [Hrehova 2024, Zidek 2017].

The majority of deployed SCADA and HMI products still run as thick-client applications on fixed operator workstations. Both Maccarana et al. [Maccarana 2016] and Salihbegovic et al. [Salihbegovic 2009] frame web-based migration as a direct response to the access and maintenance constraints of station-bound deployments, while secured remote SCADA implementations explicitly position location-independent monitoring as a requirement that desktop-centric architectures cannot satisfy [Osman 2022]. Process engineers and maintenance teams need access to production data from multiple device types and from outside the plant network. Web-based monitoring architectures meet this requirement by separating the data service from the rendering client, so any browser-capable device displays live process values and trends without software installation [Maccarana 2016, Salihbegovic 2009, Osman 2022]. Prototype implementations across process industries demonstrate end-to-end feasibility, with field data routing from PLCs through OPC UA servers or MQTT brokers to web clients [Salem 2022, He 2025, Aghenta 2020]. Comparable web-enabled monitoring concepts have also been adopted in additive manufacturing, where PLC-controlled measurement systems and HMI applications automate inspection and quality verification of printed components, demonstrating the flexibility of browser-oriented supervisory solutions beyond traditional process industries [Duhancik 2024].

The demand for real-time monitoring is likewise increasing in additive manufacturing, where online inspection systems combine embedded sensing with artificial intelligence to identify printing defects during production and enable immediate process intervention [Kondrat 2025]. The consistent gap across published prototypes is a rigorous treatment of the OPC UA transport link between the PLC and the server-side application. Braune et al. [Braune 2008] establish that placing OPC UA secure communication inside a browser scripting engine carries non-trivial cryptographic overhead, which motivates hosting the OPC UA client on the server side. Ventuneac and Gaitan [Ventuneac 2024] implement and measure this pattern on a Linux-based gateway and report acquisition cycle performance under realistic monitoring conditions. Rocha et al. [Rocha 2018] provide protocol-level benchmarks comparing OPC UA and MQTT for industrial data exchange, informing the transport choice within PLC–server–web architectures.

This paper presents the design, implementation, and evaluation of a web-based IoT monitoring system for industrial process visualisation. The system places an OPC UA client on a server-side application that collects data from a PLC-hosted OPC UA server, then serves interactive process visualisations to any web

browser without client-side software installation. The contribution covers the complete monitoring architecture, from simulated PLC data generation and OPC UA communication to server-side data processing, historical storage, and browser-based visualization. The implemented prototype demonstrates the functional integration of these components through real-time process monitoring and interactive retrieval of historical data.

2 ANALYSIS OF PROCESS MONITORING WITH INTERACTIVE VISUALIZATION

A web application serves as the primary human-machine interface in this system, offering flexibility in visualizing running processes while maintaining responsiveness across device types and enabling remote access to the control panel.

Communication reliability rests on the OPC UA protocol, a widely adopted standard in Industrial Internet of Things (IIoT) environments that bridges the PLC layer with the data processing layer. The architecture follows a three-layer IoT model: a data-generation layer built on an OPC UA server, a collection and storage layer comprising an OPC UA client and JSON-based persistence, and a visualization layer delivered as a web application (Figure 1).

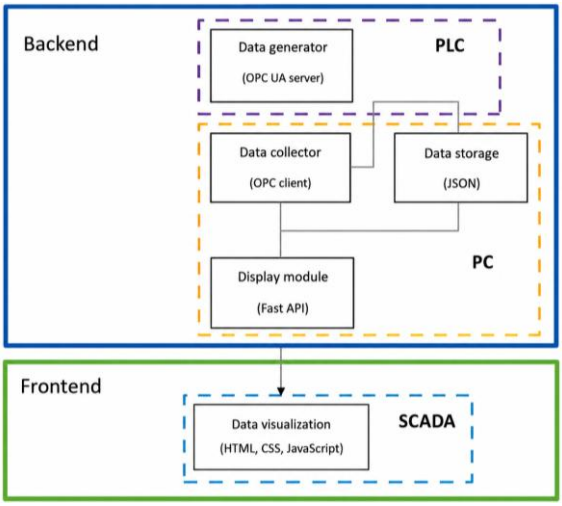


Figure 1. IoT Architecture Implementation for Monitoring a System

The role of the PLC in this architecture is to generate process data. To concentrate development effort on the software stack rather than physical hardware integration, the PLC was replaced by a simulated data-generation module running on a PC. This substitution allows the solution to function as a practical, hardware-independent prototype that can serve as a foundation for further development of monitoring and diagnostic systems in industrial environments. The monitored process models a pneumatic cylinder, with three principal variables tracked: the left chamber volume, the right chamber volume, and piston position (Figure 2).

The backend runs in Python, chosen for its readability and the maturity of its ecosystem. The following libraries compose the backend stack:

- *FastAPI* for the server and API layer;
- *Uvicorn* as the local ASGI server;
- *the OPC UA library* for client-server data transfer;
- *JSON* for data storage and retrieval;
- *threading* for concurrent task execution;
- *datetime* for timestamping;
- *pathlib* for filesystem operations;
- *time* for introducing controlled delays;
- *random* for generating simulated sensor values.

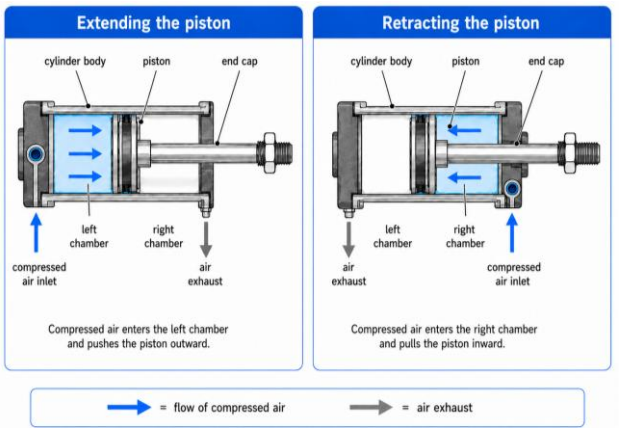


Figure 2. The modeled process of the Pneumatic Cylinder

The frontend interface combines HTML, CSS, and JavaScript. This stack requires no compilation step, runs in any modern browser, and gives developers direct control over the visual representation of process data.

3 DATA GENERATION, PROCESSING, AND STORAGE AS PART OF THE BACKEND

The backend fulfills three responsibilities: simulating the monitored process, exposing live data over OPC UA, and persisting that data for the visualization layer. The following two sections describe each component in turn.

3.1 *Process Model of the Pneumatic Cylinder (PLC Simulation)*

To generate realistic process data without physical hardware, a software model of the pneumatic cylinder was implemented as a Python class named *Piston* (Figure 3, blue block). The class holds three parameters: *total_volume*, which sets the fixed maximum air volume of the cylinder; *left_air_volume*; and *right_air_volume*, representing the current air distribution between the two chambers (Figure 3, green block).



Figure 3. Piston Class: process model with volume initialization, the update method, and the output formatter

The core behavioral method is *change_air_volume(change)* (Figure 3, orange block). The argument *change* specifies how much air volume shifts between chambers in a single step. Adding air to the right chamber reduces the left chamber by the same amount, keeping the total constant. The *min()* and *max()* functions clamp both values so neither chamber exceeds *total_volume* or drops below zero. The *__str__()* method (Figure 3, blue block) formats both chamber volumes for console output and logging.

During runtime, a random number generator feeds a stream of incremental changes into *change_air_volume()*, producing a continuous simulation of piston motion. The generator runs inside an infinite loop and stops only when the user sends a keyboard interrupt.

3.2 OPC UA Server for Local Data Transmission (PLC Simulation)

The OPC UA server acts as the digital counterpart of a physical PLC, publishing live cylinder data to a local endpoint that the collection layer reads in real time (Figure 4).

During the server initialization (Figure 4, purple and orange blocks), the server registers the namespace URI, obtains its index *idx*, and creates a node object named *PistonSimulator*. Two child variables, *left_air_volume* and *right_air_volume*, are attached to this node and marked writable so the simulation loop can update them during operation. At startup (Figure 4, green block), the server is started using *server.start()*, making the OPC UA endpoint available for client connections.

The main data loop (Figure 4, blue block) runs as a *while True* cycle. Each iteration calls *generate_change()* to produce a random volume delta, passes it to *piston.change_air_volume()*, rounds both resulting chamber volumes to three decimal places, and writes them to the OPC UA node variables using *set_value()*. A one-second *sleep()* call paces the updates, and a tick counter tracks the number of completed cycles. The current piston state is printed to the console at each step. On a *KeyboardInterrupt* (Figure 4, red block), the loop exits cleanly, and *server.stop()* shuts down the OPC UA server before the process terminates.

```
if __name__ == "__main__":
    piston = Piston()
    server = Server()
    server.set_endpoint("opc.tcp://localhost:4840/freeopcua/server/")

    uri = "http://example.org"
    idx = server.register_namespace(uri)

    piston_obj = server.nodes.objects.add_object(idx, "PistonSimulator")
    left_air_volume = piston_obj.add_variable(idx, "Left_air_volume", piston.left_air_volume)
    right_air_volume = piston_obj.add_variable(idx, "Right_air_volume", piston.right_air_volume)
    left_air_volume.set_writable()
    right_air_volume.set_writable()

    server.start()
    print(f"OPC UA Server started at {server.endpoint}")

    try:
        tick = 0
        while True:
            piston.change_air_volume(generate_change())
            left_air_volume.set_value(round(piston.left_air_volume, 3))
            right_air_volume.set_value(round(piston.right_air_volume, 3))
            print(piston, f"tick {tick}")
            tick += 1
            sleep(1)
    except KeyboardInterrupt:
        print("\nStopping OPC UA server...")
    finally:
        server.stop()
```

Figure 4. OPC UA Server: namespace registration and node setup, the live data update loop, and graceful shutdown on interrupt

3.3 Data Collection, Processing, and Storage (Server)

The system begins by establishing a communication channel between the backend and a simulated programmable logic controller (PLC). As shown in Figure 5, the OPC UA Server/PLC acts as the data source, and the OPC UA Client connects to it by targeting the server's local endpoint address. Once the connection is open, the client traverses the server's object hierarchy to locate the *PistonSimulator* node, which serves as the logical container for all piston-related process variables.

From the *PistonSimulator* object, the client retrieves two child nodes: *Right_air_volume* and *Left_air_volume*. These represent the instantaneous air volumes in the right and left chambers of the piston, respectively. Both values are read continuously using the client's value-reading interface and passed to a position calculation routine. The resulting piston position, expressed as a percentage, is combined with the raw chamber volumes and written into a shared system state object. This state variable acts as the single source of truth for the rest of the application — API endpoints and the history recording mechanism both draw from it. A short polling interval (0.2 s) governs how frequently this read–compute–update cycle repeats, ensuring that the state remains current without overloading the server connection.

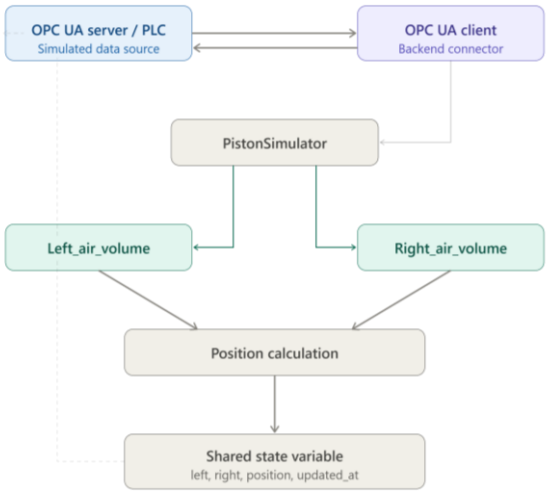


Figure 5. Connection to a simulated PLC via OPC UA

Once the shared state has been updated (Figure 6, top-left block), the history recording subsystem takes a snapshot of the current values. This snapshot captures the left and right chamber volumes, the computed piston position, and a timestamp, forming one discrete record of the system's state at that moment.

Before the record is stored, it passes through a normalisation step. The normalisation routine validates the record's structure and converts all fields into a uniform, well-typed format. If the record is malformed or any required field is missing, the routine discards it immediately and no storage takes place.

A successfully normalised record is then committed to two destinations simultaneously. First, it is appended to an in-memory list (*history*) that is protected by a mutual-exclusion lock to prevent concurrent write conflicts. The list is bounded: once its size exceeds a configurable limit, the oldest entries are pruned so that memory consumption remains controlled. Second, the record is serialised to JSON and appended as a new line to the persistent *piston_history.jsonl* file on disk, using UTF-8 encoding. This dual-write strategy means that recent records remain instantly accessible in RAM for low-latency API responses, while the full historical dataset accumulates on disk and remains available even after a process restart.

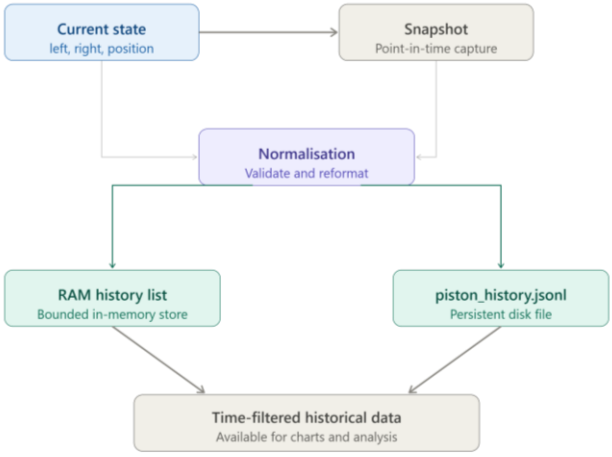


Figure 6. Storage of process data

Both storage paths feed a downstream filtering mechanism. When a user queries historical data for a specific time window, the application iterates over the stored records and retains only those whose timestamps fall within the requested interval. The filtered subset is then returned to the web interface, where it is rendered as time-series charts.

Figure 7 illustrates the two concurrent loops that together form the runtime architecture of the backend.
The primary loop begins at startup. The backend opens a connection to the OPC UA server and, provided the connection succeeds, enters a continuous read cycle. In each iteration, it reads the current air volumes from the left and right chamber nodes, calculates the piston position from those values, and writes the result back to the shared state. After each update, the loop waits 0.2 s before repeating. If the connection is lost or an exception occurs at any point, the error is logged and the OPC UA client is cleanly disconnected before any potential reconnection attempt. This loop runs indefinitely, ensuring that the shared state always reflects the most recent sensor readings.

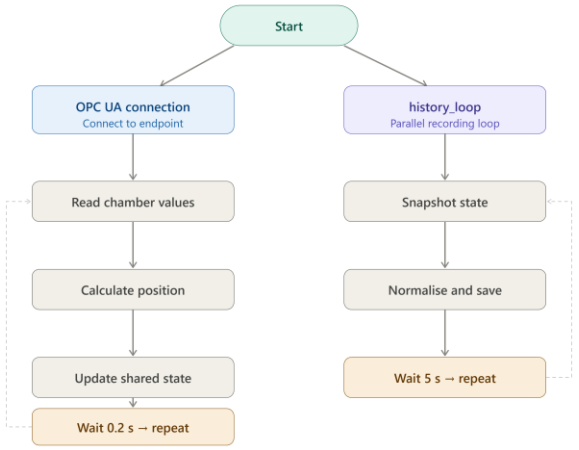


Figure 7. Program execution cycle

In parallel, a secondary loop (*history_loop*) operates independently of the OPC UA reading cycle. At a fixed interval of 5 s, it takes a snapshot of the current shared state, normalises the record, and appends it to both the in-memory history list and the JSONL file on disk, as described for Figure 6. After writing, it waits for the next 5 s interval and repeats. Because the two loops run concurrently and both access the shared state, mutex locks guard every read and write operation on that object to prevent race conditions.
Together, the two loops create a layered temporal structure: the primary loop maintains a near-real-time view of the piston (updated every 0.2 s), while the history loop captures a lower-frequency record suitable for trend analysis and charting (sampled every 5 s).

4 IMPLEMENTATION OF PROCESS DATA VISUALIZATION (FRONTEND)

Figure 8 describes the complete request-response cycle between the user's browser and the backend server, showing how two distinct views share a common data-fetching mechanism while drawing from separate API endpoints.
At the top of the diagram sits the user's browser, which is the origin of all data requests. The browser renders one of two HTML pages depending on the current view. The first, *index.html*, displays the real-time piston state — chamber volumes, computed position, and a dynamic graphical representation of the piston assembly. The second, *charts.html*, provides access to historical trends and time-series charts based on previously recorded data. Although the two pages serve different purposes, they share the same client-side communication strategy.
Both pages delegate all server communication to a JavaScript *fetch()* layer. This layer runs on a short polling interval, periodically issuing HTTP GET requests without requiring the user to reload the page. The use of *fetch()* as a shared abstraction means that neither page manages raw HTTP connections directly; instead, both rely on the same mechanism to retrieve JSON responses and update the DOM accordingly.
From this shared layer, the two pages diverge into separate API endpoints. The *index.html* page issues requests to *GET /piston*, which returns the current sensor values — left and right chamber air volumes, piston position, and the timestamp of the last update. Upon receiving the response, the JavaScript routine parses the JSON payload and writes each value into the corresponding DOM element, simultaneously updating the numeric readouts and repositioning the graphical piston to reflect the current state. The *charts.html* page instead queries *GET /history*, which accepts optional time-range parameters and returns the filtered historical dataset needed to populate the trend charts.

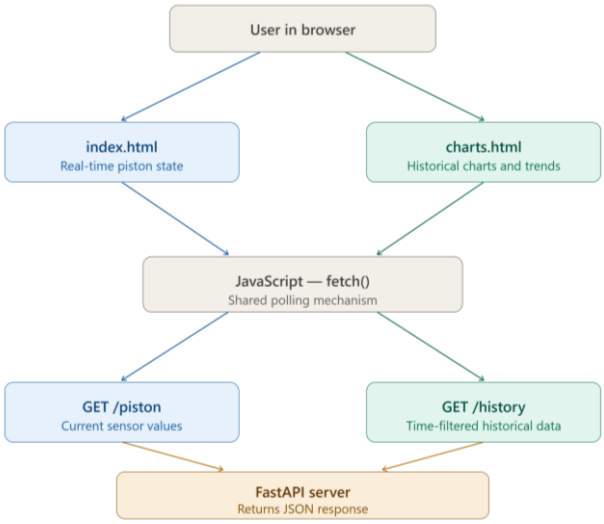


Figure 8. Frontend data flow

Both endpoints are served by the FastAPI backend, which handles incoming requests, retrieves the appropriate data from the shared state or the history store, serialises the result as a JSON response, and returns it to the browser. The backend is the single authoritative source for all data; the frontend never holds or modifies state independently.

5 RESULTS

Figure 9 shows the main monitoring view of the developed web-based HMI, which provides real-time visualization of the pneumatic piston system. The interface displays the current air volumes in the left and right chambers alongside the calculated piston position, all updated at a configurable storage interval.



Figure 9. The main monitoring view of the real-time visualization of the pneumatic piston

The central element is an animated piston diagram scaled as a percentage bar from 0 % to 100 %, which gives the operator an immediate spatial sense of how far the piston rod has

extended. Below the diagram, three data blocks present the left chamber volume, the right chamber volume, and the percentage fill level of the left chamber, each with its physical unit. A status line at the bottom of the card shows the active storage interval and a timestamp for the most recent data point, allowing the operator to verify that incoming measurements are current. Navigation to the historical data view is available from this screen without interrupting the live display.

The transition from real-time monitoring to historical analysis is handled by the chart page shown in Figure 10. The interface provides a *Quick Period selector* with predefined intervals ranging from the last minute up to the full recorded history, so the operator can load a representative data window in a single interaction. After selecting a period and confirming with the *Show control*, the system queries the stored records and renders three separate time-series charts: left chamber volume in litres, right chamber volume in litres, and piston position in percent. Each chart occupies its own panel, so the operator can compare chamber behaviour and the resulting piston displacement without the traces interfering with one another. A summary line above the charts reports *the number of loaded data points*, *the logging interval*, and *the timestamp of the most recent stored record*, giving the operator a quick sanity check before interpreting the trends. The example presented in Figure 10 demonstrates the retrieval of historical process data using the predefined "Last 7 Days" query. The selected interval defines the time period for data retrieval, within which 872 data points representing several cycles of chamber volume change was stored. These records are displayed as time-series charts of the left and right chamber air volumes and the calculated piston position. This example demonstrates the system's ability to retrieve and visualize historical process data within a specified time interval.



Figure 10. A chart page displaying a visualization of historical data on the movement of a pneumatic piston

Figure 11 demonstrates the Custom Range mode of the same chart page. The operator sets explicit *start* and *end timestamps* using date-time input fields, then confirms the selection to fetch only the records within that span. Four shortcut buttons (*Today, Yesterday, Last 30 days, Last hour*) reduce manual entry for frequently used intervals. Once the query returns, the same three charts render the data for the chosen period, and the information line updates to reflect the exact time boundaries and the point count.

The dense, high-amplitude oscillations visible across all three charts during the selected custom range confirm that the system captures rapid chamber volume changes at the configured logging rate and presents them without downsampling visible at the displayed resolution. This mode is designed for post-event analysis, where the operator needs to examine a specific incident rather than a rolling window. Together, the two query modes cover the full range of operational needs: routine trend review through *Quick Period* and targeted fault investigation through *Custom Range*.



Figure 11. Visualization of historical data on the movement of a pneumatic piston using the "Custom Range" mode

6 CONCLUSIONS

A virtual model of a pneumatic piston was developed to simulate air volume values in the left and right chambers, from which the piston position is derived. A virtual OPC UA server then acted as the process data source. The server-side application collected this data, stored it in JSON format, and exposed it through an API.

The resulting web application displays the current state of the simulated process in real time. Users can view air volume values for each chamber, the current piston position, and the timestamp of the last data update. A separate historical data page allows selection of predefined time intervals or a custom date and time range, with values rendered as time-series graphs.

The project objective was met. The solution demonstrates that a functional monitoring system prototype can be built without physical PLC hardware or a real pneumatic device, covering industrial communication, data processing, and visualization in a single integrated pipeline. The system provides a foundation for further development: connection to a real PLC, replacement of file-based storage with a database, addition of alarms, user authentication, or extension of the visualization to further process variables.

The main contribution of this work is the practical validation of the integration between a simulation model, the OPC UA communication protocol, a backend application, and a web-based visualization layer. The result is a modular, extensible process monitoring system consistent with current IoT and industrial monitoring practices.

ACKNOWLEDGMENTS

This research was funded by the projects VEGA 1/0559/26, KEGA 014TUKE-4/2026 granted by the Ministry of Education, Research, Development and Youth of the Slovak Republic and supported by the projects APVV-19-0590 and ITMS21+ code: 401101C504.

REFERENCES

[Aghenta 2020] Aghenta, L.O. and Iqbal, M.T. Design and Implementation of a Low-Cost, Open Source IoT-Based SCADA System Using ESP32 with OLED, ThingsBoard and MQTT Protocol. AIMS Electronics and Electrical Engineering, 2020, Vol. 4, No. 1, pp. 57-86. ISSN 2578-1588.

[Alves 2023] Alves, J., Lima, T.M., Gaspar, P.D. Is Industry 5.0 a Human-Centred Approach? A Systematic Review. Processes, 2023, Vol. 11, No. 1, p. 193.

[Braune 2008] Braune, A., Hennig, S., Hegler, S. Evaluation of OPC UA Secure Communication in Web Browser

Applications. In: Proc. of 6th IEEE Int. Conf. on Industrial Informatics (INDIN 2008), Daejeon, South Korea, 13-16 July 2008. Piscataway, NJ: IEEE, pp. 1660-1665. ISBN 978-1-4244-2176-3.

[Cavalieri 2013] Cavalieri, S. and Chiacchio, F. Analysis of OPC UA Performances. Computer Standards & Interfaces, 2013, Vol. 36, No. 1, pp. 165-177. ISSN 0920-5489.

[Duhancik 2024] Duhancik, M., et al. The Automated Quality Control of 3D Printing using Technical SMART Device. In: Proc. of 2024 25th Int. Carpathian Control Conference (ICCC), Velke Karlovice, Czech Republic, 27-29 May 2024. Piscataway, NJ: IEEE, pp. 1-6. ISBN 979-8-3503-5070-8.

[Ghobakhloo 2023] Ghobakhloo, M., et al. Behind the Definition of Industry 5.0: A Systematic Review of Technologies, Principles, Components, and Values. Journal of Industrial and Production Engineering, 2023, Vol. 40, No. 6, pp. 432-447. ISSN 2168-1015.

[He 2025] He, W., et al. An Internet of Things Supervisory Control and Data Acquisition (IoT-SCADA) Architecture for Photovoltaic System Monitoring, Control, and Inspection in Real Time. Electronics, 2025, Vol. 14, No. 1, Article 42. ISSN 2079-9292.

[Hrehova 2024] Hrehova, S., et al. Characteristics of Selected Software Tools for the Design of Augmented Reality Models. In: Proc. of 2024 25th Int. Carpathian Control Conference (ICCC), Velke Karlovice, Czech Republic, 27-29 May 2024. Piscataway, NJ: IEEE, pp. 117-122. ISBN 979-8-3503-5070-8.

[Kant 2025] Kant, V., et al. Supporting Cognitive Intelligence for Smart Manufacturing Systems Using HMI Design: Challenges and Fundamental Issues. Journal of Engineering Design, 2025, Vol. 36, No. 11, pp. 1961-1981. ISSN 0954-4828.

[Kondrat 2025] Kondrat, M., et al. Real-Time Monitoring of 3D Printing Process by Endoscopic Vision System Integrated in Printer Head. Applied Sciences, 2025, Vol. 15, No. 17, Article 9286. ISSN 2076-3417.

[Kondrat 2025] Kondrat, M., et al. Experimental System for Measuring the Bending Stiffness of Soft Actuators Based on Servo Drives. MM Science Journal, October 2025, pp. 8686-8691. ISSN 1803-1269.

[Ladegourdie 2022] Ladegourdie, M. and Kua, J. Performance Analysis of OPC UA for Industrial Interoperability towards Industry 4.0. IoT, 2022, Vol. 3, No. 4, pp. 507-525. ISSN 2624-831X.

[Lee 2021] Lee, C., et al. Toward Industrial IoT: Integrated Architecture of an OPC UA Synergy Platform. IEEE Access, 2021, Vol. 9, pp. 164720-164731.

[Maccarana 2016] Maccarana, Y., et al. Design of a HTML5 SCADA System. In: Proc. of 2016 IEEE 2nd Int. Forum on Research and Technologies for Society and Industry (RTSI), Bologna, Italy, 7-9 September 2016. Piscataway, NJ: IEEE, pp. 1-6. ISBN 978-1-5090-1131-5.

[Osman 2022] Osman, F.A., et al. Secured Cloud SCADA System Implementation for Industrial Applications. Multimedia Tools and Applications, 2022, Vol. 81, No. 7, pp. 9989-10005. ISSN 1380-7501.

[Pham 2023] Pham, T., et al. Heuristic Evaluation of Supervisory Control and Data Acquisition (SCADA) Displays. Proceedings of the Human Factors and Ergonomics Society Annual Meeting, 2023, Vol. 67, No. 1, pp. 1767-1774. ISSN 2169-5067.

[Rocha 2018] Rocha, M. S., et al. Performance Comparison Between OPC UA and MQTT for Data Exchange. In: Proc. of 2018 IEEE Int. Workshop on Metrology for Industry 4.0 and IoT (MetroInd 4.0 & IoT), Brescia, Italy, 16-18 April 2018. Piscataway, NJ: IEEE, pp. 175-179. ISBN 978-1-5386-2497-5.

[Salem 2022] Salem, R.M.M., et al. An Industrial Cloud-Based IoT System for Real-Time Monitoring and Controlling of Wastewater. IEEE Access, 2022, Vol. 10, pp. 6528-6540. ISSN 2169-3536.

[Salihbegovic 2009] Salihbegovic, A., et al. Web Based Multilayered Distributed SCADA/HMI System in Refinery Application. Computer Standards & Interfaces, 2009, Vol. 31, No. 3, pp. 599-612.

[Trifonov 2023] Trifonov, H. and Heffernan, D. OPC UA TSN: A Next-Generation Network for Industry 4.0 and IIoT. International Journal of Pervasive Computing and Communications, 2023, Vol. 19, No. 3, pp. 386-411. ISSN 1742-7371.

[Ventuneac 2024] Ventuneac, C. and Gaitan, V.G. Industrial Internet of Things Gateway with OPC UA Based on Sitara AM335X with ModbusE Acquisition Cycle Performance Analysis. Sensors, 2024, Vol. 24, No. 7, Art. No. 2072. ISSN 1424-8220.

[Yang 2024] Yang, J., Liu, Y., Morgan, P.L. Human-machine Interaction towards Industry 5.0: Human-centric Smart Manufacturing. Digital Engineering, 2024, Vol. 2, Article 100013. ISSN 2950-550X.

[Zidek 2017] Zidek, K., et al. Machine Learning Algorithms Implementation into Embedded Systems with Web Application User Interface. In: Proc. of IEEE 21st Int. Conf. on Intelligent Engineering Systems 2017 (INES), Larnaca, Cyprus, 20-23 October 2017. Piscataway, NJ: IEEE, pp. 77-81. ISBN 978-1-4799-7677-5.

CONTACTS:

Lina Sulska, Ing.
Artem Koliesnik, Bc.
Michal Duhancik, Ing. PhD.
Jan Pitel, Prof. Ing. PhD.
Marek Kocisko, Prof. Ing. PhD.
Technical University of Kosice
Faculty of Manufacturing Technologies with the seat in Presov
Bayerova 1, 080 01 Presov, Slovakia
lina.sulska@tuke.sk, artem.koliesnik@student.tuke.sk, michal.duhancik@tuke.sk, jan.pitel@tuke.sk, marek.kocisko@tuke.sk

LICENSE CREATIVE COMMONS:

The article is published under the terms and conditions of the Creative Commons Attribution 4.0 International License (CC BY 4.0).