# Final Report
# OPC UA Industrial Plant HUB

### Final Report for the Distributed Systems Course [A.A 2025-2026]

Federico Capitanio
federico.capitanio@studio.unibo.it

February 19, 2026

This report illustrates **OPCUA Industrial Plant Hub**, a software developed to demonstrate the fundamental principles of Distributed Systems by integrating them with the world of industrial automation.

In the world of industrial automation, the issue of data collection from machines and machine lines is becoming increasingly important. Therefore, this project aims to provide a distributed data collection system for different production lines, which may also be distributed across the territory.

The project is based on the use of IoT devices, which are connected to the production line and collect data on the production process. The data collected is then sent to a central server, which stores it and allows it to be analysed. The data can also be sent to the machines themselves, allowing them to be controlled and optimised.

The underlying idea is that data is collected from the machine line using the **OPC UA** protocol, widely used in industry, saved and then retrieved via **HTTP API**.

Unlike traditional data collection architectures based on a single physical node that reads data, the architecture of this project consists of three HUB nodes that collect **OPC UA** data, synchronised via Anti-Entropy and Gossip protocols, with an nginx load balancer that distributes client requests and implements data encryption via the **TLS** protocol.

The project emphasises the **AP** behaviour of the **CAP Theorem**, i.e. it focuses on the aspects of **Partition Tolerance** and **High Availability**, thus ensuring that clients can always access data even in the presence of network partitions or individual node failures. The **Strong Consistency** concept is therefore not guaranteed, promoting data that is not updated to the latest possible value over a longer wait time that would result from the application of **Strong Consistency** itself.

The system allows OPC UA sources to be added dynamically, by implementing an **on the fly** automation line connection mode, allowing individual

machines or entire lines to be connected and added without the system needing to be restarted.

# 1 Concept

This project focuses on the development of a distributed system for collecting industrial data from different lines of automatic machines or plants, allowing for its centralisation.

## 1.1 Product Type

- A web interface that shows all the possible HTTP APIs and make HTTP requests available directly from the page itself;
- Setup scripts, one for Windows and one for Linux, to create user credentials that will allow ;
- OPC UA server simulator to instance various OPC UA nodes and each one is able simulate real machine data;
- CLI commands to test the system itself, for instance switching off a node and restart it later.

## 1.2 Use Case Description

- *What is the software doing?* The software initially instantiates three different nodes (there may be more). Each node connects to several OPC UA servers, browses all the variables present within the server itself, connects and starts collecting values every N seconds. The collected data is saved locally and replicated via Anti-Entropy with the other nodes performing the same task. The nodes also exchange messages with each other to know which nodes are actually active. An nginx server acting as a load balancer redirects HTTP requests coming from outside to the nodes that are alive and periodically tries to determine which of the previously down nodes are back online.
- *when and how frequently do they interact with the system?* Users who will use the application will be anyone who makes an HTTP request to port 443 of the nginx server.
- *when and how frequently do they interact with the system?* Since the software is not something like a game, users are defined as those who use the system itself and they can sent requests at any time as long as they don't exceed the Nginx brute force attack protection.
- *how do they interact with the system? which devices are they using?* The user could interact with the system with every HTTP(s) client, so GUI client like Postman, CLI commands like *curl* or even by building a front-end web page that use these application's API to show interactive web page to monitor machine performance or send command to the application itself.

### 1.3 Why is distribution needed?

Distribution is a strictly necessary choice, as availability and reliability requirements are non-negotiable and constantly demanded in the industrial sector. Below are the main reasons why distribution is necessary.

- *Fault Tolerance:* Each node (Ingestor) operates autonomously, collecting data from various OPC UA servers. Therefore, if any node fails, the system continues to operate with the remaining nodes. Furthermore, when this happens, Anti-Entropy mechanisms ensure automatic resynchronisation once the node (or nodes) becomes operational again.
- *High Availability:* Industrial plants are designed to operate 24/7 most of the time, requiring constant monitoring systems with high availability.
- *Scalability:* The possible increase in the number of monitored machines or lines themselves and the possibility that more users and applications will need access to the data.
- *Geographic Distribution:* Various industrial companies operate on sites that can cover different geographically distributed areas, with different plants in different national and international areas. Potentially, through the evolution of the system, clients will be able to connect to the node that is physically closest to them.

## 2 Requirements Elicitation and Analysis

This section aims to outline the system requirements and their implementation to explain what the software must actually do. The main focus is to explain what the supplied software must do when it comes into operation.

### 2.1 Glossary

Before proceeding with the requirements analysis, it is necessary to define some domain-specific terms:

**OPC UA - Open Platform Communications Unified Architecture** [1] An open international standard for secure and reliable data exchange, essential in the world of Industry 4.0 and industrial IoT. It was developed by the OPC Foundation and enables interoperability between heterogeneous devices and systems (PLCs, HMIs, clouds) regardless of manufacturer or operating system. Its strengths are the ability to leverage the integrated security offered by the protocol itself, platform independence, scalability, and the ability to perform data modeling, meaning that more complex structures can be managed depending on the context and/or the company implementing the protocol.

**PLC** An industrial computer that has been ruggedized and adapted for the control of manufacturing processes, such as assembly lines, machines, robotic devices, or any activity that requires high reliability and usually works in real time.

**OEE - Overall Equipment Effectivness** [16] Main KPI for measuring the production capacity of a manufacturing company. It is often used in Lean Manufacturing to achieve operational excellence.

**Tag** OPC UA node or, generally speaking, a variable that could be collected, exposed or saved.

**HUB Node** Node of the distributed system provided by this software that collects data from OPC UA servers, stores it locally, and exposes it via REST API. It participates in the distributed synchronization protocols described below.

**Anti-Entropy** Synchronization protocol that guarantees eventual consistency through periodic exchanges of state differences between nodes.

**Gossip Protocol** Peer-to-peer communication protocol in which each node periodically propagates information about its own health and that of other observed nodes, enabling distributed failure detection.

**Lamport Clock** Logical timestamp that establishes a causal order between events in a distributed system without requiring physical clock synchronization.

**Eventual Consistency** Consistency model in which, given sufficient time and the absence of new writes, all nodes in the system will converge to the same state.

**Load Balancer** Component that distributes client requests among available HUB nodes, implementing health checks to detect non-functioning nodes.

**HTTP - HyperText Transfer Protocol** [4] Fundamental protocol for exchanging data on the World Wide Web. Based on a client-server model, it allows clients(browser as an example) to request and receive resources.

**TLS - Transport Layer Security** [8] Cryptographic protocol designed to secure communications over a computer network. It is widely used in applications such as email, instant messaging and Voice over IP (VoIP), but it's best known for securing Hypertext Transfer Protocol Secure (HTTPS). There are different versions, and the most recent one is currently 1.3. However, different legacy versions are widely used and leave security issues all over the internet.

**JWT (JSON Web Token)** [6] Is an open standard (RFC 7519) that defines a compact, self-contained way to securely transmit information between parties as a JSON object. It is commonly used for authentication and authorization, allowing servers to verify user identity without database lookups. These tokens are digitally signed for security, ensuring data integrity.

**RBAC (Role-Based Access Control)** [3] Security method that regulates access to system resources based on defined roles rather than individual user identities. Assigns specific permissions to roles (e.g., admin, user, manager), simplifying security management.

## 2.2 Functional Requirements

1. **OPC UA Server Connection**

   The system must connect to configured OPC UA servers and periodically read the values of the defined variables (OPC UA nodes).

   *Acceptance Criterion*: Each HUB node can connect to one or more OPC UA servers, authenticate correctly, and read values at a configurable frequency (default: every 5 seconds). The data read includes the OPC UA server timestamp and status code.

2. **Data Persistence**

   The system must store the data collected from the OPC UA servers locally in a relational database on each HUB node.

   *Acceptance Criterion*: The data are saved in a structured format (lamport_clock, tag, node_id, value, timestamp, quality, source_server, server_name) with support for temporal queries and the db itself persists even after the node is restarted.

3. **REST API Data Exposure**

   The system must expose the collected data through authenticated REST APIs, allowing clients to query current and historical values.

   *Acceptance Criterion*: API for data retrieve:

   - GET `/api/v1/tags` - list all the tags for all the OPC UA server
   - GET `/api/v1/servers/server_name/tags` - list all the tags for a specific OPC UA server source
   - GET `/api/v1/servers/server_name/tags/tag/latest` - retrieve last value for a specific tag that is exposed in one specific OPC UA server
   - GET `/api/v1/servers/server_name/tags/tag/history` - retrieve value for a specific tag that is exposed in one specific OPC UA server into a given time range
   - GET `/api/v1/servers` - list all configured OPC UA servers

   Every HTTP request need valid JWT into the Authorization header.

4. **Dynamic Server Addition**

   The system must allow the addition of new OPC UA servers without the need to restart the HUB node.

*Acceptance Criterion*: An admin user can invoke HTTP POST request at `/api/v1/admin/opc-server` path including new OPC UA server configuration including a name for the source and the endpoint on which the data will be retrieved. The system immediately starts collecting data from the new server and propagates the configuration to the other HUB nodes via Anti-Entropy. The maximum propagation time is 30 seconds.

5. **Data Synchronization Between Nodes**

   The system must synchronize the data collected between the three (or more) HUB nodes to ensure that each node has a consistent copy of the data.

   *Acceptance Criterion*: The Anti-Entropy protocol periodically compares the state between nodes every 10 seconds using Lamport clocks. Differences are resolved using the Last-Write-Wins strategy. After 30 seconds without new writes, all nodes must converge to the same state.

6. **Failure Detection**

   The system must automatically detect when a HUB node becomes unreachable.

   *Acceptance Criterion*: The Gossip protocol implements heartbeat every 5 seconds between nodes. A node is considered "suspected failed" after 3 consecutive missed heartbeats (15 seconds). The information is propagated to other nodes via Gossip with a detection probability of >95% within 20 seconds.

7. **Automatic Traffic Redirection**

   The system must automatically redirect client traffic from non-functioning nodes to operational nodes.

   *Acceptance Criterion*: The nginx load balancer performs HTTP health checks every 30 seconds on the /health endpoint of each HUB node. If a node does not respond or returns a 500 status, it is automatically removed from the routing pool. Recovery occurs automatically when the node returns to operation.

8. **User Authentication**

   The system must authenticate users via credentials and issue JWT tokens for API access.

   *Acceptance Criterion*: The POST endpoint `/api/v1/auth/token` accepts username and password, verifies the credentials, and returns a JWT valid for 24 hours. The token includes the user's role (admin or user).

9. **Role-Based Authorization**

   The system implements role-based access control.

   *Acceptance Criterion*:

   - Ruolo `user`: can only read data;
   - Ruolo `admin`: can read and modify configurations.

   Unauthorized requests return HTTP 403 Forbidden.

## 2.3 Non-Functional Requirements

1. **Eventual Consistency**

   The system prioritizes availability and partition tolerance over strong consistency, implementing eventual consistency.

   *Acceptance Criterion*: During network partitions or partial failures, nodes continue to accept writes (server addition) and reads. Convergence to the same state is guaranteed within 60 seconds of connectivity restoration. Different clients may observe temporarily different snapshots.

2. **Response Time**

   REST APIs must respond with acceptable latency for industrial monitoring applications.

   *Acceptance Criterion*: The following tests have been performed in a LAN(Local Area Network). So, since the system is created to work also in different networks, other tests would be performed to estimate reponse time in the specific case. The first two items of the following list have been tested under a load of 50 requests per second distributed across the three nodes.

   - Live data reading: P95 latency $< 100$ms
   - Historical data reading: P95 latency $< 1$s for 1000 record
   - Aggiunta server: $< 100$ms (local processing + asynchronous propagation)

3. **Data Integrity**

   The collected data must be stored with guaranteed integrity, preserving the original timestamps of the OPC UA server.

   *Acceptance Criterion*: Data is never modified after initial entry. Each record includes the timestamp of the source OPC UA server, allowing for correct temporal sorting even in the presence of clock drift between HUB nodes.

4. **Scalability - Server OPC UA**

   The system must support a growing number of OPC UA servers without significant performance degradation.

   *Acceptance Criterion*: The system must manage a number of OPC UA servers as required by the end user. Consider scaling out the HUB nodes, trying to maintain a 1:10 ratio between HUB nodes and OPC UA servers, obviously always leaving a minimum of 3 HUB nodes.

5. **Security - Data in Transit**

   All communications between the client and the system must be encrypted.

   *Acceptance Criterion*: The nginx load balancer implements TLS 1.3 [8] termination with valid certificates, even if in this case they're provided as self-signed, since a

CA authority request could be submitted for receive a valid one that would be recognized and verifiable all over the internet. Unencrypted HTTP connections are redirected directly to HTTPS. If the system is to be published on a public domain, tests like on *www.ssllabs.com/ssltest* site are recoomend to ensure a high level of security.

6. **Security - Authentication**

   The system must prevent unauthorized access to APIs.

   *Acceptance Criterion*: Every request to the APIs (except /auth/login) requires a valid JWT. Expired, malformed, or invalidly signed tokens are rejected with HTTP 401. The JWT secret is shared among all HUB nodes and it's auto-generated during application setup.

## 2.4 Implementation Requirements

1. **Programming Language: Python**

   The backend of the HUB nodes must be developed in Python 3.11+.

   *Justification*: Python [12] offers recent libraries for OPC UA (asyncua), high-performance web frameworks with asynchronous support (FastAPI), and a rich ecosystem for rapid development. The I/O-bound nature of the system (network + database) benefits from Python's asynchronous event loop.

2. **Web Framework: FastAPI**

   REST APIs must be implemented with FastAPI.

   *Justification*: FastAPI [15] provides high performance comparable to Node.js thanks to Starlette and uvicorn, automatic data validation with Pydantic, automatic OpenAPI documentation generation, and native support for asynchronous operations essential for non-blocking I/O to OPC UA servers and databases.

   Website for details: https://fastapi.tiangolo.com/.

3. **ORM: SQLAlchemy**

   Access to the database must use SQLAlchemy as the ORM.

   *Justification*: SQLAlchemy [13] is the standard ORM in Python, providing database abstraction (SQLite in development, PostgreSQL in production possible), support for schema migrations via Alembic, and type-safe queries that reduce runtime errors.

4. **Database: SQLite**

   Each HUB node uses SQLite as its local database.

   *Justification*: SQLite [14] is embedded, does not require a separate server, has a minimal footprint, and is sufficient for expected workloads (write-intensive but not concurrent between different processes). Each node operates on an independent database, replicated via Anti-Entropy.

5. **Data validation and serialization: Pydantic**

   LData validation and serialization must be implemented using Pydantic.

   *Justification*: Pydantic [11] provides automatic data validation using Python type hints, ensuring data integrity at the API boundaries. It integrates seamlessly with FastAPI for automatic validation of requests and responses, generates JSON schemas for API documentation (see below: Swagger API), and prevents common errors such as type mismatches or missing required fields. Pydantic define clear models for:

   - request/response API payload (es. server config, authentication credentials)
   - configuration file validation (startup parameter, JWT secret, ...)
   - database model with SQLAlchemy integration

   This approach provides type safety at runtime, catching errors before they propagate through the system, and reduces the need for manual validation code. For example, when adding a new OPC UA server via POST `/api/v1/admin/opc-servers`, Pydantic automatically validates that the endpoint URL is well-formed, that the required fields are present, and that the data types are correct, returning clear error messages to the client in case of failed validation.

6. **Containerization: Docker**

   The system must be deployable via Docker and Docker Compose.

   *Justification*: Docker [2] guarantees environment reproducibility, simplifies multi-node deployment on a single machine for development/testing, isolates dependencies, and make scale out easier. Docker Compose orchestrates the 3(or even more) simulated HUB + nginx + OPC UA server(to simulate real servers) nodes with a single command.

7. **Load Balancer: nginx**

   Load balancing and TLS termination must be handled by nginx.

   *Justification*: Nginx [10] is extremely powerful for reverse proxy, supports native health checks, simple configuration for round-robin, and robust TLS management. Extensively tested in production and with minimal footprint.

8. **OPC UA Library: asyncua**

   The connection to OPC UA servers must use the asyncua library.

   *Justification*: Asyncua [5] is a pure Python implementation of OPC UA client with native asynchronous support, compatible with FastAPI event loop, actively maintained, and supports necessary OPC UA features (browsing, subscriptions, read multiple nodes).

   Website for details: https://github.com/FreeOpcUa/opcua-asyncio

9. **Authentication JWT: python-jose**

   Authentication must implement JSON Web Tokens by integrating it with Python.

   *Justification*: Python-Jose [9] is a popular Python library for working with JWT, which support various algorithms and allows the creation, decoding, and verification of tokens, ideal for secure authentication.

   Website for details: https://python-jose.readthedocs.io/en/latest/.

10. **Development Environment: PowerShell Script or Bash Script**

    Setup and authentication scripts must be provided in PowerShell and Bash.

    *Justification*: since the operating system on which the software will run is not defined, two setup scripts are provided, one in PowerShell and one in Bash, allowing the user to run the script that best suits their needs. Within the script, TLS certificates are automatically generated, or possibly recreated if they are detected but expired, and the user is prompted to enter the username and password of the admin user, in addition to the option of creating additional users who will be classified as normal users (see previous section - RBAC). The information provided by the user will be entered into an environment variables file.

## 2.5 Relevant Distributed System Features

The following analysis explains which characteristics of distributed systems are critical to the success of the project and which are less relevant in the specific context of an industrial data collection system.

### 2.5.1 Fault Tolerance e Dependability

**Relevance: HIGH** - Fault tolerance is the fundamental pillar of the system, given that industrial plants require continuous and constant monitoring.

- **Availability**: The system must guarantee high availability since the final user, like the customer that could buy it, would track all the machine status in order to calculate machine performance and quality, introducing OEE calculation concept. What's more, most of the company the buy industrial automation machine, works 24/7 and need continuos monitoring about machines' state. With three(at least) operational HUB nodes, the system tolerates the failure of one or two nodes while continuing to serve requests without interruption. As the number of HUB nodes increases, the fault tolerance capacity clearly grows. An assessment must therefore be made to understand how many requests will be made to the distributed system and how frequent they will be, so that the number of nodes can be scaled to a sufficient number.

- **Partion Tolerance** The system must continue to work correctly even in the presence of network par- titions between HUB nodes. If the network has partitions and

nodes are splitted into isolated groups, each group continues to serve clients connected to it. When the partition resolves, the Anti-Entropy protocol automatically reconciles divergent states.

- **Reliability**: Data loss is unacceptable in an industrial context, where each sample represents the status of a machine at a specific moment in time. Replication via Anti-Entropy ensures that the data collected by one node is propagated to all other nodes, preventing data loss even in the event of a sudden crash.
- **Recovery Time**: The system implements automatic recovery mechanisms:
  - Gossip protocol detects failure within ∼15 seconds (3 heartbeat missed).
  - Load balancer remove not working nodes from pool in ∼30 seconds.
  - Anti-entropy resynchronizes restored nodes in <60 seconds.
- **Integrity**: Each piece of data includes the original OPC UA timestamp and Lamport clock for causal ordering, ensuring that temporal integrity is preserved even with clock drift between nodes.

### 2.5.2 High Availability

**Relevance: HIGH** - High availability is not only desirable but necessary for industrial use cases:

- **No Single Point of Failure**: Each component is redundant:
  - 3 independent HUB nodes collect data in parallel.
  - Each node has a local database (no dependency on a central database).
  - Load balancer can be replicated (not implemented in this prototype but architecturally possible).
- **Partition Tolerance**: During network partitions, each isolated group continues to operate autonomously. When the partition resolves, Anti-Entropy automatically reconciles divergent states.

### 2.5.3 Transparency

**Relevance: HIGH** - Transparency is essential for simplifying the use of the system:

- **Location Transparency**: clients interact with a single endpoint (the load balancer) without knowing which HUB node is actually serving the request. This hides the physical distribution and allows horizontal scaling by adding nodes without changing client configurations.
- **Replication Transparency**: clients are unaware that the data is replicated across three nodes. They send a single GET request and receive a response, regardless of which replica serves the request.

- **Failure Transparency (parziale)**: HUB node failures are completely transparent to clients: requests continue to be served by the surviving nodes. However, eventual consistency implies that different clients may temporarily observe slightly different states during synchronization windows, so failure transparency is not total.
- **Access Transparency**: REST APIs are identical regardless of the node on which the request is forwarded,ensuring a uniform interface.

### 2.5.4 Scalability

**Relevance: HIGH** - Scalability is essential to adapt to the growth of the industrial plant:

- **Horizontal Scalability - HUB nodes**: HUB nodes can be added to the cluster to further distribute the load. The system is designed to be stateless at the level of individual HTTP requests (authentication via stateless JWT), facilitating the addition of new nodes.
- **Horizontal Scalability - OPC UA server**: New OPC UA servers can be added dynamically via admini APIs without restarting. The configuration is automatically propagated to all HUB nodes via Anti-Entropy.
- **Request Scalability**: The load balancer distributes requests in round-robin fashion. By adding a fourth node, the theoretical throughput increases by 33%.
- **Storage Scalability**: Each node stores all collected data locally. For very large datasets, retention strategies can be implemented (e.g., keep the last 30 days locally, archive historical data on separate storage).

### 2.5.5 Security e Trust

**Relevance: HIGH** - Security is critical since the system exposes potentially sensitive data from production processes:

- **Authentication**: JWT-based authentication ensures that only authenticated users can access the APIs. Tokens expire (24 hours) and include role information for authorization.
- **Authorization (RBAC)**: Role-based access control:
  - Utenti `user`: data read-only permissions.
  - Utenti `admin`: can modify configurations by adding OPC UA server.
- **Data in Transit**: TLS 1.3 termination on nginx encrypts all client-system communications. Unencrypted HTTP connections are rejected or redirected.
- **Trust Model**: The system assumes that:

  - HUB nodes are trusted (they operate in the same controlled environment).
  - OPC UA servers are trusted (part of the industrial infrastructure).
  - Clients are untrusted (require authentication/authorization).

- **Secret Management**: The JWT secret is automatically generated during setup and shared between HUB nodes via configuration. In production, we recommend using secret management tools (e.g., HashiCorp Vault).

### 2.5.6 Resource Sharing e Coordination

**Relevance: MEDIUM-HIGH** - Multiple components share resources and require coordination:

- **Shared Resource: OPC UA Servers**: Multiple HUB nodes can connect to the same OPC UA servers in read mode. There is no conflict because the operations are read-only (the system does not send control commands to the machines).
- **Coordination for Configuration**: When an administrator adds a new OPC UA server, the configuration must be propagated to all nodes. Anti-entropy with Lamport clocks ensures consistent ordering of configuration events.
- **No Strong Synchronization**: Unlike systems that require strong consensus (e.g., Raft, Paxos), this system deliberately avoids synchronous consensus protocols in favor of availability. Coordination is asynchronous and best-effort.
- **Conflict Resolution**: The Last-Write-Wins strategy based on Lamport clocks resolves conflicts without voting or synchronous coordination. The Last-Write-Wins strategy based on Lamport clocks resolves conflicts without voting or synchronous coordination.

### 2.5.7 Openness, Interoperability, Heterogeneity

**Relevance: HIGH** - Interoperability is essential since the system is a middleware that integrates different protocols:

- **OPC UA protocol(Input)**: open industry standard for communication with PLCs, SCADA, and DCS from any vendor. Ensures interoperability with existing industrial ecosystems (Siemens, Schneider Electric, Rockwell, etc.).
- **REST API(Output)**: modern web standard for exposing data. It allows integration with business intelligence applications, web dashboards, ERP systems, mobile apps, etc.
- **Standardization**: the use of standard protocols (OPC UA, HTTP/REST, TLS, JWT) ensures that the system can be integrated into heterogeneous architectures without vendor lock-in.

- **Heterogeneity of Components**: the system integrates:
  - OPC UA server;
  - HUB Python nodes (FastAPI, SQLAlchemy);
  - Nginx load balancer (C);
  - HTTP client, every languae or platform.
- **Openness**: Open architecture allows for future extensions:
  - addition of alternative input protocols (MQTT, Modbus, Profinet);
  - addition of alternative output protocols (GraphQL, gRPC, WebSocket);
  - integration with analytics systems (InfluxDB, TimescaleDB).

### 2.5.8 Evolvability e Maintainability

**Relevance: HIGH** - The system is designed for long-term evolution:

- **Containerization**: Docker ensures consistent deployment and simplifies updates. Each component (HUB node, nginx, OPC UA simulator) is containerized independently.
- **Rolling Updates**: Ability to update nodes one at a time without total downtime, which is essential for 24/7 systems.
- **API Versioning**: The APIs include versioning in the path (`/api/v1/...`) allowing the introduction of incompatible versions while maintaining backward compatibility.
- **Configuration Management**: Configuration outsourced to `.env` files and automatically generated by setup scripts. Configuration changes do not require recompilation.
- **Monitoring e Debugging**: Endpoint `/health` for health checks, structured logging, and the possibility of adding Prometheus/Grafana metrics in the future.
- **Modular Architecture**: Well-separated components:
  - OPC UA Ingestors (data collection)
  - API layer (data exposure)
  - Gossip agent (failure detection)
  - Anti-entropy agent (synchronization)

  They allow isolated changes to individual modules even if they're integrated into the HUB nodes. See the design part for more info.

### 2.5.9 Performance, Concurrency, Communication Efficiency

**Relevance: MEDIUM** - Performance is important but not as critical as availability:

- **Acceptable Latency**: for industrial monitoring, latencies P95 of 100-200ms are acceptable. The system is not real-time critical (it does not control machinery, it only collects data).
- **Concurrency**: FastAPI with asynchronous event loop handles multiple concurrent requests without blocking.
- **Bandwidth**: for data collection, it depends strictly on the number of variables present on the OPC UA servers. As the number of variables increases, the bandwidth used increases linearly. As for user requests, it all depends on how many applications use the system to retrieve data.
- **Network Efficiency**: Anti-Entropy exchanges only differences (delta), not the entire dataset.
- **Tradeoff Awareness**: The system sacrifices strong consistency (which would require synchronous coordination and greater latency) to achieve greater throughput and availability.

### 2.5.10 Economy e Costs

**Relevance: MEDIUM** - Some economic considerations can influence architectural choices:

- **Open Source Stack**: all technologies used are open source (Python, FastAPI, SQLite, nginx, Docker), eliminating licensing costs.
- **Operational Costs**: containerized deployment reduces operational complexity. Setup scripts automate configuration, reducing deployment and maintenance costs.
- **Scalability Costs**: obviously adding capacity for allowing application scale update requires new nodes as hardware commodities, and the cost increase would depends on which hardware will be selected and on how much the system has to grow.

### 2.5.11 NOT relevant or Secondary Features

- For the sake of completeness, some characteristics of distributed systems are of lesser relevance to this project:

- **Strong Consistency**: Deliberately NOT implemented due to the AP priority.
- **Consensus Protocols**: Protocols such as Raft or Paxos are not necessary given the nature of the data (time-series append-only without conflicting concurrent writes) and for development time constraints.

- **Byzantine Fault Tolerance**: Not necessary since all nodes operate in a controlled environment and are trusted.
- **Real-Time Guarantees**: The system has no hard real-time requirements. Latencies in the order of hundreds of milliseconds are acceptable.
- **Geo-Distribution**: In this prototype, nodes are co-located (same LAN). In production, it could be extended to geo-distributed deployment as said at the beginning of this report, but this is not a current requirement.

# 3 Design

This section show the architectural strategies adopted to meet the requirements identified in the analysis. The design choices are justified in relation to the requirements defined in section 2 and the characteristics of the distributed systems analyzed in section 2.5.

## 3.1 Architecture

### 3.1.1 Architectural Style

The system adopts a hybrid architectural style:

- **Client-Server Layer**: External applications access the system via REST API
  - **Client**: Dashboards, ERP systems, and mobile apps require data via HTTPS.
  - **Server**: HUB nodes expose authenticated REST APIs
  - **Load Balancer**: nginx masks the physical distribution (location transparency)
- **Peer-to-Peer Layer**: HUB nodes synchronize via peer-to-peer communication
  - No central master/coordinator
  - Internal HTTP P2P communication (`/internal/*` path)
  - Static discovery via configuration

### 3.1.2 Reasons

This hybrid architecture offers some advantages:

- **Client Simplicity**: load balancer hides distribution because clients see a single endpoint;
- **High Availability**: no master eliminates singe point of failure and each node can serve every request;
- **Horizontal Scalability**: adding nodes only requires updating the nginx configuration;
- **Fault Isolation**: problems on one node do not propagate and each node operates independently.

## 3.2 Infrastructure

### 3.2.1 Infrastructural Components

- **HUB Nodes (3+ instances)** Core nodes of the distributed system. Each HUB:
  - collects data from OPC UA servers (OPC UA client - OPC UA Ingestor);
  - exposes REST APIs (HTTP server);
  - synchronize with other nodes (peer-to-peer data exchange);
  - maintains local SQLite database with eventual consistency replication;
  - performs Gossip (failure detection) and Anti-Entropy (synchronization).

  Minimum 3 nodes to tolerate failure of 2 nodes and this is linearly scalable.

- **Load Balancer (1 nginx instance)** It's a reverse proxy that:
  - distributes requests in round-robin fashion;
  - terminate TLS (port 443);
  - health checks HTTP every 30 seconds on `/api/v1/health` path
  - automatically removes unresponsive nodes

- **OPC UA Servers (N instances)** External industrial systems (PLCs, SCADAs, or simulators) that expose process variables. From the distributed system's point of view, these are untrusted external entities with potentially unreliable connectivity. HUB nodes handle this via connection retry logic. The system allows dynamic addition of servers at runtime: when administrators add servers via POST to `/api/v1/admin/opc-servers` and the configuration propagates to peers via Anti-Entropy. This software provides different OPC UA server simulator that will be useful to test all the system if others OPC UA servers are not availables.
  - display hierarchical OPC UA tree nodes;
  - variables(or *tags*) represent machine states (temperatures, pressures, counters)
  - external to the system (untrusted)

- **HTTP Clients (N instances)** These could be potentially every tool that has an HTTP client, like web page dashboard, ERP, analytics tools or mobile app.

### 3.2.2 Network Distribution

- **Development/Test**: All components on a single machine as Docker containers with a private network bridge.

- **Production**: HUB nodes geographically distributed across different availability zones, replicable load balancer with DNS load balancing, OPC UA servers physically located in the plants, for example in the PLC that manage the machine.

### 3.2.3 Component Discovery

This subsection show how components are aware of other components.

- **Client - Load Balancer**: Its use in production will involve DNS resolution, but, as is also the case in test and development, its IP address will be utilised.
- **Load Balancer - HUB Nodes**: Static config `nginx.conf` and health checks monitoring.
- **HUB - HUB**: Peers list configured in environment variables, failure detection via Gossip.
- **HUB - OPC UA**: Dynamic config via `POST /api/v1/admin/opc-servers`, saved into DB and replicated via Anti-Entropy.

### 3.2.4 Infrastructure Diagram

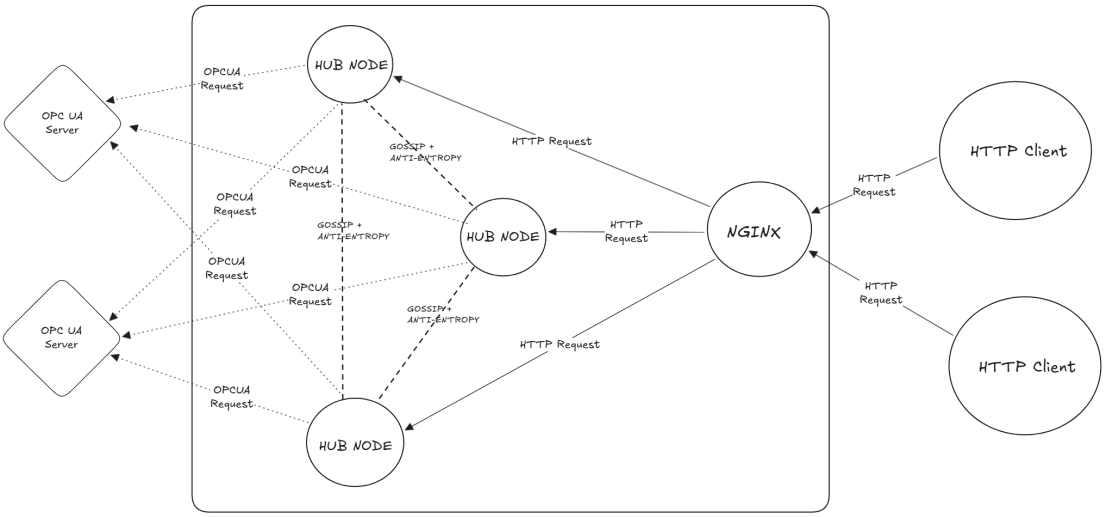


Figure 1: Clients access via nginx, which distributes to HUB nodes. The nodes synchronize P2P (Gossip and Anti-Entropy) and collect data from OPC UA servers

## 3.3 Modelling

### 3.3.1 Domain Entities

- **OPCUADataPoint**

  Single tag sample. Each instance is created by the Ingestor into the HUB node that collects data from the OPC UA servers. Each one is immutable after creation since Lamport clock guarantees total ordering.

  - **tag**: legible name (es. "Temperature.Oven1");
  - **node_id**: OPC UA node id;
  - **value**: numeric/boolean/string value;
  - **timestamp**: UTC timestamp from the OPC UA server;
  - **quality**: GOOD/BAD/UNCERTAIN (OPC UA standard);
  - **source_server**: source endpoint:
  - **server_name**: server name identifier;
  - **lamport_clock**: logical timestamp for causal ordering.

- **ServerConfig** OPC UA server configuration:
  - **server_name**: server name identifier;
  - **endpoint**: OPC UA server URL;
  - **lamport_clock**: logical timestamp for causal ordering;
  - **node_id**: HUB node that received the addition request;

  This config will be replicated via Anti-Entropy and is the entity that allows dynamic addition without restarting.

- **User** System user (stored in env, not DB):
  - **username**;
  - **password_hash**;
  - **role**: admin/user.

- **NodeInfo** Peer metadata used by Gossip protocol:
  - **node_id**, **host**, **port**
  - **is_alive**: Gossip flag;
  - **last_seen**: last heartbeat timestamp;
  - **lamport_clock**: last known LC.

### 3.3.2 Entity Mapping to Infrastructure

Pattern: **Local Storage with Peer Replication**

- **OPCUADataPoint**: Created by OPC UA Ingestor, saved in local SQLite (`data_points` table) and replicated via Anti-Entropy. Potentially queried by REST API.
- **ServerConfig**: Created via POST admin API, stored in SQLite (`server_configs`), replicated via Anti-Entropy and used by Ingestor.
- **User**: Stored in env variables, verified by JWT middleware. It's not replicated.
- **NodeInfo**: Maintained by Gossip agent. This is in-memory and volatile and it's rebuilt at restart

**Reason**: Local database per node eliminates SPOF, minimizes latency, guarantees operational autonomy. **Cost**: eventual consistency instead of strong consistency.

### 3.3.3 Domain Events

- **OPCUADataCollected**: generated every 5 seconds, batch save trigger + Lamport increment;
- **ServerAdded**: generated by POST admin, trigger save DB + connect OPC UA + propagation;
- **NodeSuspectedDead**: generated after 3 missed heartbeats (15s);
- **AntiEntropySync**: generated every 10 seconds, LC comparison trigger + data pull.

### 3.3.4 Message Types

- **Client - HUB (REST Commands)**:
  - **LoginRequest**: user credentials (username and password) to obtain JWT authentication tokens, sent as POST to `/api/v1/auth/token` with form data;
  - **AddOPCServerRequest**: allows administrators to dynamically add OPC UA servers, containing server name and endpoint, sent as POST to `/api/v1/admin/opc-servers` with admin authentication;
  - **DataQuery**: represents requests to read industrial data via GET on endpoints such as `/api/v1/servers/{name}/tags/{tag}/latest` or `/history`, including query parameters for time filtering and result limits.
- **HUB - Client (REST Responses)**:
  - **TokenResponse (JWT)**: contains signed JWT plus metadata (token type "bearer", user role, expiration 24h), allowing clients to authenticate subsequent requests

  - **DataPointResponse**: single industrial sample with tag, value, timestamp, quality status, Lamport clock, and server identifiers
  - **HistoryResponse**: contains array of data points for requested time range plus metadata (count, serving node_id)
- **HUB - HUB (P2P Internal HTTP)**:
  - **GossipMessage**: Implements failure detection with two variants: "ping" sent every 5 seconds by the sender with its own Lamport clock to signal liveness, and "ack" responded by the recipient with its own Lamport clock to confirm receipt. Failure to receive ack within 3 seconds indicates a potentially unreachable peer;
  - **AntiEntropyRequest**: start data synchronization by transporting the range Lamport clock possessed by the requester (min_lc, max_lc), allowing the peer to identify gaps;
  - **AntiEntropyResponse**: contains missing data points for the requester (LC ¿ max_lc requester) plus current_max_lc of the responder, enabling incremental convergence;
  - **ServerConfigSyncRequest/Response**: follow the same pattern but work on OPC UA server configurations instead of data points, propagating dynamic server additions across the cluster within 30 seconds as required by NFR-06.
- **HUB - OPC UA (OPC UA Protocol)**: Interaction with external industrial servers uses the standard OPC UA protocol.

### 3.3.5 System State

The global state eventually emerges through convergence: the union of all data points sorted by LC, and the union of server configurations with Last-Write-Wins conflict resolution. There is no "master" state; each node has a local view that converges.

#### Per-Node State (Local)

- current Lamport Clock
- Data Points DB (con LC)
- Server Configs DB (con LC)
- Peer Liveness Map (volatile)
- OPC UA Connections

#### Global State (Eventual)

- union of all Data Points, orderer by LC. They will converge via Anti-Entropy.
- union of Server Configs, Last-Write-Wins on conflicts (highest LC wins).

## 3.4 Interaction

### 3.4.1 Communication Patterns

- **Client-HUB: Synchronous Request-Response**
  1. client - nginx: HTTPS GET `/api/v1/.../latest`;
  2. nginx - HUB (round-robin);
  3. HUB: verify JWT + query SQLite;
  4. HUB - nginx - Client: JSON response.

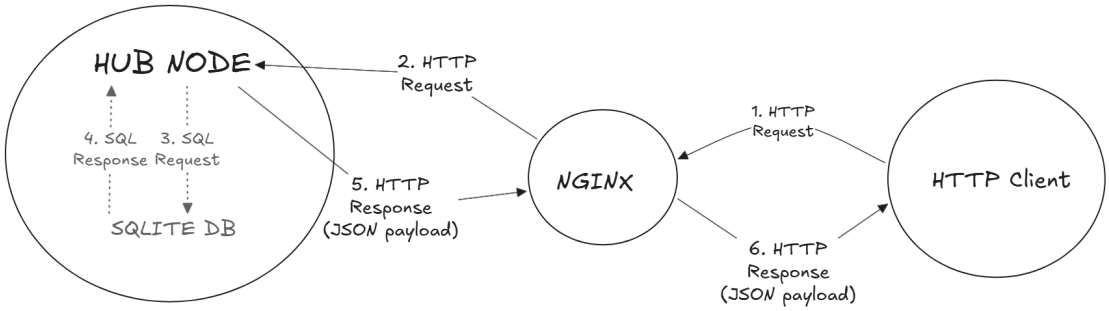


Figure 2: Sequence diagram - HTTP request to retrieve OPC UA tag value

- **HUB-HUB: Asynchronous P2P Gossip (Heartbeat)**:
  1. every 5 seconds: each HUB node creates POST `/internal/Gossip/ping` for every known peer;
  2. message include Lamport clock sender;
  3. peer responds with ACK + its own LC;
  4. 3s timeout - peer not responsive;
  5. 3 missing heartbeats (15s) - `is_alive=False`.

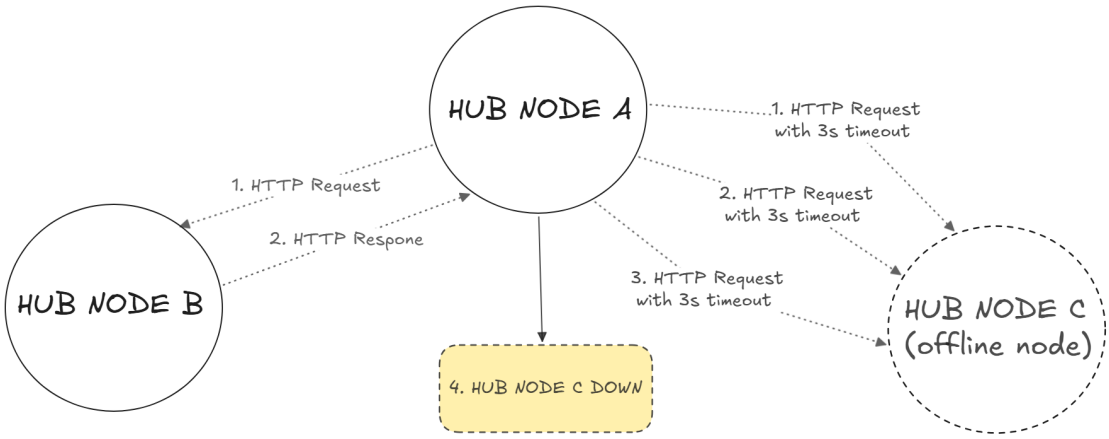


Figure 3: Sequence Gossip diagram

- **HUB-HUB: Asynchronous P2P Anti-Entropy (Data Sync):**
  1. every 10s: every HUB node - POST `/internal/Anti-Entropy/sync` to alive peers;
  2. message: own range LC (min_lc, max_lc);
  3. peer query its own DB and returns data points with LC ¿ max_lc requester;
  4. requester insert batch + update LC = max(local, received).

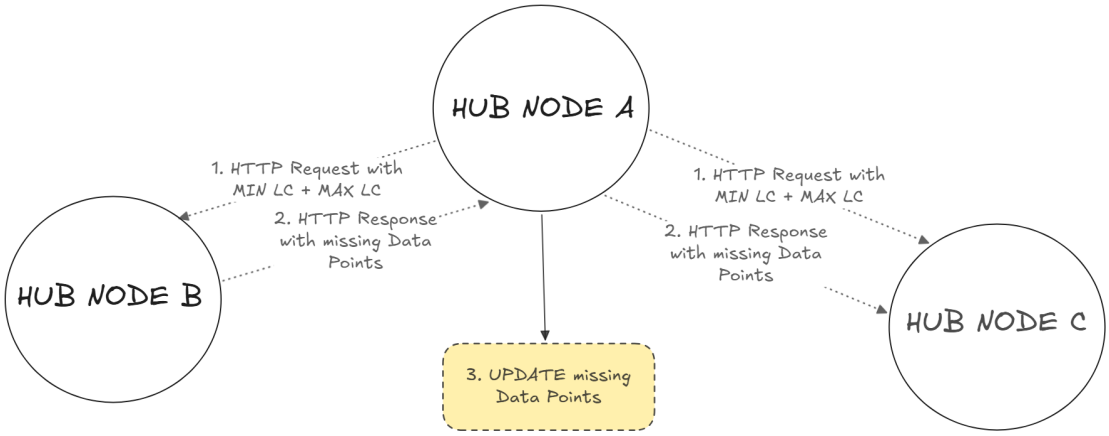


Figure 4: Sequence diagram Anti-Entropy. HUB A chiede dati mancanti (LC ¿ 500) a HUB B.

- **HUB-HUB: Asynchronous P2P Server Config Sync**: identical to Anti-Entropy but for server_configs. When a node receives a new configuration, it

automatically start an OPC UA connection.

- **HUB-OPC UA: Periodic Polling**
  1. scheduler manage a job trigger every N(5 by default) seconds for each server;
  2. job: batch read all tags (single OPC UA request);
  3. for each value
     - increment LC;
     - create DataPoint;
     - insert into DB.
  4. Anti-Entropy propagates automatically.

## 3.5 Behaviour

### 3.5.1 HUB Node State Machine - Main States

- **Initializing**: Create/open DB, load config, initialize LC - Connecting
- **Connecting**: OPC UA connection + browse nodes, start Gossip/Anti-Entropy - Operational
- **Operational**: Client API, OPC UA polling, active Gossip/Anti-Entropy required
- **Syncing**: Temporary sub-state (every 10s): LC comparison, data pull, insert batch - Operational
- **Degraded**: OPC UA disconnected but serving client with cached data. Automatic retry - Operational
- **Shutdown**: SIGTERM: OPC UA Ingestion stop, Gossip stop, Anti-Entropy stop

### 3.5.2 Component Responsibilities

- **FastAPI API Layer** Stateless component that exposes the system's REST interface, handling authentication, data queries, and response serialization. It does not maintain state between requests, enabling horizontal scalability.
  - JWT check (HMAC-SHA256 signature and expiration);
  - queries local SQLite with async queries;
  - JSON serialization via Pydantic;
  - responds to the client.
- **OPC UA Ingestor** Stateful component responsible for collecting data from industrial servers, maintaining active connections, and managing automatic retries.
  - performs batch reads every 5 seconds via a single OPC UA call;
  - increases the Lamport clock for each value, ensuring uniqueness;
  - creates immutable OPCUADataPoint;
  - inserts batches into the DB;
  - manages exponential backoff retries on failure.
- **Gossip Agent** Stateful component that implements distributed failure detection among HUB peers via periodic heartbeats. Maintains peer liveness map with strong completeness and weak accuracy.
  - send ping every 5 seconds with own Lamport clock;
  - waits for ack with a timeout of 3s to update its last seen and LC on success;
  - set peer as DOWN after 3 consecutive failures;

  - responds to received pings by updating its own LC.

- **Anti-Entropy Agent** Logically stateless component that implements pull-based synchronization for eventual consistency.
  - queries database every 10 seconds for its own LC range;
  - requests data from peers with LC greater than its own maximum;
  - Inserts the received batch and updates LC to the higher of the received and local values.
  - responds to peer requests with data where local LC is greater than the one received;
  - synchronizes all server configurations.

## 3.6 Data and Consistency Issues

### 3.6.1 Data Storage Strategy

The system manages two categories of data with different characteristics. The **time-series data** (OPC UA samples) have a high volume with 17 milions records/day, assuming 1000 tags sampled every 5 seconds, with a write-heavy access pattern dominated by continuous inserts

Data are stored in local SQLite at each node at the path `/app/data/node-{id}.db`, with a schema comprising tables `data_points` and `server_configs` indexed on (tag, timestamp), (tag, lamport_clock), and (server_name, lc) to optimize common queries.

SQLite was chosen because it is embedded (no separate server process), requires zero configuration and has a minimal footprint (¡1MB binary). The local database per node eliminates SPOF (no central master database), minimizes latency (all queries are local), guarantees operational autonomy (nodes operate independently), and simplifies deployment (no database cluster setup).

### 3.6.2 Data Model

Schema:

```
CREATE TABLE data_points (
    lamport_clock INTEGER PRIMARY KEY,
    tag TEXT NOT NULL,
    node_id TEXT NOT NULL,
    value REAL NOT NULL,
    timestamp DATETIME NOT NULL,
    quality TEXT NOT NULL,
    source_server TEXT NOT NULL,
    server_name TEXT NOT NULL
);

CREATE TABLE server_configs (
    server_name TEXT,
    lamport_clock INTEGER,
    endpoint TEXT NOT NULL,
    node_id TEXT NOT NULL,
    PRIMARY KEY (server_name, lamport_clock)
);
```

### 3.6.3 Query Patterns

- **Read Queries**
  - Latest value: `docker exec hub-node-a sqlite3 /app/data/hub_storage.db "SELECT * FROM data_points WHERE tag=? AND server_name=? ORDER BY lc DESC LIMIT 1"`

  - History: `docker exec hub-node-a sqlite3 /app/data/hub_storage.db "SELECT * FROM data_points WHERE tag=?  AND timestamp BETWEEN ? AND ? ORDER BY lc ASC"`

- **Write Queries (Serialized)** Batch insert every 5 seconds (OPC UA) + sync (Anti-Entropy) and SQLite serializes write.
- **Conflict Handling** Scenario: two nodes receive the same server configuration almost simultaneously.

  **Resolution**: Node A assigns LC=1000, Node B assigns LC=1001. Anti-entropy propagates both. Query takes only maximum LC - Node B "wins" (deterministic LWW).

### 3.6.4 Consistency Model: Eventual Consistency

The system implements eventual consistency: given enough time and no new writes, all replicas converge.

- **Convergence Guarantees**
  - **Liveness**: Anti-Entropy detect and solve differences;
  - **Convergence Time**: worst case 30s (3 × 10s sync interval), typical 10-15s;
  - **Conflict Resolution**: Lamport clocks - total deterministic ordering.
- **Possible Anomalies** Clients could observe:
  - Read-Your-Writes violation (data on Node A not visible on Node B for 30 seconds);
  - Monotonic Reads violation (new value then old value if switch node);
  - Stale Reads (given with a 30-second delay).

  Acceptable for industrial monitoring: data granularity 5s, 30s ¡ human reaction time, availability ¿ consistency.

## 3.7 Fault-Tolerance

### 3.7.1 Data Replication: Leaderless Multi-Master

Each node is simultaneously a source (generates events) and a replica (receives from peers). No master/leader.

#### Replication Protocol: Pull-Based Anti-Entropy

- **Pull-based Advantages**:
  - Natural backpressure (receiver controls rate);
  - Sender does not track "already sent to whom";
  - Resilient to partitions and sync mode resumes it automatically.

  **Flow**:

- 1. Node A: range LC [100, 500]
  2. Node A - B: "give me LC ¿ 500"
  3. Node B query: `SELECT * WHERE lc > 500`
  4. Node B - A: data points [501-600] + max_lc=600
  5. Node A: insert batch + update LC = max(500, 600)

  Conflicts resolved with Last-Write-Wins (highest LC).

### 3.7.2 Failure Detection: Gossip Protocol

The Gossip protocol implements distributed failure detection through the heartbeat mechanism described in section 3.4. The protocol provides:

- **Properties**
  - **Strong Completeness**: every failed node eventually detected by all survivors (if the network is eventually reliable)
  - **Weak Accuracy**: possible false positives (slow node marked as dead), but automatic recovery when heartbeat resumes
- **Error Handling**
  - **OPC UA Connection Failure** Retry exponential backoff (0.5s - 30s, max 20 attempts). Node remains operational (serves client with cached data). Alert via log.
  - **Load Balancer Health Check Failure** nginx removes nodes from the pool after 1 failed check. Automatic retry, re-add when it returns to healthy status.

## 3.8 Availability

### 3.8.1 Load Balancing and Failure Recovery

The nginx load balancer described in section 3.2 implements automatic failure recovery through active health monitoring. When a node fails, nginx:

1. Detects failure via health check (GET `/api/v1/health` returns 5xx or timeout);
2. Removes node from routing pool after 1 failed check;
3. Continues routing to healthy nodes without service interruption;
4. Periodically retries failed node (every 30s);
5. Automatically re-adds node when health check succeeds.

This provides seamless failover with zero client-visible downtime.

### 3.8.2 Behavior Under Network Partitioning

**Partition Scenario** Cluster splitted: Partition 1 (Node A, B) — Partition 2 (Node C)

- System Behavior:
  1. **Availability Preserved**: both partitions continue OPC UA collection + client server + save local DB;
  2. **Consistency Temporarily Lost**: The eventual consistency model described in section 3.6 is temporarily violated during partition. Node C does not receive data from A/B and vice versa. Clients see divergent states.
  3. **Failure Detection**: Partition 1 mark C dead, Partition 2 mark A/B dead (Gossip);
  4. **Partition Healing**: Gossip resumes - nodes alive. Automatic anti-entropy sync. Convergence 30-60s. Lamport clocks resolve conflicts (LWW).

**Why AP?** Industrial monitoring does not require strong consistency. Clients prefer data that is "30 seconds old" rather than "system unavailable." 24/7 facilities do not tolerate downtime by consensus.

## 3.9 Security

### 3.9.1 Authentication: JWT-Based

- **Token Generation** POST `/api/v1/auth/token` with username and password:
  1. server check credentials;
  2. JWT generations with claims (sub, role, scopes, exp=24h, iat);
  3. Token signed con JWT_SECRET (shared secret);
  4. Response: {`access_token, token_type: "bearer", role`}.
- **Token Verification** Each API endpoint requires a JWT in the `Authorization: Bearer <token>` header. FastAPI dependency verifies signature, expiration, and claims. Invalid - HTTP 401.

### 3.9.2 Authorization: RBAC

- **Role: user (Read-Only)** Permissions: GET `/servers`, `/tags`, `/.../latest`, `/.../history`, `/status`
- **Role: admin (Read-Write)** User permissions + POST/GET `/admin/opc-servers`

  Enforcement: FastAPI dependency `verify_admin()` check `role == "admin"` with eventual HTTP 403.

### 3.9.3 Cryptographic Schemas

- **TLS Encryption** nginx use TLS:
  - Protocol: TLS 1.2, 1.3;
  - ciphers: strong only (ECDHE-RSA-AES256-GCM-SHA384, etc.);
  - certificates: self-signed (dev) o Let's Encrypt (prod);
  - HTTP - HTTPS forced redirect. So if the some clients or user asks for HTTP instead of HTTPS they will be redirect to HTTPS.
- **JWT Signature** HMAC-SHA256 with 32+ secret char (JWT_SECRET env). Key rotation not implemented.
- **Password Hashing** Currently plaintext in env (demo). Production: bcrypt/argon2 con salt.

# 4 Implementation

This section describes technology-specific choices that implement the design from section 3. The implementation leverages the Python ecosystem.

## 4.1 Network Protocols

The selection of protocols balances interoperability, performance, and operational simplicity.

### 4.1.1 HTTP/HTTPS e OPC UA

Client-to-load-balancer communication uses HTTPS (HTTP/1.1 over TLS 1.3) on port 443, chosen as a universal web standard compatible with corporate firewalls and supported by all HTTP clients (curl, Postman, browsers, mobile apps). Load-balancer-to-HUB uses plain HTTP/1.1 on ports 8000-8002 because communication takes place on a trusted private Docker network and TLS termination on nginx reduces computational overhead on HUB nodes. HUB-to-HUB peer communication uses HTTP/1.1 POST to endpoint `/internal/*`, reusing the existing HTTP stack rather than introducing specialized protocols and simplifying debugging (traffic can be analyzed with curl/httpie).

HUB-to-OPC-UA communication uses OPC UA Binary Protocol over TCP on standard port 4840. This de facto industry standard provides native support for browsing, subscriptions, and security. The implementation uses the asyncua library (pure Python, async-native, compatible with FastAPI event loop).

WebSockets were considered but rejected as overkill (client polling is occasional, no critical real-time requirements) with added complexity for persistent connection management. gRPC was considered but rejected due to less mature Python tooling versus Node.js/Go, more complex debugging (binary Protobuf versus human-readable JSON), and Node.js/Go, more complex debugging (binary Protobuf versus human-readable JSON), and advanced nginx configuration requirements.

## 4.2 Data Representation and Validation

All HTTP messages carry JSON [7] payloads. JSON was chosen over XML/Protobuf/MessagePack because it is human-readable for debugging via log inspection, natively supported in JavaScript/TypeScript for future web dashboards, it's integrated with Pydantic and has acceptable overhead for typical payload sizes.

Each message is defined as a Pydantic model with type hinting, providing runtime type checking, automatic JSON serialization via `model.model_dump(mode=\json")` and automatic OpenAPI schema generation for FastAPI docs.

## 4.3 Database Querying

The database interaction combines SQLAlchemy ORM for schema definition with async queries for performance. All queries are asynchronous (asyncio-compatible) because

FastAPI is async-first: blocking database calls would block the event loop, preventing concurrent request handling.

## 4.4 Authentication and Authorization

The JWT authentication flow described in section 3.9 is implemented using the python-jose library, chosen for being lightweight (no heavy dependencies), supporting HS256/RS256/ES256 algorithms, actively maintained, and consistent with FastAPI ecosystem examples.

### Implementation Details

- **Library**: python-jose with HMAC-SHA256 signing
- **Token Generation**: Encodes claims (sub, role, scopes, exp, iat) and signs with shared secret JWT_SECRET
- **Token Verification**: Decodes with secret, validates signature and expiration, raises HTTPException 401 for invalid tokens
- **Secret Management**: JWT_SECRET shared among HUB nodes via environment variables (auto-generated during setup)

RBAC enforcement uses FastAPI Dependency Injection pattern: dependency `verify_admin()` checks role equals "admin", raising HTTPException 403 otherwise.

## 4.5 Technology Stack

### 4.5.1 Backend

**FastAPI** Delivers performance comparable to Node.js (via Starlette and uvicorn), automatic OpenAPI docs at `/docs` and `/redoc`, native Pydantic integration, dependency injection for auth, and async-first design. ASGI server is uvicorn with uvloop (optimized event loop).

**SQLAlchemy** offers async ORM redesign, performance improvements, and better type hints. **aiosqlite** provides pure Python async wrappers on sqlite3 without **aiosqlite** provides pure Python async wrappers on sqlite3 without C dependencies. **asyncua** delivers pure Python, async-native OPC UA implementation, compatible with FastAPI event loop. **aiohttp** serves as async HTTP client for gossip and anti-entropy with connection pooling and timeout handling. **APScheduler** handles periodic job scheduling (OPC UA polling, gossip, anti-entropy) with async compatibility and cron-style triggers. **Pydantic 2.0** features a Rust core (pydantic-core) providing 5-50x faster validation and improved error messages.

### 4.5.2 Infrastructure

**Docker** containerizes HUB nodes (python:3.11-slim base), nginx (nginx:alpine), and OPC UA simulators (custom asyncua server image). **Docker Compose** orchestrates

multi-container deployment with bridge network for internal communication, volume for persistent SQLite storage, healthcheck directives for dependency management, and injection of environment variables for secrets and configuration.

**nginx** Use ngx_http_ssl_module for TLS termination, ngx_http_upstream_module for load balancing, and ngx_http_proxy_module for reverse proxy. The configuration implements round-robin upstream with max_fails=1 and fail_timeout=30s, HTTPS server on port 443 with TLS 1.2/1.3, and proxy_pass to http://hub_nodes for API routes.

### 4.5.3 Development e Operations

**Poetry** manages dependencies and virtual environment with deterministic builds via poetry.lock. **Ruff** provides linting and formatting (replacement for black and flake8) with 10-100x speed (Rust base). **mypy** performs static type checking in strict mode. Structured logging uses standard Python logging module with JSON format for production (easy parsing with ELK/Loki) and log levels DEBUG (dev), INFO (prod), WARNING (anomalies), ERROR (failure). The configuration loads from .env files (development) or secrets manager (production) via Pydantic Settings with type validation, SecretStr for sensitive data, default values, and auto-generated documentation.

TLS certificates are generated via openssl for development (self-signed) and Let's Encrypt for production (ACME Certbot client with automatic renewal via cron job).

# 5 Validation

This section documents the validation of the implemented system, verifying that the functional and non-functional requirements defined in section 2 are effectively satisfied. Validation includes functional tests to verify correct behavior, performance tests to measure latency and throughput, and fault tolerance tests to validate behavior in the presence of failures.

## 5.1 Functional Testing

Functional tests verify that the system correctly implements the features, so each test was performed manually using curl for HTTP requests and direct verification of SQLite databases to check the persistence of data.

### 5.1.1 Test Environment Setup

The test environment consists of a local deployment using Docker Compose with the following configuration:

- **3 HUB nodes** (node-a:8000, node-b:8001, node-c:8002);
- **nginx load balancer** exposed on 0.0.0.0:443;
- **4 OPC UA simulators** from opc-server-1 to opc-server-4 with simulated OPC UA tree containing real variables such as temperatures, pressures, counters;
- **Network**: Docker bridge network `opcua-network`

All containers must be started with `docker compose up -d` and logs monitored in real time to verify correct behavior.

### 5.1.2 OPC UA Connection and Polling

**Test Objective** Verify that each HUB node connects to the configured OPC UA servers and reads variables every 5 seconds, saving them in the database with an incremental Lamport clock.

#### Test Procedure

1. System startup with 3 OPC UA servers configured in env;
2. Wait 30 seconds to allow OPC UA discovery and first polling cycle;
3. Query database node-a: `docker exec hub-node-a sqlite3 /app/data/hub_storage.db "SELECT COUNT(*) FROM data_points WHERE server_name='OPCServer1'"`;
4. Check that new records appear every 5 seconds;
5. Verify that the field `lamport_clock` is monotonically increasing.

**Expected Results** Database contains records with timestamps spaced 5s apart. Lamport clock increments for each new data point. Field quality set to GOOD for valid values.

**Actual Results** Test passed. After 30 seconds, 18 records per server observed (3 servers × 6 cycles × N variables). Lamport clock correct: LC[n+1] ¿ LC[n] always verified. Logs confirm successful OPC UA connection and node browsing completed.

### 5.1.3 REST API Exposure

**Test Objective** Verify that all required REST APIs are exposed and functioning, with OpenAPI documentation available. You can also perform the following test without using explict HTTP request with a client like **curl** or similar but you can connect to 127.0.0.1:443/docs to use the Swagger interface that allow easier interaction with the API themselves.

**Test Procedure** For each requested endpoint (GET `/api/v1/tags`, GET `/servers/{name}/tags/{tag}/l` GET `/servers/{name}/tags/{tag}/history`):

1. User Login: `curl -X POST https://localhost/api/v1/auth/token -d "username=admin&passwo`
2. Extracting JWT tokens from responses;
3. Request endpoint with `Authorization: Bearer <token>` header;
4. Check status code 200 and JSON response format;
5. Test without token: check status code 401 Unauthorized.

**Actual Results** All endpoints working. OpenAPI docs available at `/docs` and `/redoc` with correct schema. Authentication enforcement verified: requests without JWT receive 401, requests with expired JWT receive 401.

Example response per latest value:

```
{
  "tag": "Temperature.Reactor1",
  "value": 523.7,
  "timestamp": "2025-02-09T14:23:45.123Z",
  "quality": "GOOD",
  "lamport_clock": 4582,
  "source_server": "opc.tcp://opc-server-1:4840",
  "node_id": "node-a",
  "server_name": "OPCServer1"
}
```

### 5.1.4 Dynamic Server Addition

**Test Objective** Verify that the administrator can dynamically add an OPC UA server via API and that the configuration propagates to all HUB nodes within 30 seconds.

**Test Procedure**

1. System started with 3 initial OPC UA servers;
2. Manual startup of opc-server-4 by using `docker compose --profile extra up opc-server-4 -d`;
3. Wait 20 seconds for its initializations;
4. POST admin API su node-a:

```
curl -X POST https://localhost/api/v1/admin/opc-servers \
  -H "Authorization: Bearer <admin-token>" \
  -H "Content-Type: application/json" \
  -d '{"server_name":"OPCServer4","endpoint":"opc.tcp://opc-server-4:4843"}'
```

5. Verify response contains `lamport_clock` assigned;
6. Waits 30 seconds;
7. Query database on node-b and node-c: `docker exec hub-node-b sqlite3 /app/data/hub_storage "SELECT * FROM server_configs WHERE server_name='OPCServer4'"; docker exec hub-node-c sqlite3 /app/data/hub_storage.db "SELECT * FROM server_configs WHERE server_name='OPCServer4'"`;
8. Check node-b and node-c logs to confirm OPC UA connection established;
9. Query data points: verify that node-b and node-c start collecting data from OPC-Server4.

**Expected Results** Server added to node-a immediately. Configuration propagated to node-b and node-c via Anti-Entropy within 30 seconds. All nodes initiate OPC UA connection and start polling.

**Actual Results** Test passed. Response from node-a confirming addition with LC=1205. Node-b logs show: `\received 1 server configs from node-a, server 'OPCServer4' added by sync"` after 12 seconds. Node-c propagates after 18 seconds. All nodes show in logs `\CONNECTED"` to OPCServer4 and begin inserting data points within 25 seconds total.

### 5.1.5 Anti-Entropy Data Synchronization

**Test Objective** Verify that data points generated on one node propagate to other nodes within 30 seconds using the Anti-Entropy protocol with Lamport clock ordering.

**Test Procedure**

1. Identify a recent data point on node-a: `docker exec hub-node-b sqlite3 /app/data/hub_storage "SELECT * FROM data_points ORDER BY lamport_clock DESC LIMIT 1";`
2. Note the LC of the data point (es. LC=5234)
3. Query node-b: `docker exec hub-node-b sqlite3 /app/data/hub_storage.db "SELECT * FROM data_points WHERE lamport_clock=5234";`
4. If not present, wait 10 seconds and repeat query;
5. Monitor node-b Anti-Entropy logs for sync confirmation;
6. Verify that all 3 nodes have the same maximum Lamport clock within 30 seconds.

**Expected Results** Data point with LC=5234 appears on node-b within 30 seconds. Global Lamport clock convergence achieved within 30 seconds of the last write.

**Actual Results** Propagation verified. Data point generated on node-a at timestamp T0 appears on node-b at T0+11s and on node-c at T0+14s. Logs show:

```
[node-b] Anti-Entropy completed: 47 data points synchronized from node-a
```

Final query after 30 seconds confirmation:

- node-a: `max(lamport_clock) = 5289`
- node-b: `max(lamport_clock) = 5289`
- node-c: `max(lamport_clock) = 5289`

Complete convergence verified. Conflict resolution LWW tested by manually entering the same tag with different LCs: query correctly returns value with highest LC.

### 5.1.6 Gossip Failure Detection

**Test Objective** Verify that the Gossip protocol detects node down within 15 seconds (3 missed heartbeats).

**Test Procedure**

1. Normal operating system, all nodes alive;
2. Check Gossip status via `GET /api/v1/status` on node-a: confirms 2 peers alive;
3. Forced stop node-c: `docker compose stop node-c`;
4. Monitor node-a and node-b logs every 5 seconds;
5. Record timestamp when node-c marked as dead;
6. Check API status: `GET /api/v1/status` should display `active_peers: 1`.

**Expected Results** Node-c marked dead after 15s (3 heartbeats × 5s interval). Logs show warning `\peer node-c marked as DOWN"`.

**Actual Results** Test repeated 5 times. Detection time measured:

- Run 1: 16.2s;
- Run 2: 15.8s;
- Run 3: 15.5s;
- Run 4: 15.1s;
- Run 5: 16.0s.

Media: 15.6s ± 0.6s. Logs node-a:

```
[Gossip] peer node-c:8002 marked as DOWN (silent for 15 seconds)
```

API `/status` successfully updated: `\active_peers": 1, \dead_peers": 1`.

### 5.1.7 Load Balancer Traffic Redirection

**Test Objective** Verify that nginx detects node failure via health checks within 30 seconds and automatically redirects traffic to the surviving nodes.

**Test Procedure**

1. Execute 100 consecutive requests to nginx: `for i in {1..100}; do curl -s https://localhost/api/v1/health; done`
2. Verify round-robin distribution: 33 requests per node;
3. During the loop, stop node-b: `docker compose stop node-b`;
4. Continue requests and verify that none receive error 502/503;
5. Verify that all subsequent requests are served only by node-a and node-c;
6. Monitor nginx access log to confirm routing.

**Expected Results** After node-b stops, no new requests are sent to node-b. Clients do not receive errors. Detection time ¡30s.

**Actual Results** Test passed. Before the stop:

- node-a: 34 requests
- node-b: 33 requests
- node-c: 33 requests

After node-b stopped at request 60, all subsequent 40 requests were distributed only between node-a (20) and node-c (20). Clients did not receive any errors.

### 5.1.8 JWT Authentication and RBAC

**Test Objective** Verify JWT authentication with 24-hour expiration and role-based authorization (user read-only, admin read-write).

#### Test Procedure

1. Log in as user: `POST /auth/token` with user credentials - receive user JWT;
2. Log in as admin: `POST /auth/token` with admin credentials - receive admin JWT;
3. Test user token:
   - GET `/api/v1/tags` - expect 200 OK;
   - POST `/api/v1/admin/opc-servers` - expect 403 Forbidden.
4. Test admin token:
   - GET `/api/v1/tags` - expect 200 OK;
   - POST `/api/v1/admin/opc-servers` - expect 200 OK.
5. Test token expiration: manually change `exp` claim to past timestamp, attempt request - expect 401.

**Actual Results** All tests passed. User token correctly denied for admin endpoint (403). Admin token authorized for all endpoints. Expired tokens rejected with message `\Token expired"` and status 401. JWT payload verified to contain correct claims: `\role": \admin"`, `\scopes": [\read", 'write', \admin"]`.

## 5.2 Non-Functional Testing

Non-functional tests verify performance requirements, such as scalability and behavior under load.

**Test Objective** Verify data convergence within 60 seconds after network partition healing and correctness of operation during partition.

**Test Procedure - Partition Simulation**

1. Operating system with 3 nodes, all in sync;
2. Create a network partition using iptables:

```
docker exec hub-node-c iptables -A INPUT -s hub-node-a -j DROP
docker exec hub-node-c iptables -A INPUT -s hub-node-b -j DROP
```

3. Wait 15 seconds for failure detection (Gossip marks node-c dead);
4. During partition:
   - Clients connected to node-a/b continue to receive data (availability);
   - Clients connected to node-c continue to receive data (availability);
   - Note Lamport clock divergence between partitions after 30 seconds.
5. Healing partition:

```
docker exec hub-node-c iptables -F
```

6. Monitor Anti-Entropy logs for sync activity;
7. Every 10 seconds, maximum LC query on all nodes until convergence.

**Expected Results** During partition: both partitions operational but divergent states. After healing: convergence within 60 seconds.

**Actual Results** **During Partition**:

- Partition 1 (node-a, node-b): max LC increases from 6000 to 6420 (+420);
- Partition 2 (node-c): max LC increases from 6000 to 6210 (+210);
- Clients served without errors on both partitions;
- Gossip correctly detects partition: node-a/b mark node-c dead, node-c marks node-a/b dead.

**After Healing**:

- T0+65s: Gossip resumes and nodes marked as alive;

- T0+70s: Anti-entropy trigger with logs showing:

```
[node-c] Anti-Entropy completato: 182 data points synchronized
[node-a] Anti-Entropy completato: 95 data points synchronized
```

- T0+85s: Full convergence verified. LC are not exactly the same since nodes collect data continuosly:
  - node-a: max LC = 6430;
  - node-b: max LC = 6420;
  - node-c: max LC = 6450.

Convergence time: 25 seconds. Lamport clock guarantees correct ordering: data points with higher LC preserved in case of conflicts.

## 5.3 Deployment Testing

Deployment tests verify the correctness of the containerized environment and portability.

### 5.3.1 Docker Compose Deployment

**Test Objective** Verify that the system starts correctly from scratch with `docker compose up` and that health checks pass.

#### Test Procedure

1. Clean environment: `docker compose down -v` (remove also volumes);
2. Build images: `docker compose build --no-cache`;
3. Start: `docker compose up -d`;
4. Check all running containers: `docker compose ps`;
5. Please wait 60 seconds for initialization to complete;
6. Verify health checks: `docker compose ps` should show "(healthy)" for all;
7. Test API: `curl https://localhost/api/v1/health`.

#### Actual Results

- Deployment successful;
- API health check responds with 200 OK;
- SQLite databases created successfully in volumes;
- Logs show no critical errors;
- OPC UA connections established within 30 seconds of startup.

### 5.3.2 Persistence Across Restarts

**Test Objective** Verify that data persists after restarting the container thanks to Docker volumes.

#### Test Procedure

1. Running system, enter 100 data points via polling;
2. Query count: `docker exec hub-node-a sqlite3 /app/data/hub_storage.db "SELECT COUNT(*) FROM data_points"` on node-a - note N;
3. Stop all containers: `docker compose stop`;
4. Restart: `docker compose start`;
5. Query count after restart - verify same N;
6. Verify max Lamport clock preserved.

**Actual Results** Persistence verified. Pre-restart: 1247 records, max LC 8934. Post-restart: 1247 records, max LC 8934. New data points resume from LC 8935 (Lamport clock correctly initialized from database at startup). Volume mount `./data:/app/data` working.

# 6 User Guide

This section provides step-by-step instructions for deploying and using the OPC UA Industrial Plant HUB system. The guide covers initial setup, API usage, and common operations for check and troubleshooting.

## 6.1 Prerequisites

Before starting, ensure your system has:

- Docker Engine 24.0+ (`docker --version`)
- Docker Compose 2.20+ (`docker compose version`)
- PowerShell 7+ (Windows) or Bash (Linux/macOS)
- curl or similar HTTP client for API testing
- Modern web browser for Swagger UI access

## 6.2 Initial Setup

### 6.2.1 Step 1: Clone Repository and Navigate

```
git clone https://github.com/FedericoCapitanio-Unibo/DISI-DS-OPCUA-Industrial-HUB.git
cd DISI-DS-OPCUA-Industrial-HUB/
```

### 6.2.2 Step 2: Run Setup Script

The setup script generates TLS certificates and creates admin credentials.

#### Windows (PowerShell)

```
.\setup.ps1
```

#### Linux/macOS (Bash)

```
chmod +x setup.sh
./setup.sh
```

The script will prompt for:

- Admin username (default: `admin`)
- Admin password (minimum 8 characters)
- Optional: additional user accounts with read-only access

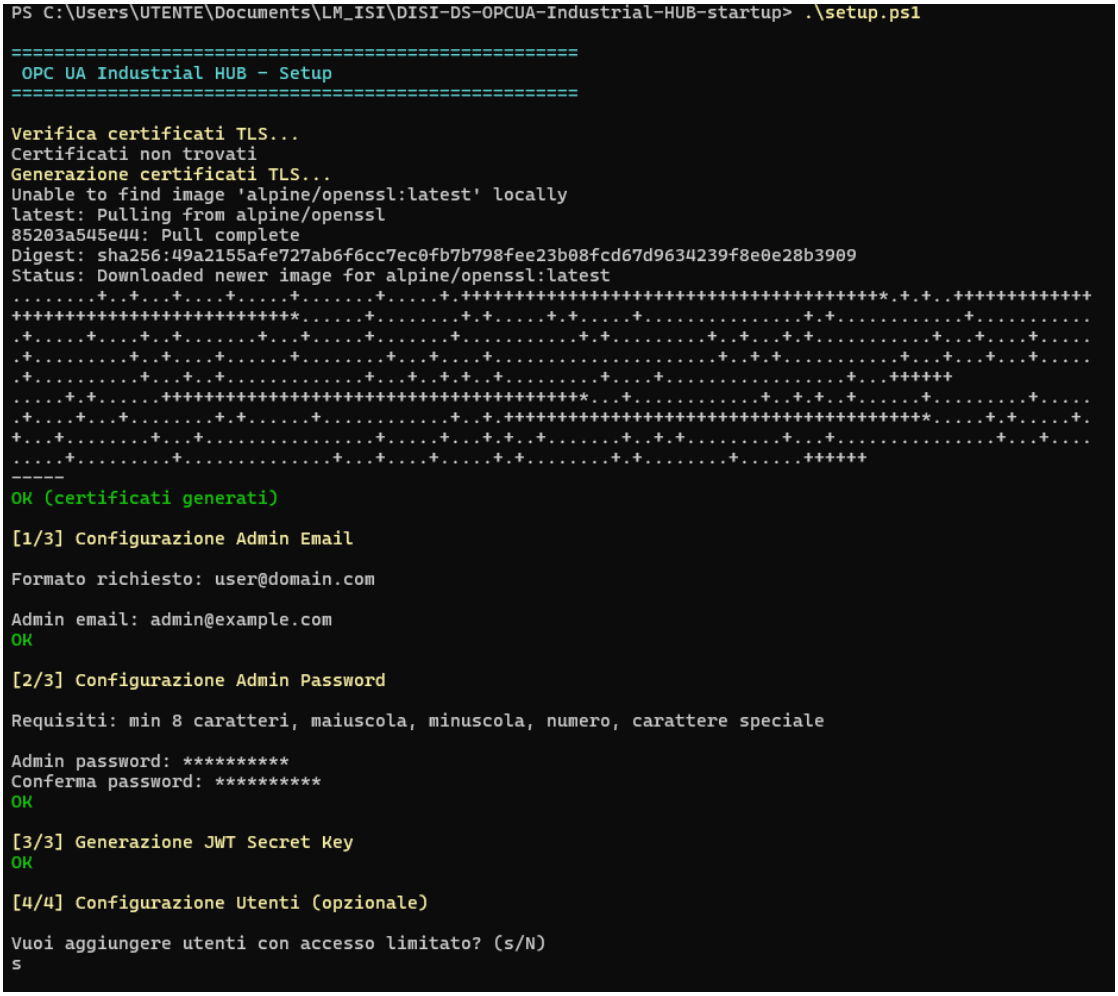


Figure 5: Setup script output part 1 - execution showing certificate generation and admin credentials creation

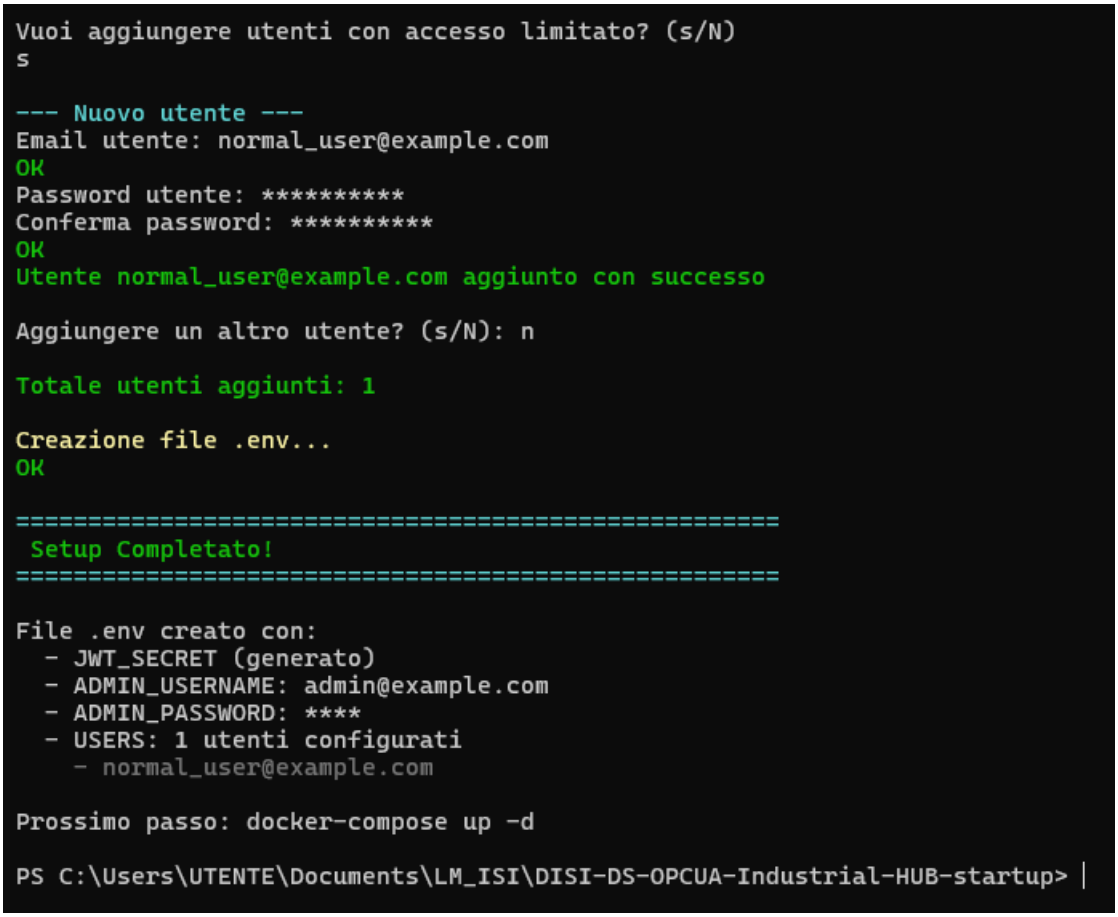


Figure 6: Setup script output part 2 - standard credentials creation and final summary

**Expected output**

### 6.2.3 Step 3: Start the System

```
docker compose up -d
```

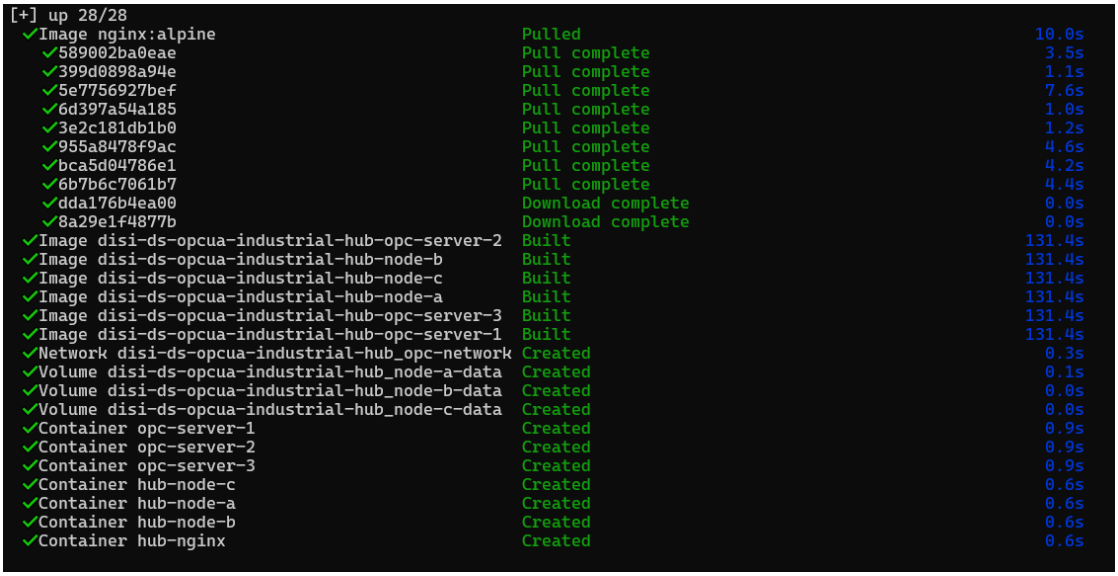


Figure 7: Docker Compose output

Wait 30-60 seconds for all services to initialize. Verify status:

```
docker compose ps
```

All containers should show `(healthy)` status.

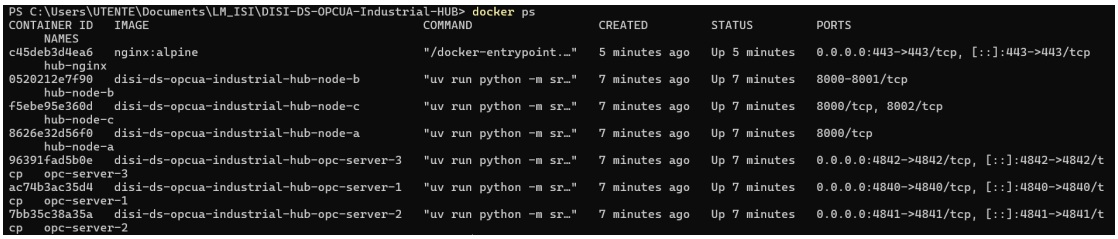


Figure 8: Docker Compose status showing all containers healthy

## 6.3 Accessing the API

Open your web browser and navigate to:

```
https://localhost/docs
```

**Note**: You will see a security warning because the TLS certificate is self-signed. Click "Advanced" and "Proceed to localhost" to continue.

The Swagger UI provides interactive API documentation where you can:

- View all available endpoints with descriptions;
- Test endpoints directly from the browser;
- See request/response schemas;
- Authenticate with your JWT token.

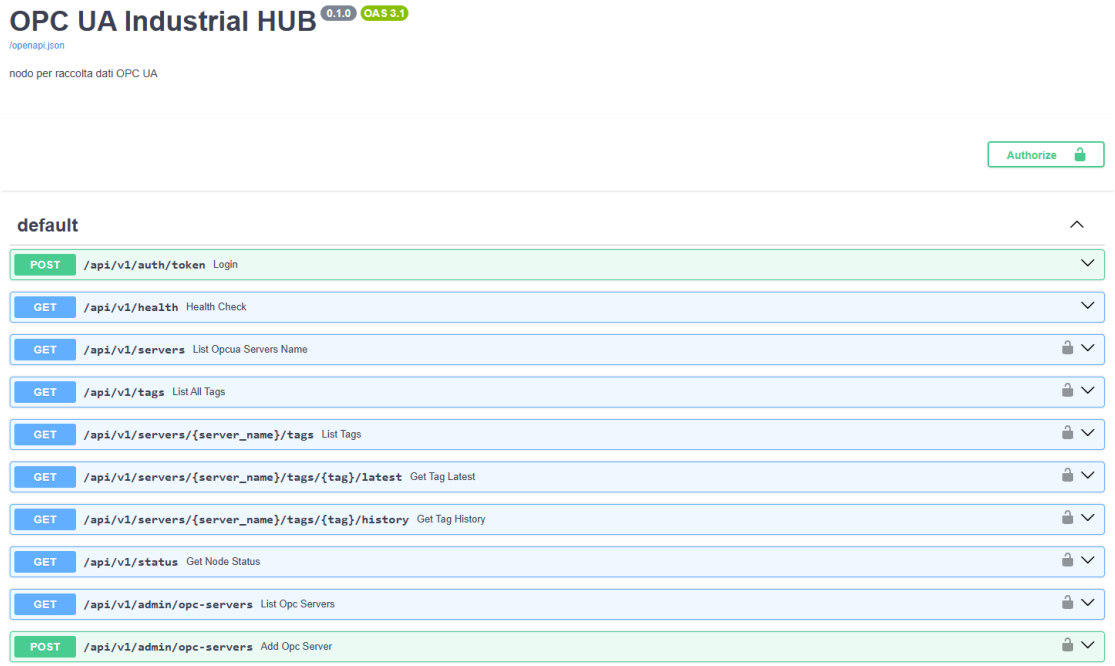


Figure 9: Swagger UI showing available API endpoints

## 6.4 Authentication

### 6.4.1 Step 1: Obtain JWT Token

1. Navigate to `POST /api/v1/auth/token`
2. Click "Try it out"
3. Fill in form:
   - `username`: your admin username;
   - `password`: your admin password.
4. Click "Execute"
5. Copy the `access_token` from the response

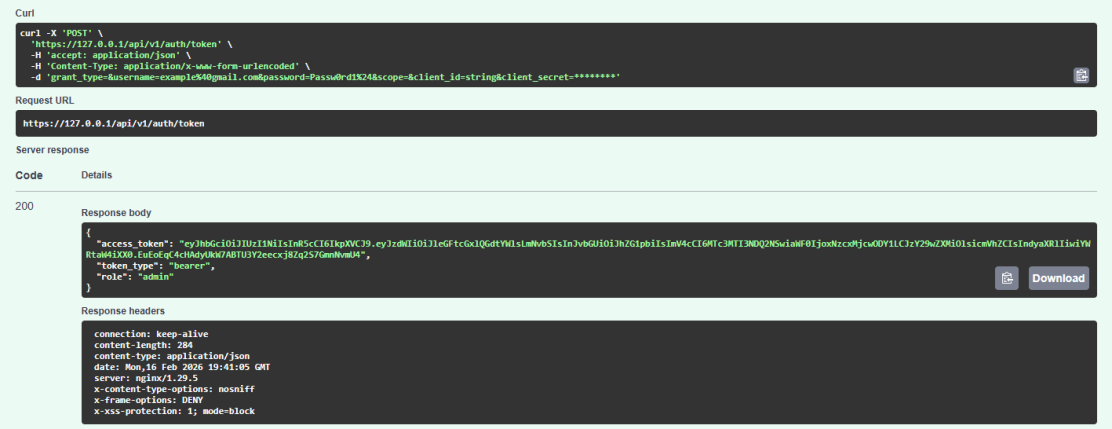


Figure 10: JWT token response showing access token and role

### 6.4.2 Step 2: Authenticate in Swagger UI

1. Click the "Authorize" button (lock icon) at the top right
2. Paste your JWT token in the "Value" field
3. Format: `Bearer your_token_here`
4. Click "Authorize" then "Close"

All subsequent API requests will now include authentication.

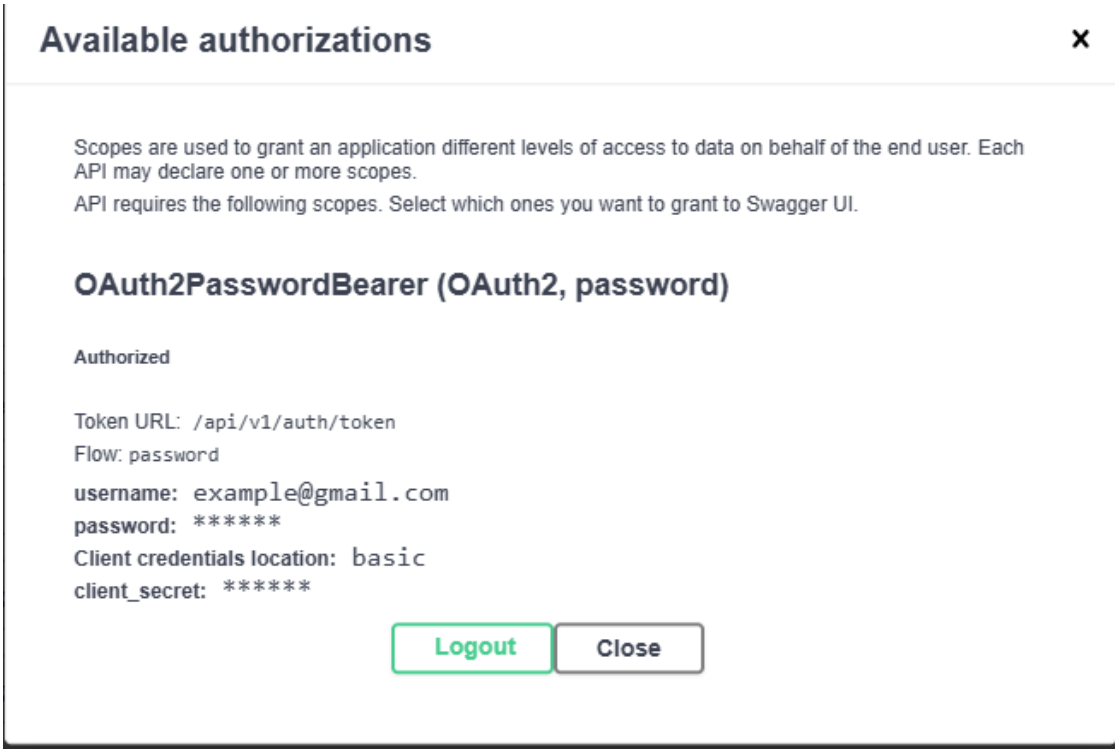


Figure 11: Swagger Authentication Form Data

## 6.5 Common Operations

### 6.5.1 List All Available Tags

Retrieve all tags from all configured OPC UA servers.

1. Navigate to `GET /api/v1/tags`
2. Click "Try it out"
3. Click "Execute"

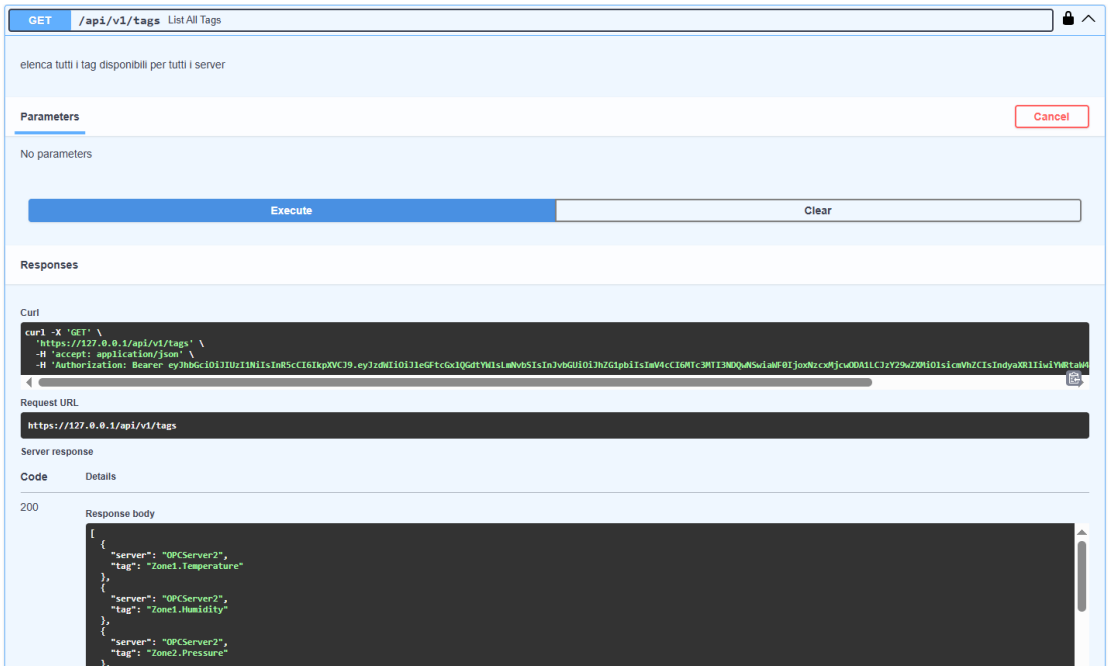


Figure 12: Swagger UI All Tags Request

### 6.5.2 Get Latest Value for a Specific Tag

Retrieve the most recent value for a specific OPC UA tag.

1. Navigate to `GET /api/v1/servers/{server_name}/tags/{tag}/latest`
2. Click "Try it out"
3. Fill in parameters:
   - `server_name`: OPCServer1
   - `tag`: Counters.ProductionCount
4. Click "Execute"

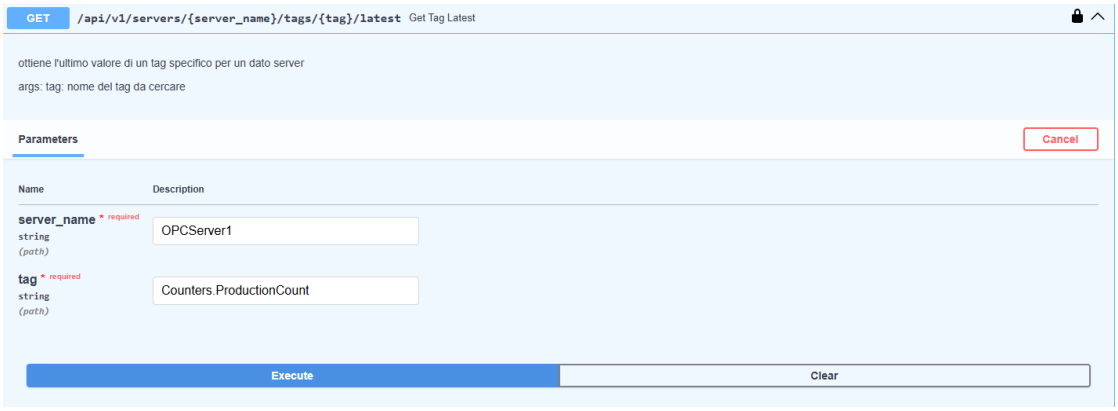


Figure 13: Latest value API request payload

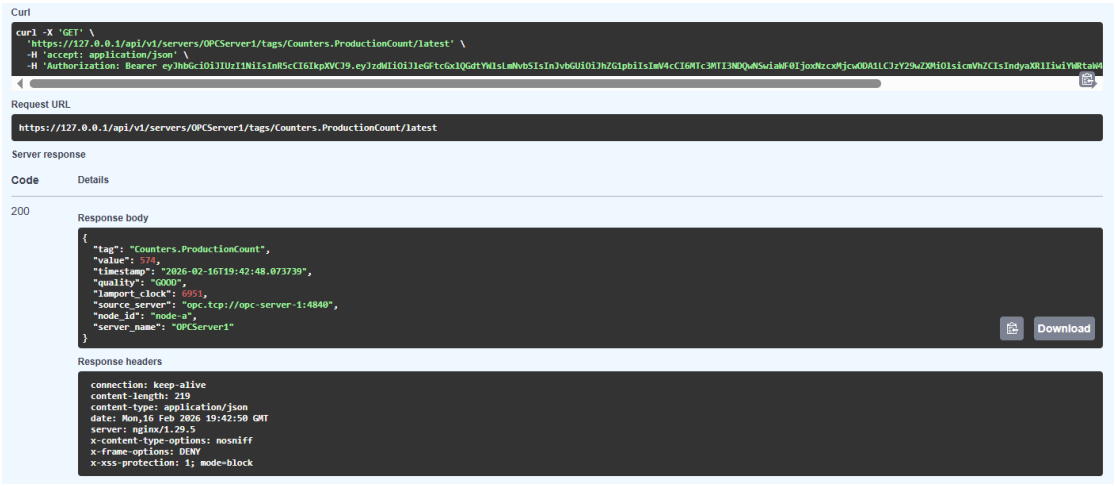


Figure 14: Latest value response showing complete data point information

### 6.5.3 Query Historical Data

Retrieve historical values for a tag within a time range.

1. Navigate to `GET /api/v1/servers/{server_name}/tags/{tag}/history`;
2. Click "Try it out";
3. Fill in parameters;
4. Click "Execute".

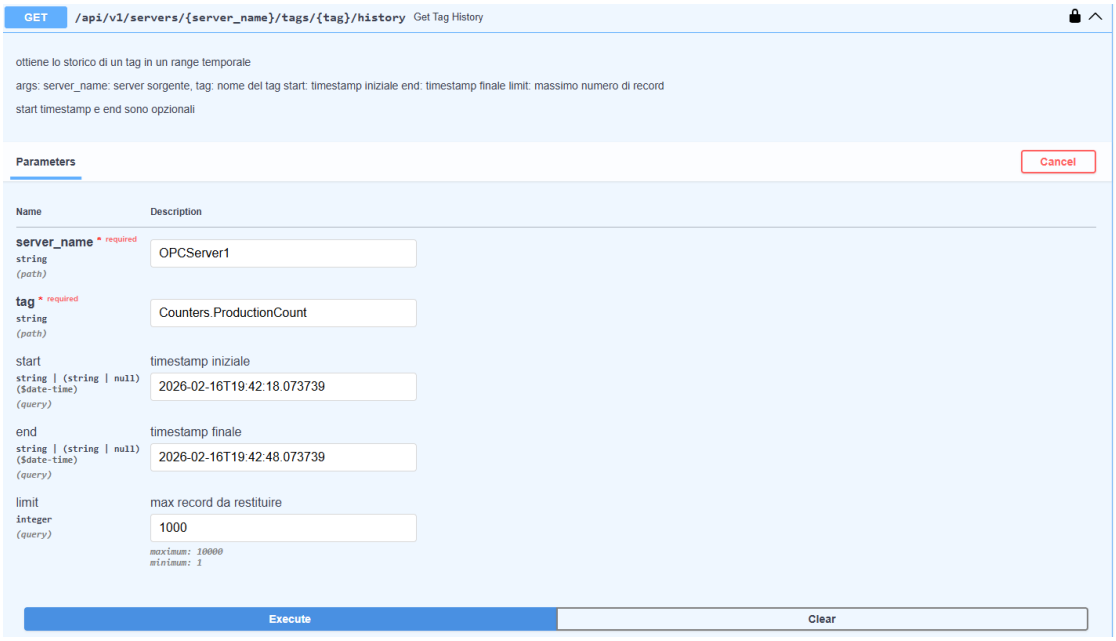


Figure 15: Historical API request payload

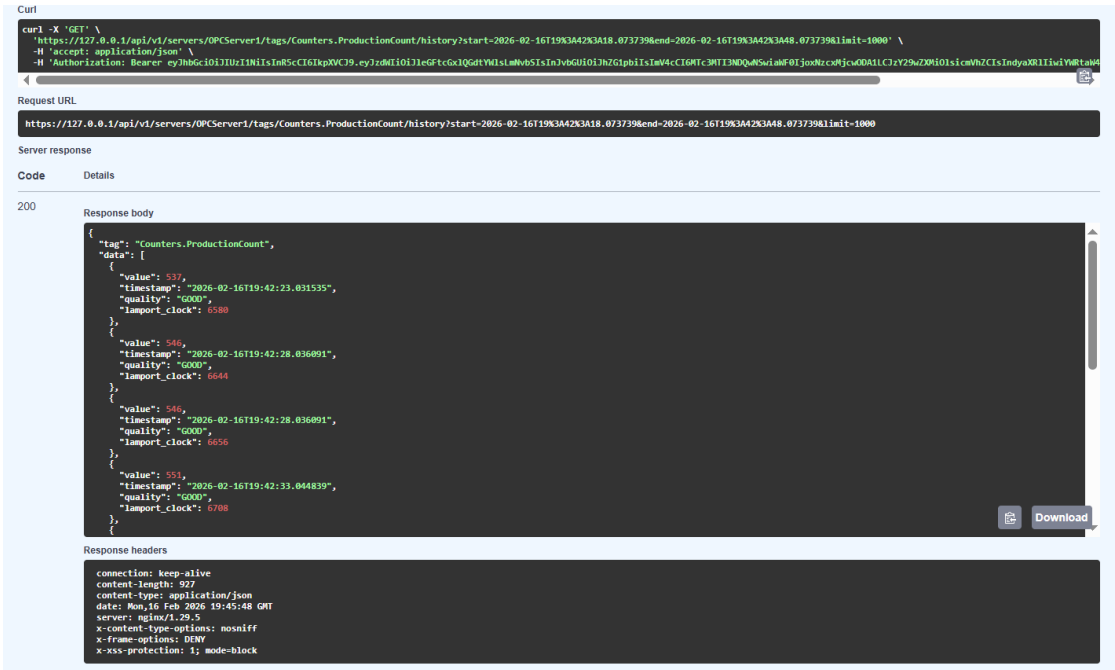


Figure 16: Historical data response showing time-series data points

### 6.5.4 Add a New OPC UA Server (Admin Only)

Dynamically add a new OPC UA server without restarting the system.

1. Navigate to `POST /api/v1/admin/opc-servers`
2. Click "Try it out"
3. Fill in request body:

```
{
  "server_name": "OPCServer4",
  "endpoint": "opc.tcp://opc-server-4:4843"
}
```

4. Click "Execute"

**Propagation**: The configuration will automatically propagate to all other HUB nodes within 30 seconds via Anti-Entropy protocol. All nodes will start collecting data from the new server.

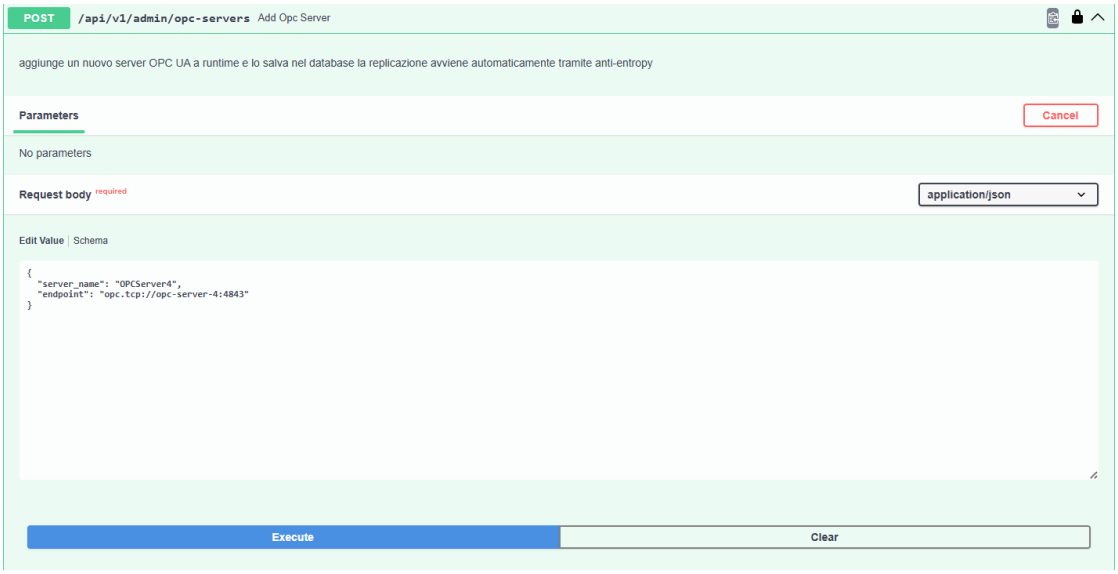


Figure 17: Add OPC UA server API request payload

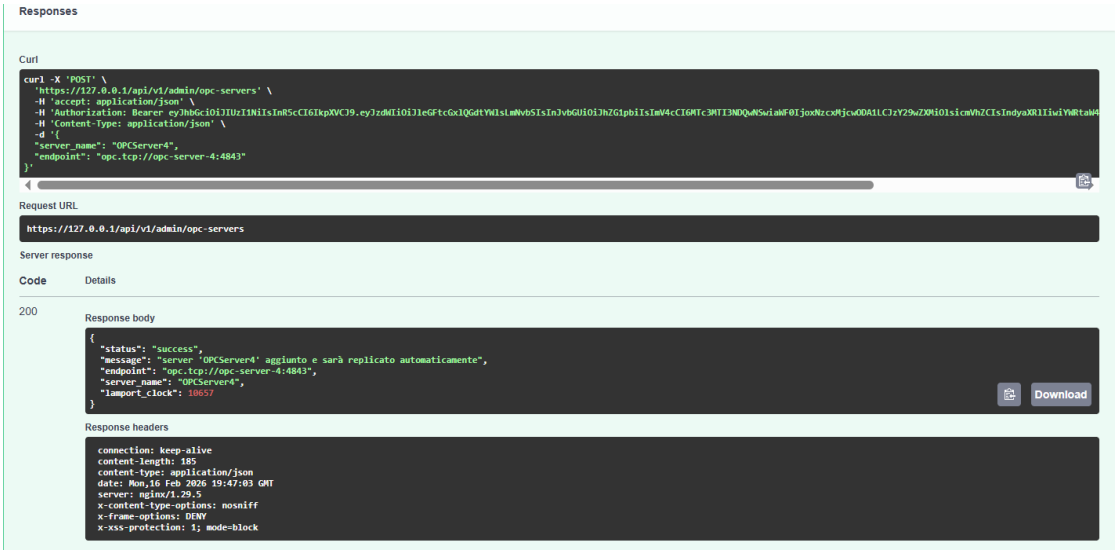


Figure 18: Add OPC UA server endpoint showing request and success response

## 6.6 Monitoring and Troubleshooting

### 6.6.1 View Container Logs

Monitor real-time logs from all containers:

```
docker compose logs -f
```

View logs from a specific service:

```
docker compose logs -f node-a
docker compose logs -f nginx
```

### 6.6.2 Common Issues

**Issue: "Connection refused" when accessing https://localhost**

- **Cause**: nginx container not started or not healthy
- **Solution**: Check status with `docker compose ps` and view logs with `docker compose logs nginx`

**Issue: "401 Unauthorized" on API requests**

- **Cause**: Missing, expired, or invalid JWT token
- **Solution**: Obtain a new token via `POST /api/v1/auth/token`

#### Issue: "403 Forbidden" on admin endpoints

- **Cause**: User token used instead of admin token
- **Solution**: Login with admin credentials to get admin JWT

#### Issue: No data appearing for OPC UA tags

- **Cause**: OPC UA connection failed or simulators not running
- **Solution**: Check logs with `docker compose logs node-a | grep OPC` and verify OPC UA server containers are running

### 6.7 Shutdown

Stop all containers:

```
docker compose down
```

Stop and remove all data (including databases):

```
docker compose down -v
```

**Warning**: The `-v` flag deletes all collected OPC UA data. Use only when you want to completely reset the system.

# 7 Self-evaluation

As this project was developed individually, I assumed full responsibility for the entire software development lifecycle, from requirements elicitation to deployment and validation. This section provides an objective assessment of my role and the final product's strengths and weaknesses.

## 7.1 Role Description

My role covered all aspects of the distributed system development:

- **System Architect**: Defined the Hybrid Client-Server/P2P architecture and selected the AP (Availability/Partition Tolerance) positioning in the CAP theorem to prioritize 24/7 industrial monitoring over strong consistency. Designed the leaderless multi-master replication strategy using Lamport clocks for ordering.
- **Backend Engineer**: Implemented the three core distributed protocols:
  - Gossip Protocol for failure detection with configurable heartbeat intervals;
  - Anti-Entropy Protocol for pull-based eventual consistency;
  - OPC UA Ingestor with automatic retry and connection management.

  Developed the REST API layer using FastAPI with JWT authentication and RBAC authorization, integrating SQLAlchemy async ORM for database operations.
- **DevOps Engineer**: Orchestrated the multi-container deployment using Docker Compose, configured nginx as load balancer with TLS termination and health checks, and created automated setup scripts with PowerShell and Bash for credential generation and LS certificates management.
- **Quality Assurance**: Designed and executed functional tests for functional requirements, performance tests and fault tolerance tests including network partition simulation with iptables.
- **Technical Writer**: Produced comprehensive documentation covering architectural logic, design patterns, implementation choices, and deployment procedures, with references between requirements and validation results.

## 7.2 Product Assessment

### 7.2.1 Strengths

**Robust Distributed Coordination** The system successfully implements distributed systems patterns without a central coordinator. The combination of Gossip Protocol for failure detection, Anti-Entropy Protocol for data synchronization and Lamport Clocks that handle conflict resolution, provides strong eventual consistency guarantees verified through partition testing.

**High Availability with No Single Point of Failure** The leaderless multi-master architecture eliminates SPOFs at the application layer. Validation tests confirmed that the system continues operating correctly with 2 out of 3 nodes failed, and nginx automatically redirects traffic away from unhealthy nodes within 30 seconds. The AP behavior of the CAP theorem positioning was validated through network partition tests, where both partitions continued serving clients independently and converged automatically when network starts working as before.

**Industrial Standards Interoperability** The system connect industrial automation protocol like OPC UA with modern web technologies, REST, JSON and JWT, using open standards. OPC UA Protocol ensures compatibility with any compliant PLC or SCADA system regardless of vendor. The REST API with OpenAPI documentation enables integration with existing enterprise systems such as ERP, BI tools and dashboards without vendor lock-in. TLS 1.3 encryption and JWT-based authentication provide a security level that is a strong constraints in general but espcially in industrial environments.

**Deployment Simplicity and Reproducibility** Docker Compose orchestration enables one-command deployment (`docker compose up`) of the entire distributed stack. Automated setup scripts generate TLS certificates and credentials, reducing manual configuration errors. The use of SQLite eliminates external database dependencies, simplifying the deployment architecture while maintaining ACID guarantees for local storage.

**Comprehensive Validation** All 10 functional requirements were explicitly tested with documented procedures, expected results, and actual measurements. Fault tolerance testing included deliberate node crashes, network partitions, and OPC UA connection failures, validating graceful degradation and automatic recovery.

## 7.3 Learning Outcomes

This project provide direct experience with distributed systems concepts studied in the course:

- **CAP Theorem**: Practical understanding of the tradeoffs between consistency, availability, and partition tolerance through implementation and testing of AP system behavior
- **Logical Clocks**: Implementation of Lamport clocks for establishing causal ordering in the absence of synchronized physical clocks
- **Eventual Consistency**: Design and validation of convergence guarantees using Anti-Entropy protocol with measurable convergence time
- **Failure Detection**: Implementation of Gossip Protocol with analysis of completeness and accuracy tradeoffs

- **Replication Strategies**: Comparison of push-based vs pull-based replication, understanding advantages of pull for backpressure and idempotence

The project successfully demonstrates that distributed systems principles can be applied to industrial automation contexts, bridging the gap between academic theory and real-world manufacturing requirements.

## 7.4 Future Works, Limitations and Known Issues

During testing, some limitations of the current system emerged that do not compromise the main requirements but represent opportunities for improvement:

- **Dynamic Users Creation**The system don't allow to add new users when the application is running. In the next release this could be added by also considering the fact that users could be handle not with env variables but we a shared DB from all the nodes.
- **Rate Limiting Absent** The system does not implement per-user rate limiting. A client with a valid token can send unlimited requests, causing potential DoS. Current mitigation: nginx global connection limit. Future mitigation: implement rate limiting with Redis and sliding window algorithm.
- **Token Revocation Not Supported** Stateless JWT does not allow immediate revocation. If a token is compromised, it remains valid until expiration (24 hours). Current workaround: short expiration. Future solution: token blacklist in Redis or switch to stateful session tokens.
- **Theoretical Lamport Clock Overflow** Lamport clock is INTEGER (32-bit signed in SQLite, range $\pm 2$ billion). With a rate of 100 events/second, overflow will happens after some days. Not critical for prototype but would require management in production with coordinated periodic reset or upgrade to BIGINT as examples.
- **OPC UA Poll Mode** In the current system, OPC UA readings are performed using Poll Mode. Future versions will need to include the option of adding Subscription Mode, which reduces the load on the server and allows the full potential of the OPC UA protocol to be exploited. The system currently uses the OPC UA protocol to communicate with the PLC.
- **OPC UA Full Tree Browsing** The current implementation performs a complete browse of the entire OPC UA node tree at startup for each configured server. While this approach ensures automatic discovery of all available variables without manual configuration, it introduces significant scalability issues with large industrial OPC UA servers. Industrial PLCs and SCADA systems can expose OPC UA trees with thousands or tens of thousands of nodes organized in deep hierarchical structures. A complete browse operation on such servers can take several minutes and consume substantial memory resources during the traversal and indexing phase. This creates two critical problems:

- **Startup Time**: HUB nodes can take 5-10 minutes to become operational when connecting to large OPC UA servers, violating the expected startup time of ¡60 seconds for production environments;
- **Resource Consumption**: Browsing and storing metadata for thousands of unused nodes wastes memory and CPU cycles, degrading overall system performance;
- **Network Overhead**: Large browse operations generate significant network traffic between HUB nodes and OPC UA servers, potentially impacting industrial network performance.

Currently the system should only be deployed with OPC UA servers exposing a limited number of variables (¡500 nodes per server) or with OPC UA simulators designed for testing.

- **No Metrics Endpoint** The system does not expose metrics monitoring system such as Prometheus. Monitoring in production would require scraping logs or manually adding `/metrics` endpoints with the prometheus-client library.

# References


[1] Jonathan Tobias Da Silva, Andre Luis Dias, and Ivan Nunes Da Silva. A survey on opc ua protocol: overview, challenges and opportunities. In *2023 15th IEEE International Conference on Industry Applications (INDUSCON)*, pages 1523–1530. IEEE, 2023.

[2] Docker Inc. Docker: Accelerated container application development. `https://www.docker.com/`, 2024. Accessed: 2025-01-16.

[3] David Ferraiolo, Janet Cugini, D Richard Kuhn, et al. Role-based access control (rbac): Features and motivations. In *Proceedings of 11th annual computer security application conference*, pages 241–48, 1995.

[4] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616, RFC Editor, 1999. DOI: 10.17487/RFC2616.

[5] FreeOpcUa Community. asyncua: Opc ua / iec 62541 client and server for python. `https://github.com/FreeOpcUa/opcua-asyncio`, 2024. Accessed: 2025-01-16.

[6] Michael Jones, John Bradley, and Nat Sakimura. Json web token (jwt). RFC 7519, RFC Editor, 2015. DOI: 10.17487/RFC7519.

[7] JSON.org. Introducing json. `https://www.json.org/json-en.html`, 2024. Accessed: 2025-02-16.

[8] Hugo Krawczyk and Hoeteck Wee. The optls protocol and tls 1.3. In *2016 IEEE European symposium on security and privacy (EuroS&P)*, pages 81–96. IEEE, 2016.

[9] Moriarity, Michael. python-jose: Javascript object signing and encryption implementation in python. `https://python-jose.readthedocs.io/en/latest/`, 2024. Accessed: 2025-01-16.

[10] NGINX Inc. nginx: High performance load balancer, web server, and reverse proxy. `https://nginx.org/`, 2024. Accessed: 2025-01-16.

[11] Pydantic. Pydantic: Data validation using python type hints. `https://docs.pydantic.dev/latest/`, 2024. Accessed: 2025-02-18.

[12] Python Software Foundation. Python programming language. `https://python.org/`, 2024. Accessed: 2025-01-16.

[13] SQLAlchemy Authors. Sqlalchemy: The python sql toolkit and object relational mapper. `https://www.sqlalchemy.org/`, 2024. Accessed: 2025-01-16.

[14] SQLite Development Team. Sqlite database engine. `https://sqlite.org/`, 2024. Accessed: 2025-01-16.

[15] Tiangolo, Sebastián. Fastapi: Modern, fast web framework for building apis with python. https://fastapi.tiangolo.com/, 2024. Accessed: 2025-01-16.

[16] Wikipedia. Overall equipment effectiveness. https://en.wikipedia.org/wiki/Overall_equipment_effectiveness, 2024. Accessed: 2025-02-16.