Factory Monitoring System: Architecture and Implementation Strategy

1. Introduction

This report details the architecture and implementation strategy for a comprehensive factory monitoring system. The system leverages ESP32-based sensor nodes capable of dual network connectivity (Wi-Fi and W5500 Ethernet) to collect environmental data (temperature, voltage). Data transmission relies on the MQTT protocol to a central Mosquitto broker. For localized inter-node communication or data relaying, the ESP-NOW protocol is explored. A robust data pipeline, potentially implemented using Node-RED or custom scripts (Python/Node.js), processes incoming MQTT messages and stores them in a MongoDB database, specifically utilizing Time Series collections for optimal handling of sensor readings. A web-based dashboard, built with Nuxt.js, provides visualization and interaction capabilities. This includes displaying a factory layout derived from an AutoCAD PDF export (rendered as SVG), overlaying interactive elements, presenting real-time data via MQTT over WebSockets, and allowing users to query historical data through backend API endpoints built with Nuxt 3's Nitro server engine. The report addresses key implementation details for each component and analyzes architectural considerations regarding reliability, scalability, and security within a demanding factory environment.

2. ESP32 Sensor Node Development

The foundation of the monitoring system lies in the ESP32 sensor nodes deployed throughout the factory. These nodes are responsible for acquiring sensor data, maintaining network connectivity through potentially redundant paths, and publishing the collected information via MQTT.

2.1. Sensor Data Acquisition (Temperature & Voltage)

The primary function of the ESP32 nodes is to read data from connected sensors. For this system, temperature and voltage measurements are required.

- **Temperature Sensing (DS18B20 Example):** The DS18B20 is a common choice for digital temperature sensing due to its 1-Wire interface, requiring only one data pin on the ESP32.[1] It operates within a suitable voltage range (3.0V-5.5V) and temperature range (-55°C to +125°C) for many industrial applications.[1] Each sensor possesses a unique 64-bit address, enabling multiple sensors to be connected to the same GPIO pin.[1]
    - **Wiring:** Connect the DS18B20 data pin to a designated ESP32 GPIO (e.g.,

GPIO 4), VDD to 3.3V/5V, and GND to GND. A 4.7kΩ pull-up resistor is required between the data line and VDD.[1]
    - **Libraries (Arduino IDE):** Integration requires the OneWire library (by Paul Stoffregen) and the DallasTemperature library (by Miles Burton).[1] These can be installed via the Arduino IDE Library Manager.
    - **Code Logic:**
        1. Include necessary libraries: #include <OneWire.h> and #include <DallasTemperature.h>.
        2. Instantiate library objects: OneWire oneWire(ONE_WIRE_BUS); and DallasTemperature sensors(&oneWire);, where ONE_WIRE_BUS is the GPIO pin number.
        3. Initialize sensors in setup(): sensors.begin();.
        4. Request readings in loop(): sensors.requestTemperatures();.
        5. Retrieve temperature: float tempC = sensors.getTempCByIndex(0); for the first sensor on the bus.[1] Use getTempFByIndex() for Fahrenheit. Handle potential read errors (e.g., DEVICE_DISCONNECTED_C [2]). For multiple sensors, iterate using indices (0, 1, 2...).
- **Voltage Sensing:** Specific implementation depends on the voltage range and type (AC/DC). Common methods include:
    - **Voltage Dividers:** For DC voltages higher than the ESP32's ADC input limit (typically 3.3V), a resistor voltage divider scales the voltage down to a measurable range. Careful resistor selection is needed for accuracy.
    - **Voltage Sensor Modules:** Dedicated modules (e.g., ZMPT101B for AC, simple resistive dividers for DC) simplify interfacing and provide isolation if needed.
    - **ADC Reading:** Use the ESP32's built-in Analog-to-Digital Converter (ADC) via analogRead(PIN_NUMBER);. Note that ESP32 ADCs can have non-linearities; calibration or using lookup tables might be necessary for high accuracy. Refer to ESP32 ADC documentation for specific usage and characteristics.

2.2. Dual Network Connectivity (WiFi & W5500 Ethernet)

To enhance reliability in a potentially challenging factory network environment, nodes will support both built-in Wi-Fi and external W5500 Ethernet modules, with logic to manage the connection and failover between them.

- **Wi-Fi Implementation:**
    - **Library:** The standard WiFi.h library, included with the ESP32 Arduino core, provides the necessary functions.[3]
    - **Connection Logic:**
        1. Set the mode to station: WiFi.mode(WIFI_STA);.[3]

2. Initiate connection: WiFi.begin(ssid, password);.[3]
3. Monitor connection status in a loop: while (WiFi.status()!= WL_CONNECTED) { delay(500); Serial.print("."); }.[3] Referencing connection status codes (e.g., WL_CONNECTED, WL_CONNECT_FAILED) provides detailed feedback.[3]
4. Retrieve network information: Use WiFi.localIP() for the assigned IP address and WiFi.RSSI() for signal strength.[3]
5. Advanced Handling: The ESP32 Wi-Fi event system (WiFiEvent) can be used for more sophisticated, non-blocking connection management.[3]

- **W5500 Ethernet Implementation:**
  - **Hardware Interface:** The W5500 module typically connects to the ESP32 via the SPI bus (MOSI, MISO, SCK pins), along with Chip Select (CS), Interrupt (INT), and Reset (RST) pins. Standard SPI pin assignments should be used unless configured otherwise.
  - **Library Selection:** Choosing the correct library is crucial and depends on the Arduino core version and potential conflicts.
    - The standard Arduino Ethernet.h library may require adaptation for the ESP32.
    - The EthernetESP32 library is suggested for Arduino Core v3.0 and later.[5]
    - Libraries like AsyncWebServer_ESP32_W5500 [5] or its SparkFun remix SparkFun_WebServer_ESP32_W5500 [6] exist but are primarily web server libraries and, critically, may conflict with Arduino core SPI and interrupt handling.[6] This conflict arises because these libraries might use lower-level ESP-IDF SPI/interrupt methods that clash with the Arduino core's abstractions. Careful code structuring and testing are essential if using these. The end() method should be called before restarting if switching between these libraries and standard Arduino SPI/Ethernet usage.[6]
    - For recent ESP32 Arduino cores (v2.0+ based on ESP-IDF), the built-in ETH.h library with ETH_PHY_W5500 configuration appears to be the most integrated approach.[5]
  - **Configuration:** Network parameters can be set statically (ETH.config(myIP, myGW, mySN, myDNS);) or obtained via DHCP using ETH.begin(ETH_PHY_TYPE, ETH_PHY_ADDR,...);.[5] Static configuration requires defining IPAddress objects for IP, Gateway, Subnet Mask, and optionally DNS.[5]
- **Dual Connectivity Logic (Selection & Failover):**
  - The standard WiFiMulti library is designed for managing connections to multiple Wi-Fi access points based on signal strength and availability [7], not for handling heterogeneous interfaces like Wi-Fi and Ethernet. Therefore, custom

logic is required.

- A state machine approach provides a structured way to manage the connections:
    1. **States:** Define states like STATE_INIT, STATE_CONNECTING_ETH, STATE_ETH_CONNECTED, STATE_CONNECTING_WIFI, STATE_WIFI_CONNECTED, STATE_BOTH_FAILED.
    2. **Preference:** Prioritize Ethernet connectivity due to its generally higher stability in industrial environments.
    3. **Startup (setup()):** Transition to STATE_CONNECTING_ETH.
    4. **Main Loop (loop()):**
        - In STATE_CONNECTING_ETH: Attempt Ethernet connection (ETH.begin()). Check status (e.g., ETH.linkStatus(), ETH.localIP()). On success, move to STATE_ETH_CONNECTED. After a timeout or failure, move to STATE_CONNECTING_WIFI.
        - In STATE_ETH_CONNECTED: Periodically check link status (ETH.linkStatus()) and application-level connectivity (e.g., MQTT connection status via client.connected()). If connection lost, move to STATE_CONNECTING_WIFI.
        - In STATE_CONNECTING_WIFI: Attempt Wi-Fi connection (WiFi.begin()). Check status (WiFi.status()). On success, move to STATE_WIFI_CONNECTED. After a timeout or failure, move to STATE_BOTH_FAILED.
        - In STATE_WIFI_CONNECTED: Periodically check link status (WiFi.status()) and application-level connectivity. If connection lost, move to STATE_BOTH_FAILED. Optionally, periodically attempt to reconnect to Ethernet in the background and switch back if successful, moving to STATE_CONNECTING_ETH.
        - In STATE_BOTH_FAILED: Implement a retry mechanism, possibly with exponential backoff, attempting Ethernet first (STATE_CONNECTING_ETH), then Wi-Fi (STATE_CONNECTING_WIFI).
    5. **Non-Blocking Checks:** Ensure connection checks within the loop are non-blocking or have short timeouts to avoid delaying sensor readings and MQTT publishing. Use timers (millis()) for periodic checks and timeouts.

2.3. MQTT Data Publishing

Once network connectivity is established, the node must publish the sensor data to the central Mosquitto MQTT broker.

- **Development Environment and Protocol Choice:**
  - **Arduino C++ (PubSubClient):** This is a widely adopted library for MQTT in the Arduino ecosystem, with numerous examples available.[4] It integrates well with other Arduino libraries like WiFi.h or Ethernet libraries. While some sources mention potential past incompatibilities or reliability concerns with ESP32 [9], many successful implementations exist. If issues arise, investigating the underlying ESP-IDF esp-mqtt client library might be necessary, though this adds complexity.[15] Given the focus on hardware integration (sensors, W5500) and the potential need for debugging low-level issues like SPI conflicts [6], the C++ environment offers more direct control.
  - **MicroPython (umqttsimple):** Offers a simpler, Pythonic syntax [16] and an interactive REPL for rapid development and testing.[17] Requires flashing MicroPython firmware onto the ESP32 and managing libraries using tools like upip.[17] The umqttsimple library provides basic MQTT client functionality.[18] While viable, debugging hardware-specific issues (like the W5500 SPI conflict) might be more challenging than in C++.
  - **Recommendation:** For this project, prioritizing robust hardware integration and leveraging the extensive Arduino library ecosystem, **Arduino C++ with the PubSubClient library** is the recommended starting point. MicroPython remains an alternative if development speed or Python familiarity is paramount, but requires careful validation of the W5500 Ethernet integration.
- **Implementation (PubSubClient Example):**
  - **Include Libraries:** Add #include <PubSubClient.h> alongside the active network library (WiFi.h or the chosen Ethernet library).[9]
  - **Client Initialization:** Instantiate the network client (WiFiClient or EthernetClient) and the MQTT client: WiFiClient espClient; PubSubClient client(espClient);.[9] The network client instance used depends on the active connection (Ethernet or Wi-Fi).
  - **Broker Connection:**
    - Set the broker address and port: client.setServer(mqtt_server_ip, 1883); (adjust port for TLS if used).[9]
    - Implement a reconnect() function to handle connection attempts and retries. This function should loop until client.connected() is true.[4] Inside the loop, attempt connection using client.connect("ESP32_Node_01", mqtt_user, mqtt_password);. Use a unique client ID for each node.[4] Implement delays between retries.[9]
    - Call client.loop() frequently within the main loop() function. This is essential for processing incoming messages (if subscribed) and maintaining the connection.[4]

- **Publishing:**
    1. Read sensor values (temperature, voltage).
    2. Format the payload. JSON is recommended for easy parsing downstream: {"value": 25.7, "unit": "C"}. Use functions like dtostrf() to convert float values to strings for inclusion in the payload [9], or use libraries like ArduinoJson to build the JSON string.
    3. Publish the data to specific topics using client.publish(topic, payload);. Define clear, hierarchical topics, e.g., factory/nodes/{nodeId}/temperature and factory/nodes/{nodeId}/voltage, replacing {nodeId} with the unique identifier of the ESP32 node.[9]
    4. Implement a timer (millis()) to publish data at regular intervals (e.g., every 5 or 10 seconds).[9]

3. ESP-NOW for Inter-Node Communication

ESP-NOW offers a mechanism for direct, low-latency communication between ESP32 nodes without requiring a Wi-Fi access point, suitable for specific localized interactions within the factory.

3.1. Protocol Overview & Use Cases

- **Definition:** ESP-NOW is a connectionless, 2.4GHz wireless protocol developed by Espressif. It enables fast transmission of small data packets (up to 250 bytes) directly between ESP devices.[21] It can operate concurrently with standard Wi-Fi connections.[23]
- **Characteristics:** Key features include low latency, reduced power consumption compared to a full Wi-Fi connection, and support for one-way or two-way communication.[21]
- **Potential Factory Use Cases:**
    - **Local Coordination:** Enabling rapid interaction between nearby devices, such as one sensor triggering an immediate action on a nearby actuator, bypassing the latency of the MQTT broker and cloud infrastructure.
    - **Data Relaying:** Nodes located in areas with poor Wi-Fi or Ethernet coverage could relay their sensor data via ESP-NOW to a nearby "gateway" node that has reliable network access.
    - **Low-Power Nodes:** Battery-operated sensors could wake periodically, transmit readings via ESP-NOW to a mains-powered gateway node, and then return to deep sleep, conserving energy.[24]

3.2. Pairing Mechanisms

Establishing communication requires the sending device to know the MAC address of the receiving device.[21] Each ESP32 possesses a unique MAC address burned in at the factory.[21]

- **Manual Pairing:**
  - **Method:** The MAC address of the receiver node is explicitly hardcoded into the sender node's firmware. The sender uses esp_now_add_peer() with the receiver's MAC address to register it before sending messages.[21] Example structures are shown in.[28]
  - **Pros:** Simple, highly reliable for fixed network topologies where node addresses are known beforehand.
  - **Cons:** Inflexible. Replacing a node requires updating the firmware of all nodes that communicate with it. Not suitable for dynamic environments where nodes might be added or removed frequently.
- **Auto-Pairing (using Broadcast):**
  - **Mechanism:** This approach allows nodes to discover each other dynamically.
    1. A new node (sender/slave) broadcasts a pairing request message using the special broadcast MAC address FF:FF:FF:FF:FF:FF.[26] All ESP-NOW devices on the same Wi-Fi channel receive this.
    2. A designated server/hub node receives the broadcast. The onDataRecv callback provides the sender's MAC address.[21]
    3. The server adds the sender as a peer using esp_now_add_peer() with the received MAC address.[28]
    4. The server sends a confirmation message directly back to the sender's MAC address, potentially including its own MAC address and channel information.[28]
    5. The sender receives the confirmation, extracts the server's MAC address from the callback, and adds the server as a peer.[28]
  - **Considerations:** The server typically needs to operate in WIFI_AP_STA mode.[27] Senders might need to cycle through Wi-Fi channels (1-13) during the broadcast phase if the server's channel is unknown, adding delay.[27]
  - **Pros:** Dynamic pairing, no need to hardcode MAC addresses, facilitates easier node replacement or addition.
  - **Cons:** More complex implementation, initial pairing relies on broadcast reliability (potential for missed packets), channel scanning adds latency to initial connection.
- **Recommendation:** For initial deployment in a factory setting where node locations are likely planned, **manual pairing** offers greater simplicity and predictability. If the system requires dynamic addition or replacement of nodes

without firmware updates, **auto-pairing** can be implemented, but its robustness must be thoroughly validated through on-site testing, considering potential broadcast failures and channel discovery delays.[27]

3.3. Range, Reliability, and Limitations

- **Range:** ESP-NOW generally offers a longer range than direct peer-to-peer Wi-Fi connections. Outdoor tests suggest ranges potentially reaching 100-500 meters under ideal conditions [24], with reliable communication observed up to 220 meters in some tests.[22] Espressif documentation mentions a Long Range (LR) mode that can further extend this [29], potentially reaching 500m+.[29] However, **indoor performance, especially in factories, is heavily degraded** by obstacles (walls, machinery), reflections, and RF interference.[30] Practical indoor range might be significantly less than outdoor figures.[30] Compared to LoRa, ESP-NOW has a considerably shorter range.[24]
- **Data Rate:** Achievable rates depend on conditions but can be around 200-500 Kbps, significantly faster than LoRa but slower than full Wi-Fi.[31]
- **Packet Size:** Limited to a maximum payload of 250 bytes per transmission.[21] This is generally sufficient for sensor readings or simple commands but not for large data transfers.
- **Encryption:** AES-128 encryption is supported for secure communication. However, there's a limit on the number of encrypted peers a device can manage simultaneously. For ESP32, the limit is typically up to 17 (default 7) [31], while for ESP8266, it's 10 in Station mode or 6 in SoftAP/Station mode.[24] The total number of peers (encrypted + unencrypted) is limited to 20.[23]
- **Reliability:** ESP-NOW is a connectionless protocol, meaning delivery is not inherently guaranteed like TCP. However, the esp_now_register_send_cb() function provides a callback that indicates whether the transmission was acknowledged by the peer (ESP_NOW_SEND_SUCCESS or ESP_NOW_SEND_FAIL).[21] This provides a basic level of delivery confirmation. Application-level logic (retries, sequence numbers, acknowledgments within the payload) is necessary if higher reliability is required. The 2.4GHz band used by ESP-NOW is shared with Wi-Fi and Bluetooth, making it susceptible to interference from other wireless devices and industrial equipment (motors, microwaves), which can impact reliability in a factory setting.[30] Environmental factors are a major performance driver.[30]
- **Operational Considerations:** ESP-NOW communication requires devices to be on the same Wi-Fi channel. If a device connects to a Wi-Fi AP, its channel is fixed to that of the AP, which might affect communication with ESP-NOW peers on different channels.[23] Power-saving modes can also interfere with ESP-NOW

reception unless configured carefully.[23]

The use of 2.4GHz means ESP-NOW faces significant reliability challenges in a typical factory environment due to RF noise and physical obstructions. While offering advantages in power and latency over Wi-Fi for short hops [24], its range and robustness should not be overestimated in this context. The limits on encrypted peers [23] also need careful consideration based on the deployment scale and security needs. **Extensive on-site testing is mandatory** to determine the practical range and packet delivery ratio before relying on ESP-NOW for any critical functions.

3.4. Integration Strategy

To incorporate data from ESP-NOW nodes into the central monitoring system, a gateway approach is recommended.

- **Direct Node-to-Node:** Suitable for simple, localized control loops where data does not need to reach the central system (e.g., a proximity sensor directly triggering a local indicator light via ESP-NOW).
- **ESP-NOW to MQTT Gateway:** This is the most practical strategy for integrating sensor data from ESP-NOW-only nodes.
    - **Concept:** Designate one or more ESP32 nodes that have reliable primary network connectivity (Ethernet preferred, or stable Wi-Fi) to act as gateways.[24] These gateways listen for incoming ESP-NOW messages from other nodes.
    - **Implementation:**
        1. The gateway node initializes both its primary network connection (Ethernet/Wi-Fi) and MQTT client, as well as ESP-NOW.
        2. It registers an ESP-NOW receive callback function using esp_now_register_recv_cb(OnDataRecv);.[21]
        3. Inside the OnDataRecv callback function (const uint8_t * mac, const uint8_t *incomingData, int len), the gateway receives the data payload and the MAC address of the sending node.
        4. The gateway parses the incomingData. The payload format must be predefined and should include the actual sensor value(s) and potentially an identifier for the originating sensor/node if the MAC address alone isn't sufficient.
        5. The gateway formats this information into a JSON payload suitable for the central system.
        6. Using its active MQTT client connection, the gateway publishes this data to the appropriate MQTT topic, clearly indicating the originating node (e.g., factory/nodes/{original_node_mac}/temperature/via_gateway).

- **Considerations:** The gateway node becomes a critical point of failure for all nodes relying on it. Redundant gateways might be considered for critical areas. Ensure the gateway's ESP-NOW channel matches the sender nodes or implement channel scanning/agreement. The gateway must be able to handle the aggregated message rate from all its associated ESP-NOW nodes.

4. Data Pipeline: MQTT to MongoDB

This section details the process of transferring sensor data from the Mosquitto MQTT broker to the MongoDB database for storage and later retrieval.

4.1. MQTT Broker Setup (Mosquitto)

It is assumed that a Mosquitto MQTT broker instance is operational, accessible by both the ESP32 nodes and the data pipeline component. Key configuration aspects include:

- **Ports:** Standard ports are 1883 (TCP), 8883 (TLS/SSL), 8083 (WebSocket), and 8084 (Secure WebSocket/WSS). Ensure the necessary ports are open in any firewalls.
- **Authentication:** Implement username/password authentication for clients (ESP32 nodes, pipeline, dashboard) to prevent unauthorized access. Client certificate authentication offers higher security if required.
- **Persistence:** Configure broker persistence if message buffering is needed during pipeline downtime, although relying on client-side buffering or QoS levels might be preferable.
- **TLS/SSL:** Enable TLS encryption for all connections (MQTT and WebSocket) to protect data in transit, especially if data traverses untrusted networks.

4.2. Pipeline Implementation Options

Several approaches can be used to subscribe to MQTT topics and insert data into MongoDB. The choice depends on factors like development speed, required complexity, performance needs, and team expertise.

Feature	Node-RED	Custom Python Script (paho-mqtt + pymongo)	Custom Node.js Script (mqtt.js + mongodb/mongoose)	Pro Mosquitto Plugin

Pros	Visual dev, rapid prototyping [32], many nodes, easy MQTT/DB integration [33]	Full control, high efficiency potential, Python ecosystem [35], good for complex logic	Full control, async nature, Node.js ecosystem [37], potentially efficient	Tightly integrated [39], config-based, potentially efficient
Cons	Can become complex, runtime overhead, dependency management	Requires coding/testing, script execution management needed	Requires coding/testing, script execution management needed	Broker lock-in (Pro Mosquitto) [39], less flexible, potential cost
Key Libraries/Tools	node-red-contrib-mongodb4, MQTT In/Out nodes	paho-mqtt [35], pymongo [36]	mqtt.js [37], mongodb driver [38] or mongoose [42]	Pro Mosquitto MongoDB Bridge Plugin [39]
Development Effort	Low (initially)	Medium to High	Medium to High	Low to Medium (configuration)
Performance	Moderate	Potentially High (with optimization like batching)	Potentially High (with async/batching)	Potentially High
Use Case Suitability	PoC, simple transformations, low-medium volume	Production, complex logic, high volume, scalability	Production, complex logic, high volume, scalability	If already using Pro Mosquitto, simple mapping

- **Node-RED Implementation:** Utilizes MQTT input nodes to subscribe (e.g., to factory/nodes/#). A JSON node parses the payload if necessary.[33] Function or Change nodes transform the data structure (e.g., adding server-side timestamps, extracting values). Finally, a MongoDB output node (configured for insertOne or insertMany) writes the data to the specified collection. This is excellent for getting started quickly but can become difficult to manage for complex transformations or high-throughput scenarios.
- **Custom Python Script:** Employs the paho-mqtt library to connect and subscribe.[35] The on_message callback receives messages.[35] Inside the callback,

the payload is decoded and parsed (e.g., json.loads(message.payload.decode())). The pymongo library connects to MongoDB (ideally establishing a persistent connection outside the callback [44]). Data is structured according to the schema and inserted using collection.insert_one() or, more efficiently for higher volumes, batched using collection.insert_many().[36] Robust error handling for both MQTT and database operations is crucial. Requires managing the script as a service/daemon.

- **Custom Node.js Script:** Similar to Python, using mqtt.js for MQTT connection and subscription (client.on('message',...) [37]). The native mongodb driver [38] or mongoose ODM [42] handles database interaction. A persistent database connection should be established.[44] The message handler parses the payload (JSON.parse(message.toString())), structures data, and inserts using methods like collection.insertOne() [38] or Mongoose's save().[42] Node.js's asynchronous nature is well-suited for handling concurrent MQTT messages and database operations. Batching inserts is recommended for performance. Requires service management. The example in [38] demonstrates the concept but uses an inefficient pattern of opening/closing the DB connection for each message.
- **Broker-Specific Bridge:** Solutions like the Pro Mosquitto MongoDB Bridge Plugin [39] handle the transfer directly within the broker based on configuration files (mongodb-bridge.json [39]). This defines mappings between MQTT topics and MongoDB collections, potentially including basic schema transformations.[39] While potentially very efficient, it locks the solution to that specific broker version and might offer less flexibility than custom code.
- **Recommendation:**
    - **Node-RED** is suitable for initial development, proof-of-concepts, or systems with moderate data volume and simple transformation needs.
    - For a production factory environment demanding high reliability, scalability, and potentially complex data enrichment or validation logic, a **custom script (Python or Node.js)** is generally preferred. This allows for fine-grained control over error handling, connection management (using persistent connections [44]), and performance optimization (e.g., implementing efficient batch inserts [46]). The choice between Python and Node.js often depends on team familiarity and specific library needs for data manipulation.

4.3. MongoDB Schema Design

Proper schema design is critical for efficiently storing and querying the high-volume, time-stamped data generated by IoT sensors.

- **Collection Type:** Utilize MongoDB's native **Time Series Collections**, introduced

in version 5.0.[47] These collections are specifically optimized for time-series data, offering benefits like improved query performance, reduced disk usage through optimized storage and compression, and simplified data lifecycle management.[46]

- **Core Fields:** Time series collections require specific fields:
  - **timeField:** Stores the timestamp of the measurement. This field must be of BSON Date type and should be indexed.[46] (e.g., timestamp). MongoDB 6.3+ automatically creates an index on this field.[46]
  - **metaField:** Contains metadata that identifies the source of the time series and rarely changes. This field is crucial for distinguishing data from different sensors/nodes.[46] It should also be indexed (automatically in 6.3+ [46]). Querying should target specific sub-fields within the metaField (e.g., metadata.nodeId) rather than the entire object to ensure consistent query performance, as MongoDB might reorder fields internally.[46]
    - *Example metaField Structure:* { "nodeId": "ESP32_NODE_01", "sensorId": "TEMP_DS18B20_1", "sensorType": "temperature", "location": "MachineA_Zone3" }
  - **Measurements:** Fields containing the actual sensor readings (e.g., value) and any associated units or qualitative data (e.g., unit: "C").[47]
- **Granularity:** This setting defines the approximate time span covered by the data buckets MongoDB creates internally. It should be configured based on the typical interval between measurements from a single source (identified by the metaField).[46] Options are "seconds", "minutes", or "hours". If data arrives every 5 seconds, "seconds" is appropriate. If every 5 minutes, "minutes" is better. Setting granularity too coarse degrades query performance; setting it too fine increases overhead.[46] It's recommended to start with a finer granularity if unsure, as it can be coarsened later, but not made finer.[47] Custom bucketing parameters (bucketMaxSpanSeconds, bucketRoundingSeconds) offer more control in MongoDB 6.3+.[46]
- **Embedding vs. Referencing:** For the high-frequency, potentially unbounded stream of sensor readings typical in IoT (a "one-to-squillions" relationship), **referencing** is the strongly recommended pattern.[50] Each sensor reading should be stored as a separate document in the time series collection. The metaField within each reading document serves as the reference back to the specific sensor/node source. Embedding readings within a parent "device" document would quickly exceed MongoDB's 16MB document size limit and be highly inefficient for writes and queries.[50] Static information about the devices themselves (e.g., model number, installation date, physical location details) should be stored in a separate, standard MongoDB collection (e.g., devices), linked via the nodeId present in the time series data's metaField.

- **Indexing:** Ensure appropriate indexes exist on the timeField and key sub-fields within the metaField (like metadata.nodeId, metadata.sensorType, metadata.location) to optimize common query patterns (filtering by device, sensor type, time range, location).[46] MongoDB 6.3 and later automatically create a compound index on the timeField and metaField, which covers many common cases.[46] For queries needing distinct metadata values (e.g., list all unique nodeIds), avoid the distinct() command, which is inefficient on time series collections. Instead, use the $group aggregation stage, supported by an appropriate compound index.[46]
- **Optimizations:** To enhance performance and reduce storage:
  - Use batch inserts (insertMany) instead of single inserts whenever possible, setting ordered: false for higher throughput if insertion order isn't critical within the batch.[46]
  - Maintain a consistent field order within documents being inserted.[46]
  - Omit fields that contain empty objects or arrays, as these can negatively impact compression.[46]
  - Round numeric sensor data to the required precision; fewer decimal places improve compression.[46]
  - Implement Time-To-Live (TTL) indexes on the timeField to automatically delete old data and manage storage growth.[47]
- **Proposed Schema (Time Series Collection: sensor_readings)**

```JSON
{
  "timestamp": ISODate("2025-04-15T10:30:05.123Z"), // timeField (Indexed) - BSON Date
  "metadata": { // metaField (Indexed on sub-fields like nodeId, sensorType)
    "nodeId": "ESP32_NODE_01",      // String - Identifies the ESP32 node
    "sensorId": "TEMP_DS18B20_1", // String - Unique ID for the specific sensor on the node
    "sensorType": "temperature",  // String - e.g., 'temperature', 'voltage'
    "location": "MachineA_Zone3"    // String - Physical location identifier
  },
  "value": 25.7,                    // Number - The actual sensor reading
  "unit": "C"                       // String - Unit of measurement (optional but recommended)
}
```

  A separate standard collection, devices, could store static node details:

```JSON
// devices collection
{
  "_id": "ESP32_NODE_01",          // Matches nodeId in sensor_readings metadata
```

```
  "model": "ESP32-DevKitC-V4",
  "installDate": ISODate("2025-01-10T00:00:00Z"),
  "firmwareVersion": "1.2.0",
  "ipAddress_eth": "192.168.1.101", // Last known IP (optional)
  "ipAddress_wifi": "192.168.2.101" // Last known IP (optional)
  // Other static device properties
}
```

5. Nuxt.js Dashboard Development

The Nuxt.js web application serves as the user interface for monitoring the factory floor, visualizing real-time and historical data, and interacting with sensor information via a map-based layout.

5.1. Interactive Factory Map Implementation

A core requirement is displaying an interactive representation of the factory layout, derived from an AutoCAD PDF export, allowing users to select specific areas or nodes.

- **Input Format:** PDF file exported from AutoCAD.
- **Challenge:** Rendering vector-based PDF content within a web browser while enabling interactivity (overlaying a grid, handling clicks on specific map locations).
- **Rendering Techniques Comparison:**

Feature	Client-Side PDF Rendering (pdf.js wrappers)	Pre-conversion to SVG	Pre-conversion to Image (PNG/JPG)
Pros	Direct PDF display, potential vector fidelity	Web-native vector format, scalable, CSS/JS manipulation, easier interactivity	Simplest display, widest compatibility
Cons	Heavy client processing, performance issues, complex interactivity mapping [52], requires PDF transfer, SSR/Nuxt setup	Conversion step needed, fidelity depends on converter/PDF complexity, large SVGs can impact	Loses vector scalability (pixelation), manual coordinate mapping for interactivity, larger file size

	needs care [52]	perf.	
Implementation (Display)	Moderate (library setup, CSS [52])	Low (embed SVG/img tag) + Conversion step	Low (img tag) + Conversion step
Implementation (Interact)	High (coordinate mapping, layer handling)	Moderate (DOM events on SVG elements)	High (manual coordinate mapping)
Performance (Client)	Potentially High Impact	Moderate Impact (depends on SVG complexity)	Low Impact (display), High (if complex mapping)
Visual Fidelity/Scale	High (Vector)	High (Vector)	Low (Raster)
Libraries/Tools	pdf.js, @tato30/vue-pdf [52], nuxt-pdf-frame [53]	Inkscape [54], pdf2svg [54], online converters [55], Illustrator [54]	Same as SVG converters, image editors, Leaflet LImageOverlay [57]

- **Recommendation: Pre-converting the PDF to SVG** offers the most advantageous balance for this application. SVG is a web-standard vector format, ensuring scalability without pixelation on zoom. It allows for relatively straightforward implementation of interactivity by leveraging standard DOM events on SVG elements (paths, rects) representing factory areas or grid cells. While requiring an initial conversion step (using tools like Inkscape [54] or online services [55]), this avoids heavy client-side PDF processing [52] and provides a more flexible foundation for overlays and interactions compared to raster images.[57]
- **Map Component Implementation (using SVG):**
  1. **Conversion:** Convert the AutoCAD PDF to an SVG file using a chosen tool (e.g., Inkscape [54]). Optimize the SVG if necessary (remove unnecessary metadata, simplify paths).
  2. **Loading:** Embed the SVG content directly within a Vue component (e.g., in the <template> section) or load it dynamically. Ensure Nuxt configuration supports SVG loading if needed.[58]
  3. **Grid/Area Overlay:**
     - Option A (Dynamic SVG): Use Vue's rendering capabilities to dynamically generate SVG elements (e.g., <rect>, <circle>, or <path>) on top of the base factory layout SVG. These elements represent the clickable grid cells

or sensor locations. Position them based on coordinates derived from the original layout or a predefined grid system.
    - Option B (Embedded IDs): If possible during the SVG conversion or using an SVG editor, assign unique IDs to the SVG elements (e.g., <path id="machine_area_1">...</path>) that represent the desired interactive zones.
4. **Interactivity:** Attach @click event listeners to the overlay elements (Option A) or the identified SVG elements (Option B).
5. **Coordinate/ID Mapping:** When an element is clicked, retrieve its identifier (e.g., grid cell coordinates, element ID). Maintain a mapping (potentially fetched from an API or configured locally) that links these UI identifiers to the corresponding nodeId or sensorId used in the MQTT topics and MongoDB metaField.
6. **Modal Trigger:** On click, use the mapped nodeId/sensorId to identify the selected sensor/area. Trigger the opening of a modal dialog (Section 5.4), passing the identifier so the modal can display relevant real-time and historical data. General concepts for interactive maps can be found in various examples, though they often use different libraries like Leaflet or Mapbox.[57]

5.2. Real-time Data Display via MQTT over WebSockets

To display live sensor readings on the dashboard (e.g., updating values on the map overlay or within modals), MQTT over WebSockets provides the necessary real-time communication channel.

- **Technology:** WebSockets establish a persistent, bidirectional connection between the browser (Nuxt.js frontend) and the MQTT broker, allowing the broker to push messages to the client in real-time without polling.[66] MQTT provides the publish/subscribe messaging layer over this WebSocket connection.[67]
- **Client Library:** mqtt.js is the de facto standard JavaScript library for MQTT communication in browsers and Node.js, supporting MQTT over WebSockets.[37] It can be installed via npm/yarn or included via CDN.[37]
- **Implementation in Nuxt:**
    1. **Client-Side Execution:** The MQTT client requires browser APIs (WebSocket) and should only run on the client side. Encapsulate the MQTT logic within a Nuxt plugin having a .client.ts suffix or within a composable (useMQTT) that internally checks process.client before initializing the client. This prevents errors during server-side rendering. Using <ClientOnly> around components needing MQTT is another option but less clean for managing a shared connection.

2. **Connection:** Instantiate the client using mqtt.connect(brokerUrl, options).[37]
    - brokerUrl: Must use the WebSocket protocol prefix: ws://your_broker_ip:8083/mqtt or wss://your_broker_ip:8084/mqtt (adjust port and path /mqtt as needed for your broker configuration).[37] Use wss:// for secure connections (requires TLS configured on the broker).
    - options: Include clientId (generate a unique one), username, password for authentication.[66] For wss:// with self-signed certificates during development *only*, rejectUnauthorized: false might be needed, but **this is insecure for production**.[72] Production deployments require valid certificates trusted by the browser (e.g., from Let's Encrypt [72]).
3. **Event Handlers:** Implement callback functions for key client events:
    - client.on('connect', () => {... }): Log success, subscribe to relevant topics (e.g., factory/nodes/+/temperature, factory/nodes/+/voltage) using client.subscribe('topic/filter', { qos: 0 }).[37]
    - client.on('message', (topic, message) => {... }): This is the core handler for incoming data. Parse the topic to identify the node and sensor type. Parse the message payload (e.g., JSON.parse(message.toString())). Update the application's reactive state with the new value.[37]
    - client.on('error', (error) => {... }): Log connection or protocol errors.[66]
    - client.on('close', () => {... }): Log disconnection events. Implement reconnection logic if needed (mqtt.js has built-in reconnection options reconnectPeriod [73]).[66]
4. **State Management:** Use Nuxt's useState composable to create a reactive, SSR-friendly state object to hold the latest sensor readings (e.g., useState('liveSensorData', () => ({}))). Update this state within the on.message handler.
5. **Component Updates:** Vue components (map overlays, modals, gauges) should compute or directly display values from this shared reactive state. When the state updates due to a new MQTT message, the components will automatically re-render with the latest data.

Managing the WebSocket connection lifecycle and integrating it with Nuxt's reactivity system requires careful planning. Using a client-side plugin or composable provides a clean, centralized approach.

5.3. Historical Data Retrieval & Display

To allow users to analyze trends, the dashboard must provide access to historical sensor data stored in MongoDB.

- **Backend API (Nuxt 3 Nitro):** Nitro provides the server-side engine within Nuxt 3

to create API endpoints.[42]

- **Endpoint Definition:** Create API routes under the server/api/ directory. For example, server/api/historical.ts could handle requests for historical data.
- **Request Handling:** Use defineEventHandler(async (event) => {... }) to define the route handler.[43] Access query parameters (like nodeId, sensorType, startTime, endTime) from the event object (e.g., using getQuery(event)).
- **Database Interaction:**
  1. Establish a connection to MongoDB. This should ideally be managed globally via a Nitro server plugin to avoid reconnecting on every request.[74] Import Mongoose models if using Mongoose.[43]
  2. Construct a MongoDB query based on the parsed request parameters. For the sensor_readings time series collection, this typically involves:
     - A $match stage filtering on metadata.nodeId, metadata.sensorType, and timestamp using $gte (greater than or equal to startTime) and $lte (less than or equal to endTime).
     - A $sort stage on timestamp: 1 to ensure data is chronological.
     - Potentially a $project stage to select only necessary fields (timestamp, value).
  3. Execute the query using the MongoDB driver (collection.find({...}).sort({...}).toArray()) or Mongoose methods (Model.find({...}).sort({...})).
  4. Implement error handling for database operations.
  5. Return the query results as a JSON array in the response.
- **Guidance:** Snippets [42] provide examples of creating Nitro API routes and interacting with data sources, although specific MongoDB query examples need adaptation for the time series schema.

- **Frontend Data Fetching:**
  - **Trigger:** Typically initiated by user interaction within a modal dialog (e.g., selecting a sensor on the map and choosing a time range).
  - **Fetching:** Use Nuxt's data fetching composables. $fetch is well-suited for triggering fetches based on user events (like clicking a "Load History" button).[75] Call the backend API endpoint: $fetch('/api/historical', { params: { nodeId, sensorType, start, end } }).
  - **State Management:** Store the fetched historical data (an array of {timestamp, value} objects) in a local reactive variable (ref()) within the modal component. Use the status (pending, error, success) provided by $fetch or useFetch to display loading indicators or error messages.[76]
- **Data Visualization:**
  - **Libraries:** Use JavaScript charting libraries like ECharts [66] or Chart.js [78] to

display the historical data. Look for Nuxt-specific integration packages (e.g., nuxt-echarts [77]) or wrap the library in a custom Vue component.
    - **Implementation:** Create a chart component (e.g., <LineChart>) that accepts the historical data array as a prop. Pass the fetched data from the modal's state to this component. Configure the chart options to display a time-series line graph with appropriate axes (time on X, sensor value on Y), labels, and interactive tooltips. Examples like [66] show ECharts used with real-time data, adaptable for historical display.

5.4. UI Structure and Components

The dashboard requires a logical structure and reusable components for a maintainable and user-friendly interface.

- **Landing Page/Main Layout:** Provides overall application structure, navigation (e.g., sidebar or header menu), and potentially a high-level overview or entry point to the main monitoring view.
- **Map View Component:** A dedicated component responsible for rendering the interactive factory map (using the chosen SVG approach from Section 5.1) and handling click events to trigger modal displays.
- **Modal Dialog Component:** A reusable component for displaying detailed information about a selected sensor or area.
    - **Purpose:** Shows sensor identifier, location, the latest real-time value (bound to the MQTT state), controls for selecting a historical time range (e.g., date pickers, predefined ranges like "Last Hour", "Last 24 Hours"), and the historical data chart.
    - **Implementation:** Leverage a UI framework's modal component, such as Nuxt UI's <UModal> [79], or build a custom one based on examples like timb-103/nuxt-modal [80] or design patterns from themes like Black Dashboard.[81] The modal should accept the nodeId/sensorId as a prop to know which sensor's data to display and fetch.
- **Data Visualization Components:** Create reusable Vue components that wrap the chosen charting library (ECharts/Chart.js). These components accept data and configuration options as props, simplifying the display of charts within modals or potentially other dashboard sections.
- **UI Framework:** Consider using a Vue/Nuxt UI framework like Nuxt UI [79], PrimeVue [82], or others to provide pre-built components (buttons, modals, forms, layout elements), ensuring a consistent look and feel and accelerating development.[82]

6. Architectural Considerations

Building a robust factory monitoring system requires careful consideration of reliability, scalability, and security throughout the architecture.

6.1. Reliability

Ensuring continuous operation and data integrity is paramount in an industrial setting.

- **Node Connectivity:** The dual Wi-Fi/Ethernet strategy with automated failover logic (Section 2.2) is the primary defense against single-point network failures at the node level. Continuous monitoring of the active connection status (ETH.linkStatus(), WiFi.status()) and application-level checks (MQTT connectivity) within the node firmware is essential.
- **MQTT Communication:**
  - **QoS Levels:** Use MQTT QoS 1 (At Least Once) or QoS 2 (Exactly Once) for critical sensor data delivery from nodes to the broker. This adds overhead but provides delivery guarantees over unreliable networks. QoS 0 (At Most Once) may suffice for non-critical data.
  - **Broker Availability:** Configure the Mosquitto broker for high availability if possible (e.g., using clustering features if available, or deploying redundant brokers with a load balancer, although this adds complexity). Ensure broker persistence settings are appropriate to buffer messages if the pipeline is temporarily down. Cloud-based managed MQTT brokers often offer built-in high availability.[83]
  - **Client Reconnection:** Both ESP32 firmware [9] and pipeline components [35] must implement robust, automatic reconnection logic to the MQTT broker in case of temporary disconnections.
- **Data Pipeline:** The pipeline component (Node-RED flow or custom script) must be designed for resilience. This includes handling potential disconnections from both the MQTT broker and the MongoDB database, implementing retry mechanisms, and potentially utilizing persistent MQTT sessions or an intermediate queue to prevent data loss during extended downtime. Monitoring the health and throughput of the pipeline is critical.
- **Database:** Employ MongoDB replica sets for automatic failover and high availability. This is a standard feature in MongoDB Atlas [84] and should be configured for self-hosted deployments. Regular, automated backups are essential for disaster recovery.
- **ESP-NOW:** If used, acknowledge its inherent limitations in noisy 2.4GHz environments.[30] Use send callbacks (esp_now_send_status_t) for basic delivery confirmation.[21] For critical data transfer via ESP-NOW (e.g., in a gateway scenario), implement application-level acknowledgments and retry logic within

the firmware. Thorough on-site testing is crucial to validate its reliability in the specific factory environment.

6.2. Scalability

The system must be designed to handle potential growth in the number of sensor nodes, data volume, and dashboard users.

- **Nodes & Connections:**
  - **MQTT Broker:** The broker must handle concurrent connections from all active ESP32 nodes. While Mosquitto can handle many connections, performance depends heavily on server resources. High-performance brokers like HiveMQ are designed for massive scale (millions of connections).[83] Evaluate broker capacity based on the expected number of nodes.
  - **ESP-NOW:** Be mindful of the peer limits, especially for encrypted communication (max ~17 encrypted peers for ESP32 [31], 20 total peers [23]), if ESP-NOW is used extensively for node-to-node or node-to-gateway communication.
- **Data Volume & Storage:**
  - **MongoDB:** Time Series collections are inherently designed for high-volume ingestion and storage.[47] Proper schema design (granularity, metaField indexing) is key to maintaining query performance as data grows.[46] Utilize batch writes (insertMany with ordered: false) in the pipeline for optimal ingestion throughput.[46] Implement TTL indexes to automatically purge old data and manage long-term storage costs.[47]
  - **Database Scaling:** MongoDB Atlas offers seamless horizontal scaling (sharding).[84] Self-hosted MongoDB requires manual scaling planning (adding replica set members, eventually implementing sharding).
- **Data Pipeline:** The pipeline component must be able to process the aggregate message rate from all nodes. If a single instance becomes a bottleneck, consider:
  - Optimizing the processing logic (e.g., efficient data transformations, asynchronous database writes).
  - Scaling vertically (more CPU/RAM for the pipeline instance).
  - Scaling horizontally (running multiple instances of the script/Node-RED flow, potentially using shared MQTT subscriptions or a load balancer).
- **Dashboard & API:**
  - **Backend Queries:** Efficient MongoDB queries are crucial. Ensure appropriate indexes are used for historical data lookups.[46] Avoid fetching excessive data points for frontend charts; implement server-side aggregation or sampling via the Nitro API if needed for long time ranges.

    - **Real-time Updates:** The MQTT broker must handle WebSocket connections from all active dashboard users. Frontend applications should efficiently process incoming MQTT messages and update the UI without performance degradation. Optimize SVG rendering and chart updates.
    - **Nuxt Performance:** Leverage Nuxt features like code splitting and potentially server-side rendering (SSR) or island components [77] for the dashboard if initial load time becomes critical, though this adds complexity.
- **Reference Architectures:** Scalable IoT platforms often incorporate components like Apache Kafka for high-throughput buffering and stream processing between ingestion and storage/analysis layers [85], alongside brokers and databases designed for scale.[84] While Kafka isn't explicitly included here, understanding these patterns is useful if the system needs to scale significantly beyond the initial scope.

6.3. Security

Protecting the system from unauthorized access and data breaches is critical, especially in an industrial context.

- **Device-to-Broker Communication (MQTT):**
    - **Encryption:** Mandate TLS encryption for all MQTT connections (MQTTS for native TCP, WSS for WebSockets [68]). This protects data confidentiality and integrity during transit.
    - **Authentication:** Use strong username/password credentials for each ESP32 node connecting to the Mosquitto broker.[9] Consider using unique credentials per device or device group. Client certificate authentication provides a higher level of security.
- **Inter-Node Communication (ESP-NOW):**
    - **Encryption:** Utilize ESP-NOW's built-in AES-128 encryption if the data being exchanged locally is sensitive.[23] Assess the trade-off with peer limits based on security requirements and network topology.
- **Pipeline Security:**
    - **MQTT Connection:** The pipeline component must connect to the MQTT broker using TLS and provide valid authentication credentials.
    - **Database Connection:** Secure the connection from the pipeline to MongoDB using TLS and database user authentication (username/password).[87] Ensure the database user has only the necessary permissions (e.g., insert, find).
- **Database Security:**
    - Enable MongoDB authentication and authorization. Create specific database roles with least-privilege access for the pipeline user and the dashboard API

user.
    - Enable network encryption (TLS) for all connections to the database.
    - Configure firewall rules to restrict database access to only authorized components (pipeline, API server).
- **Dashboard & API Security:**
    - **Transport Security:** Serve the Nuxt.js dashboard application over HTTPS.
    - **User Authentication:** Implement robust user authentication and authorization for accessing the dashboard. Only authorized personnel should be able to view factory data.
    - **API Security:** Secure the Nitro backend API endpoints. Use authentication middleware (e.g., checking for valid session cookies or JWT tokens [43]) to protect endpoints serving historical data. Implement authorization checks to ensure users can only access data they are permitted to see.
- **Network Security:**
    - **Segmentation:** If feasible, isolate the network segment containing the IoT devices and MQTT broker from general corporate or guest networks using VLANs and firewall rules.
    - **Firewalls:** Configure firewalls to strictly control traffic flow, allowing only necessary ports and protocols between devices, the broker, the pipeline, the database, and the web server.

7. Conclusion & Recommendations

This report outlines a robust and scalable architecture for a factory monitoring system utilizing ESP32 nodes, MQTT, MongoDB, and a Nuxt.js dashboard. The proposed system incorporates dual network connectivity for enhanced node reliability, leverages ESP-NOW for optional localized communication, employs MongoDB Time Series collections for efficient data storage, and provides both real-time and historical data visualization through a modern web interface.

Key Architectural Recommendations:

1. **ESP32 Development:** Utilize the Arduino C++ environment with the PubSubClient library as the default for MQTT communication, given the focus on hardware integration and library availability. Implement custom state-machine logic for robust Ethernet/Wi-Fi failover, prioritizing Ethernet.
2. **ESP-NOW Integration:** Use ESP-NOW cautiously for short-range, non-critical local communication or via a gateway pattern to relay data from isolated nodes to the MQTT broker. Thorough on-site testing of range and reliability in the factory environment is essential. Manual pairing is recommended for initial fixed

deployments.

3. **MQTT->MongoDB Pipeline:** Evaluate Node-RED for initial setup or simpler requirements. For production scale and complex logic, develop a custom Python (paho-mqtt + pymongo) or Node.js (mqtt.js + mongodb/mongoose) script, ensuring persistent database connections and efficient batch inserts.
4. **Database Schema:** Employ MongoDB Time Series collections. Define a schema with appropriate timeField, metaField (containing nodeId, sensorType, location, etc.), and granularity. Use the referencing pattern for sensor readings, storing each reading as a document. Index timeField and relevant metaField sub-fields. Utilize TTL indexes for data retention.
5. **Dashboard Map:** Pre-convert the AutoCAD PDF factory layout to SVG format for optimal balance of vector quality, web compatibility, and interactivity implementation via standard DOM events.
6. **Dashboard Data:** Use MQTT over WebSockets (via mqtt.js) for real-time data updates, managed within a client-side Nuxt plugin or composable updating a shared reactive state. Implement Nitro API endpoints for fetching historical data from MongoDB, queried by the frontend using $fetch. Visualize data using ECharts or Chart.js.
7. **Security:** Implement TLS encryption and strong authentication at all communication points (Device-Broker, Pipeline-Broker, Pipeline-DB, Browser-Dashboard, Browser-API, DB Access). Configure database and broker security features rigorously.

Implementation Phases:

A phased approach is recommended to manage complexity and allow for iterative testing:

1. **Phase 1: Core Node & MQTT:** Develop basic ESP32 firmware to read sensors, connect via a single method (e.g., Wi-Fi), and publish data to the Mosquitto broker using PubSubClient.
2. **Phase 2: Basic Pipeline & Storage:** Set up the initial MQTT-to-MongoDB pipeline (e.g., using Node-RED) and configure the MongoDB Time Series collection schema. Verify data ingestion.
3. **Phase 3: Basic Dashboard:** Create a minimal Nuxt.js dashboard to connect via MQTT/WebSockets and display live data values for a few nodes.
4. **Phase 4: Enhanced Node Reliability:** Implement the dual connectivity (WiFi/Ethernet) and failover logic in the ESP32 firmware. Test thoroughly.
5. **Phase 5: Historical Data:** Develop the Nitro API endpoint for historical data queries and integrate historical charting (e.g., ECharts) into a basic modal

triggered from a simple list view.

6. **Phase 6: Interactive Map:** Convert the PDF to SVG. Implement the interactive SVG map component in Nuxt, including grid/area overlays and click handlers that trigger the data modal developed in Phase 5.
7. **Phase 7: ESP-NOW Integration (If Required):** Implement ESP-NOW communication (node-to-node or gateway pattern) based on specific use case needs. Test range and reliability extensively on-site.
8. **Phase 8: Hardening & Scaling:** Implement full security measures (TLS everywhere, authentication). Conduct load testing on the pipeline, broker, and database. Optimize queries and configurations based on performance results.

Final Considerations:

The success of this system hinges on rigorous testing within the actual factory environment. Network conditions (Wi-Fi coverage, Ethernet availability, RF interference) can vary significantly from lab settings. Validating the reliability of the dual connectivity failover and the practical range/robustness of ESP-NOW (if used) is crucial. Similarly, performance testing under realistic load conditions will be necessary to ensure the scalability of the MQTT broker, data pipeline, and database components. Continuous monitoring of all system components post-deployment is recommended to ensure ongoing reliability and performance.

Works cited

1. ESP32 DS18B20 Temperature Sensor with Arduino IDE (Single, Multiple, Web Server), accessed April 14, 2025, https://randomnerdtutorials.com/esp32-ds18b20-temperature-arduino-ide/
2. ESP32 DS18B20 Digital Temperature Sensor - JustDoElectronics, accessed April 14, 2025, https://justdoelectronics.com/esp32-ds18b20-digital-temperature-sensor/
3. ESP32 Useful Wi-Fi Library Functions (Arduino IDE) - Random Nerd Tutorials, accessed April 14, 2025, https://randomnerdtutorials.com/esp32-useful-wi-fi-functions-arduino/
4. Subscribe to MQTT Topic ESP32 - Programming - Arduino Forum, accessed April 14, 2025, https://forum.arduino.cc/t/subscribe-to-mqtt-topic-esp32/1074902
5. Interfacing ESP32 with W5500 ethernet module - Arduino Forum, accessed April 14, 2025, https://forum.arduino.cc/t/interfacing-esp32-with-w5500-ethernet-module/1363018
6. sparkfun/SparkFun_WebServer_ESP32_W5500: A remix of Khoi Hoang's WebServer_ESP32_W5500 - GitHub, accessed April 14, 2025, https://github.com/sparkfun/SparkFun_WebServer_ESP32_W5500
7. ESP32 WiFiMulti Connect to the Strongest Wi-Fi Network - Random Nerd

Tutorials, accessed April 14, 2025, https://randomnerdtutorials.com/esp32-wifimulti/
8. ESP32 With Two Wifi - Programming - Arduino Forum, accessed April 14, 2025, https://forum.arduino.cc/t/esp32-with-two-wifi/1112287
9. ESP32 MQTT Publish Subscribe with Arduino IDE - Random Nerd Tutorials, accessed April 14, 2025, https://randomnerdtutorials.com/esp32-mqtt-publish-subscribe-arduino-ide/
10. MQTT on ESP32: A Beginner's Guide - EMQX, accessed April 14, 2025, https://www.emqx.com/en/blog/esp32-connects-to-the-free-public-mqtt-broker
11. MQTT Tutorial for NodeMCU-32S Using ESP32 WiFi Module - PubNub, accessed April 14, 2025, https://www.pubnub.com/blog/pubsub-nodemcu-32s-esp32-mqtt-pubnub-arduino-sdk/
12. MQTT Tutorial using Arduino Framework - YouTube, accessed April 14, 2025, https://www.youtube.com/watch?v=7mYNaVNoIg0
13. ESP32 MQTT - Publish DHT11/DHT22 Temperature and Humidity Readings (Arduino IDE), accessed April 14, 2025, https://randomnerdtutorials.com/esp32-mqtt-publish-dht11-dht22-arduino/
14. ESP32 MQTT client - Programming - Arduino Forum, accessed April 14, 2025, https://forum.arduino.cc/t/esp32-mqtt-client/973629
15. ESP32 PubSubClient, accessed April 14, 2025, https://esp32.com/viewtopic.php?t=41701
16. Python vs JavaScript: Key Differences, Features - eLuminous Technologies, accessed April 14, 2025, https://eluminoustechnologies.com/blog/python-vs-javascript/
17. ESP32 MicroPython MQTT TLS - DEV Community, accessed April 14, 2025, https://dev.to/bassparanoya/esp32-micropython-mqtt-tls-28fd
18. MicroPython - Getting Started with MQTT on ESP32/ESP8266 ..., accessed April 14, 2025, https://randomnerdtutorials.com/micropython-mqtt-esp32-esp8266/
19. Intro to MicroPython on the ESP32: MQTT, accessed April 14, 2025, https://boneskull.com/micropython-on-esp32-part-2/
20. MicroPython: MQTT Publish DHT11/DHT22 with ESP32/ESP8266 | Random Nerd Tutorials, accessed April 14, 2025, https://randomnerdtutorials.com/micropython-mqtt-publish-dht11-dht22-esp32-esp8266/
21. Getting Started with ESP-NOW (ESP32 with Arduino IDE) - Random Nerd Tutorials, accessed April 14, 2025, https://randomnerdtutorials.com/esp-now-esp32-arduino-ide/
22. ESP-NOW Two-Way Communication Between ESP32 Boards - Random Nerd Tutorials, accessed April 14, 2025, https://randomnerdtutorials.com/esp-now-two-way-communication-esp32/
23. espnow — support for the ESP-NOW wireless protocol — MicroPython latest documentation, accessed April 14, 2025, https://docs.micropython.org/en/latest/library/espnow.html
24. Radio communication for ESP8266 and ESP32 with ESPNOW - eMariete, accessed

April 14, 2025, https://emariete.com/en/esp8266-esp32-espnow/
25. Comparative Performance Study of ESP-NOW, Wi-Fi, Bluetooth Protocols based on Range, Transmission Speed, Latency, Energy Usage and Barrier Resistance | Request PDF - ResearchGate, accessed April 14, 2025, https://www.researchgate.net/publication/355656368_Comparative_Performance_Study_of_ESP-NOW_Wi-Fi_Bluetooth_Protocols_based_on_Range_Transmission_Speed_Latency_Energy_Usage_and_Barrier_Resistance
26. How To Get Started With ESP-NOW - DigiKey, accessed April 14, 2025, https://www.digikey.com/en/maker/tutorials/2024/how-to-get-started-with-esp-now
27. ESP-NOW: Auto-pairing for ESP32/ESP8266 - Random Nerd Tutorials, accessed April 14, 2025, https://randomnerdtutorials.com/esp-now-auto-pairing-esp32-esp8266/
28. Using ESP-NOW for bi-directional communication between esp32's, accessed April 14, 2025, https://esp32.com/viewtopic.php?t=38079
29. ESP-NOW vs LoRa range : r/esp32 - Reddit, accessed April 14, 2025, https://www.reddit.com/r/esp32/comments/j7l4fy/espnow_vs_lora_range/
30. (PDF) Indoor Performance Evaluation of ESP-NOW - ResearchGate, accessed April 14, 2025, https://www.researchgate.net/publication/369626626_Indoor_Performance_Evaluation_of_ESP-NOW
31. ESP-NOW - - — ESP-FAQ latest documentation - Technical Documents, accessed April 14, 2025, https://docs.espressif.com/projects/esp-faq/en/latest/application-solution/esp-now.html
32. (PDF) From Data to Decisions: A Smart IoT and Cloud Approach to ..., accessed April 14, 2025, https://www.researchgate.net/publication/388075621_From_Data_to_Decisions_A_Smart_IoT_and_Cloud_Approach_to_Environmental_Monitoring
33. Node-Red + Sensors – Store data in MongoDB Database and exploring Thingspeak & MQTTool – Internet of Things - Leelu Chowdary Ravi, accessed April 14, 2025, https://leeluchowdaryravi.wordpress.com/2018/02/26/node-red-sensors-store-data-in-mongodb-database-and-exploring-thingspeak-mqttool/
34. How can i save data from Mqtt to Mongo? - Reddit, accessed April 14, 2025, https://www.reddit.com/r/MQTT/comments/1hnht7f/how_can_i_save_data_from_mqtt_to_mongo/
35. Paho MQTT Python client: a tutorial with examples | Cedalo blog, accessed April 14, 2025, https://cedalo.com/blog/configuring-paho-mqtt-python-client-with-examples/
36. Build A Python Database With MongoDB, accessed April 14, 2025, https://www.mongodb.com/resources/languages/python
37. JavaScript MQTT Client: A Beginner's Guide to MQTT.js | EMQ - EMQX, accessed April 14, 2025, https://www.emqx.com/en/blog/mqtt-js-tutorial
38. javascript - How to subscribe client and save the data to MongoDB ..., accessed

April 14, 2025, https://stackoverflow.com/questions/42809581/how-to-subscribe-client-and-save-the-data-to-mongodb-in-node-js-mosca-and-mqtt
39. Integrating MQTT Data to MongoDB | Cedalo, accessed April 14, 2025, https://cedalo.com/blog/mqtt-to-mongodb-integration/
40. How to Subscribe on Multiple topic using PAHO-MQTT on python - Stack Overflow, accessed April 14, 2025, https://stackoverflow.com/questions/48942538/how-to-subscribe-on-multiple-topic-using-paho-mqtt-on-python
41. Keeping MQTT Data History with Node.js - ReductStore, accessed April 14, 2025, https://www.reduct.store/blog/tutorials/iot/how-to-keep-mqtt-data-node
42. How to use Nodejs as Backend in your Nuxtjs App: A Beginner Tutorial - Bukoye Olaniyi's Blog, accessed April 14, 2025, https://radiantcodes.hashnode.dev/how-to-use-nodejs-as-backend-in-your-nuxtjs-app-a-beginner-tutorial
43. Nuxt mongoose integration guide | Restackio, accessed April 14, 2025, https://www.restack.io/p/nuxt-mongoose-answer-integration-guide
44. How to use collection in MQTT onMessage event - Node.js Frameworks - MongoDB, accessed April 14, 2025, https://www.mongodb.com/community/forums/t/how-to-use-collection-in-mqtt-onmessage-event/305457
45. Python-based MQTT #3: HBMQTT, Paho & MongoDB - YouTube, accessed April 14, 2025, https://www.youtube.com/watch?v=Ywtt-Mt1Emc
46. Best Practices for Time Series Collections - Database Manual v7.0 ..., accessed April 14, 2025, https://www.mongodb.com/docs/v7.0/core/timeseries/timeseries-best-practices/
47. MongoDB Time Series Data, accessed April 14, 2025, https://www.mongodb.com/resources/products/capabilities/mongodb-time-series-data
48. Optimizing Data Ingestion for High-Frequency IoT Sensors in MongoDB Time Series Database, accessed April 14, 2025, https://www.mongodb.com/community/forums/t/optimizing-data-ingestion-for-high-frequency-iot-sensors-in-mongodb-time-series-database/254983
49. IoT and MongoDB: Powering Time Series Analysis of Household Power Consumption, accessed April 14, 2025, https://www.mongodb.com/developer/products/atlas/iot-mongodb-powering-time-series-analysis-household-power-consumption/
50. Data Modeling - Database Manual v8.0 - MongoDB Docs, accessed April 14, 2025, https://www.mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/
51. MongoDB Schema Design Best Practices - YouTube, accessed April 14, 2025, https://m.youtube.com/watch?v=leNCfU5SYR8&pp=ygUMI21vbmdvZGJ0aXBz
52. TaTo30/vue-pdf: PDF component for Vue 3 - GitHub, accessed April 14, 2025, https://github.com/TaTo30/vue-pdf

53. How to printing and preview pdf file - Nuxt - Reddit, accessed April 14, 2025, https://www.reddit.com/r/Nuxt/comments/1i1uh2i/how_to_printing_and_preview_pdf_file/
54. How To Convert PDF to SVG for Free in 5 Ways - Wondershare PDFelement, accessed April 14, 2025, https://pdf.wondershare.com/convert-pdf/convert-pdf-to-svg.html
55. Convert your PDF to SVG for Free Online - Zamzar, accessed April 14, 2025, https://www.zamzar.com/convert/pdf-to-svg/
56. PDF to SVG converter - quick, online, free - PDF24 Tools, accessed April 14, 2025, https://tools.pdf24.org/en/pdf-to-svg
57. LImageOverlay | Nuxt Leaflet, accessed April 14, 2025, https://leaflet.nuxtjs.org/components/l-image-overlay
58. Nuxt Configuration, accessed April 14, 2025, https://nuxt.com/docs/api/nuxt-config
59. Build an Interactive Map with React Leaflet and Strapi, accessed April 14, 2025, https://strapi.io/blog/how-to-build-an-interactive-map-with-react-leaflet-and-strapi
60. Making Interactive Canvas Maps (QGIS3) - QGIS Tutorials and Tips, accessed April 14, 2025, https://www.qgistutorials.com/en/docs/3/interactive_canvas_maps.html
61. Examples | Mapbox GL JS, accessed April 14, 2025, https://docs.mapbox.com/mapbox-gl-js/example/
62. Custom Overlays | Maps JavaScript API - Google for Developers, accessed April 14, 2025, https://developers.google.com/maps/documentation/javascript/customoverlays
63. Leaflet Examples - MapTiler, accessed April 14, 2025, https://docs.maptiler.com/leaflet/examples/
64. Basic interactive svg map for Nuxt. - GitHub, accessed April 14, 2025, https://github.com/bryanspearman/interactive-svg-map-with-nuxt
65. Building an Interactive SVG Map in Webflow - YouTube, accessed April 14, 2025, https://www.youtube.com/watch?v=Iz_6IBfCtBQ
66. Adding real time using MQTT + Websockets - Ubidots Help Center, accessed April 14, 2025, https://help.ubidots.com/en/articles/9572884-adding-real-time-using-mqtt-websockets
67. What is MQTT over WebSockets and when is it used? - Engineers Garage, accessed April 14, 2025, https://www.engineersgarage.com/mqtt-over-websockets-arduino-iot/
68. A Quickstart Guide to Using MQTT over WebSocket | EMQ - EMQX, accessed April 14, 2025, https://www.emqx.com/en/blog/connect-to-mqtt-broker-with-websocket
69. mop/docs/using-mqtt-over-websocket.md at master - GitHub, accessed April 14, 2025, https://github.com/streamnative/mop/blob/master/docs/using-mqtt-over-websocket.md

70. How to Monitor Real time MQTT Data in Browser using JavaScript and WebSocket | IoT | IIoT, accessed April 14, 2025, https://www.youtube.com/watch?v=xgf3V6A-5Tg
71. Setting up SSE/Websockets in Nuxt 3 - Reddit, accessed April 14, 2025, https://www.reddit.com/r/Nuxt/comments/192pkxd/setting_up_ssewebsockets_in_nuxt_3/
72. mqtt over Secure WebSockets using mqtt.js - Stack Overflow, accessed April 14, 2025, https://stackoverflow.com/questions/78162170/mqtt-over-secure-websockets-using-mqtt-js
73. How to Use MQTT in the Vue Project - DEV Community, accessed April 14, 2025, https://dev.to/emqx/how-to-use-mqtt-in-the-vue-project-1em7
74. Using Vue.js & Nuxt.js with MongoDB & Docker | Metadrop, accessed April 14, 2025, https://metadrop.net/en/articles/using-vuejs-nuxtjs-mongodb-docker
75. NUXT 3 fetching data on the client side - vue.js - Stack Overflow, accessed April 14, 2025, https://stackoverflow.com/questions/75698949/nuxt-3-fetching-data-on-the-client-side
76. Data Fetching · Get Started with Nuxt, accessed April 14, 2025, https://nuxt.com/docs/getting-started/data-fetching
77. Server-Side Rendering - Nuxt ECharts - Nuxt Module for Apache ECharts™, accessed April 14, 2025, https://echarts.nuxt.dev/guides/ssr/
78. How To Build Realtime Charts With JavaScript and WebSocket - PieHost, accessed April 14, 2025, https://piehost.com/blog/building-realtime-charts-in-javascript
79. Modal Vue Component - Nuxt UI, accessed April 14, 2025, https://ui.nuxt.com/components/modal
80. timb-103/nuxt-modal: Custom modal component for Nuxt 3. - GitHub, accessed April 14, 2025, https://github.com/timb-103/nuxt-modal
81. Modal | Nuxt Black Dashboard @ Creative Tim, accessed April 14, 2025, https://www.creative-tim.com/learning-lab/nuxt/modal/black-dashboard
82. Which UI framework is best for building dashboards and business apps? : r/vuejs - Reddit, accessed April 14, 2025, https://www.reddit.com/r/vuejs/comments/1j5j3kr/which_ui_framework_is_best_for_building/
83. awesome-mqtt/README.md at master - GitHub, accessed April 14, 2025, https://github.com/hobbyquaker/awesome-mqtt/blob/master/README.md?plain=1
84. Building a Reliable and Scalable IoT Platform - HiveMQ, accessed April 14, 2025, https://www.hivemq.com/blog/building-a-reliable-and-scalable-iot-platform/
85. IoT Reference Architecture and Implementation Guide Using ..., accessed April 14, 2025, https://www.confluent.io/blog/build-an-iot-platform-with-confluent-and-mongodb/
86. MongoDB & IIoT: A 4-Step Data Integration, accessed April 14, 2025,

https://www.mongodb.com/blog/post/mongodb-iiot-4-step-data-integration
87. Using MongoDB With Node-RED • FlowFuse, accessed April 14, 2025, https://flowfuse.com/node-red/database/mongodb/