Sensing and Actuation Networks and Systems [2025-2026]

Autonomous Warehouse using IIoT

Esta versão apenas tem umas pequenas alterações ou esclarecimentos, indicadas com um fundo cor de laranja

1. Introduction

This assignment aims to develop a comprehensive full-stack IoT solution for an **Autonomous Warehouse**, covering data collection, autonomous processing, task dispatching, and visualization. The student's focus will be on managing multiple I/O sources (MQTT, UDP Sockets) within an event-driven loop.

2. Objectives

Students successfully concluding this work should be able to:

- Simulate and collect real-time data from **multiple, single-process** sources.
- Transfer data reliably using the **MQTT** protocol.
- **Process** and **aggregate** distributed state information.
- Store all state changes and events in an **InfluxDB** database.
- Implement a **UDP socket-based** interface for system alerts and event injection.
- Display and analyze real-time and historical data in a **Grafana** dashboard.

3. Support Material

- MQTT broker running on a Raspberry Pi available at the Department of Informatics Engineering (srsa-pi-8.dei.uc.pt).
- Files and examples from previous classes (PL2, PL 04 - 09).
- Python JSON: https://www.w3schools.com/python/python_json.asp
- Python socket library: https://docs.python.org/3/library/socket.html
- Python struct library: https://docs.python.org/3/library/struct.html
- Python selectors library: https://docs.python.org/3/library/selectors.html

4. Scenario

This project simulates a smart warehouse logistics system. The warehouse consists of three types of simulated entities:

- **Autonomous Mobile Robots (AMRs)** – 4 AMR: Follow a deterministic state cycle, combined with a random probability of failure:
    - I**DLE:** The robot remains in this state indefinitely until it receives a task (via MQTT).
    - **Execution Cycle:** Upon receiving a task, the robot follows this fixed timing sequence.
        - *MOVING_TO_PICK:* Lasts exactly 3 seconds.
        - *PICKING:* Lasts exactly 1 second.
        - *MOVING_TO_DROP:* Lasts exactly 2 seconds.
        - *DROPPING:* Lasts exactly 1 second.
        - After dropping, the robot returns to IDLE.
    - **Battery & Charging:**
        - *Activity Time:* The robot has a functional autonomy of 100 seconds of active operation (moving/picking/dropping). Students must implement a battery-decay rate of 1% per active second.
        - *CHARGING*: When the robot goes to charge (either voluntarily or forced), it stays in the CHARGING state for 10 seconds before returning to IDLE with 100% battery.
    - **Simulated Failure (STALLED):**
        - During any MOVING state, there should be a small, random probability (e.g., 5%) that the robot will enter the STALLED state.
        - *Behavior when STALLED:* The robot stops all movement and timer progression. It ignores standard task commands. It will only recover when it receives a high-priority Override Command (e.g., FORCE_CHARGE) from the System Monitor.
- **Smart Shelves** – 10 shelves: Monitor their own inventory. They report their **item_id** and **current_stock** (units). The management of these stations is not defined. Each group of students may define how it works. On reaching 0 each shelf automatically refills after 2 seconds.
- **Packing Stations** – 3 stations: Report their **status** (AVAILABLE, BUSY). The management of these stations is not defined. Each group of students may define how it works.

The goal is to ensure efficient fleet management and near-real-time monitoring of the warehouse. A central **Fleet Coordinator** receives "new orders" via a UDP socket from the **Client Manager** and dispatches an available IDLE robot to fulfill the order.

When system-level problems are detected (e.g., a stalled robot or a low-battery robot that is not charging), a separate **System Monitor** issues high-priority override commands to ensure safety and efficiency. All collected data is stored in a time-series database, allowing continuous access and analysis of the fleet's performance.

5. Network Configuration

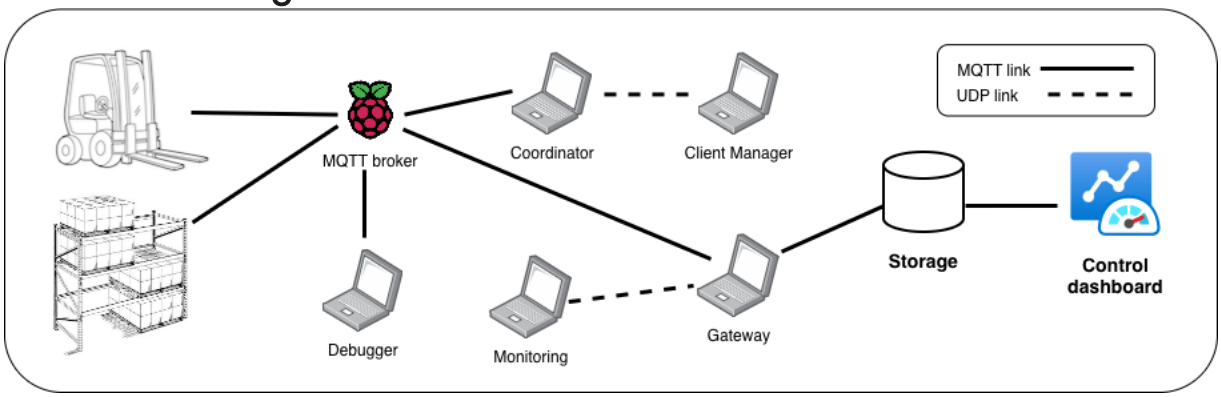

Figure 1. Autonomous Warehouse components.

The network configuration to be used consists of different parts:

1. **AMR Robot Simulator** (`amr_robot.py`): A script that simulates **one** robot. To simulate the entire fleet, **4 instances** of this script must be run.
2. **Smart Shelves Simulator** (`shelves.py`): A script that simulates **one** sensor. To simulate all sensors, **ten (10) instances** must be run.
3. **MQTT Broker:** A Mosquitto MQTT broker provides the communication backbone.
4. **Warehouse Gateway** (`warehouse_gateway.py`): The universal translator and the record keeper of the entire system. It stands between the raw, chaotic world of the physical warehouse (robots, sensors) and the organized, high-level logic of the management software (**Fleet Coordinator**). It performs three main jobs: *Data Normalization*, *Protocol Translation*, and *Archiving*.
5. **Fleet Coordinator** (`fleet_coordinator.py`): While the **Warehouse Gateway** acts as the translator, the **Fleet Coordinator** acts as the manager or the brain of the operation. It does not care about low-level details (such as how many bytes are needed to move a robot wheel); instead, it focuses on high-level logistics and strategy.
6. **System Monitor** (system_monitor.py)**:** This service analyzes long-term health. It tracks robot states and, on detecting an issue, sends a **UDP message** with an override command (as a JSON string) to the Warehouse Gateway.
7. **Client Manager** (`client_order_injector.py`): A Python script that acts as a **UDP client** to send a new order (as a JSON string) to the **Fleet Coordinator**.
8. **MQTT Debugger**: Displays all MQTT messages exchanged.
9. **Storage**: All data is stored in an InfluxDB Cloud (free plan) time-series database.
10. **Control Dashboard**: Information is visualized in a Grafana Cloud (free plan) dashboard.

All scripts that appear with a laptop in Figure 1 run on the student's computer. Robots and Shelves can also run on the student's computer but, during the defence they will be run on a Raspberry Pi. The work uses an external MQTT broker located at srsa-pi-8.dei.uc.pt. To access the MQTT broker within DEI premises, students must be connected to the "DEI" network. To access it from outside DEI, students need to connect to the department's VPN. Both InfluxDB Cloud and Grafana Cloud are available with an Internet connection.

6. Description of Work

This project integrates the core concepts of the SRSA course into a functional Multi-Layer IoT Architecture:

- Sensing & Actuation. You will simulate physical devices (robots/shelves) and their constraints.
- Middleware & Interoperability. You will build a Gateway to translate between high-level JSON and low-level binary protocols.
- Network Management. You will implement Orchestration (Fleet Coordinator) and Self-Healing mechanisms (System Monitor) to ensure resilience.
- Data Engineering. You will practice the full ETL pipeline, which involves collecting data via MQTT, storing it in a Time-Series Database (InfluxDB), and visualizing it (Grafana).

The interactions between the components that you should implement for your solution are depicted in Figure 2.

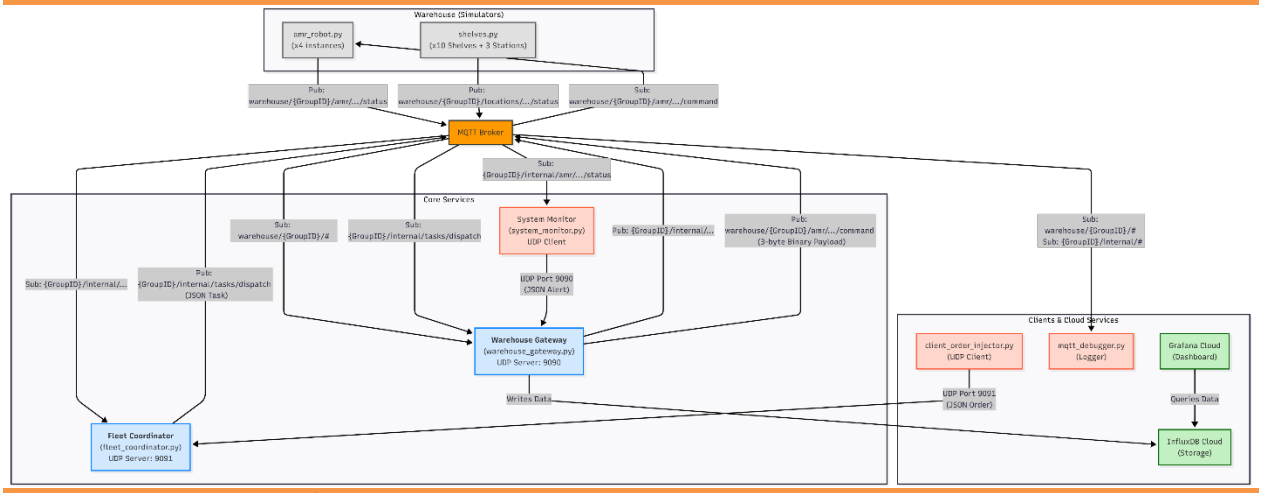

Figure 2. Interaction between Warehouse components.

Note: In this project, as in real-world engineering scenarios, some functional details may be left open to interpretation (e.g., specific logic for static stations). Any ambiguity or open issues identified in this document should be discussed with the professor during the PL sessions for clarification. If a specific behavior remains undefined, the group is expected to make a reasonable engineering assumption to proceed. Crucially, all such assumptions must be clearly identified and justified during the final defense, with the rationale for the chosen architectural or logical approach demonstrated.

Each team has to write the Python code for the following scripts:

1. `amr_robot.py` (The single-robot simulator)
2. `shelves.py` (The single-sensor simulator)
3. `warehouse_gateway.py` (The main data hub and alert server)
4. `fleet_coordinator.py` (The task manager and order server)
5. `system_monitor.py` (The health monitor and alert client)
6. `client_order_injector.py` (The UDP client for sending new orders)

7. `mqtt_debugger.py` (The network logging utility)

The data is stored in a time series database (**InfluxDB**) and displayed on a Control Dashboard (**Grafana**).

Topics used in the simulation must follow this new, hierarchical structure:

- **Data (Publish):**
  - `warehouse/{GroupID}/amr/{robot_id}/status` → (For mobile assets)
  - `warehouse/{GroupID}/locations/{zone_id}/{asset_id}/status` → (For static assets)
- **Commands (Subscribe):**
  - `warehouse/{GroupID}/amr/{robot_id}/command`
- **Internal (Gateway, Coordinator):**
  - `{GroupID}/internal/amr/{robot_id}/status` → (Clean data)
  - `{GroupID}/internal/static/{asset_id}/status` → (Clean data)
  - `{GroupID}/internal/tasks/dispatch` → (Task commands)

Note: The Group ID is the number of student of one of the group members.

6.1. Warehouse Simulators (`amr_robot.py, shelves.py`)

This module will be simulated by running multiple instances of two scripts representing the robots and the sensors.

AMR Robot Data is sent from each robot using a JSON object with the following structure. Below is an example:

```
#AMR Robot Data
{
  "robot_id": "AMR-1",
  "timestamp": "2025-03-15T12:34:56Z",
  "location_id": {"SHELF-S1"},
  "battery": 88,
  "status": "IDLE"
}
```

Since the cycle is deterministic, the `location_id` updates based on the robot's state: **IDLE** → *"DOCK"*, **MOVING_TO_PICK** → *"TRANSIT"*, **PICKING** → *"SHELF-Sx"*, **MOVING_TO_DROP** → *"TRANSIT"*, **DROPPING** → *"PACKING_ZONE"*, **CHARGING** → *"CHARGING_STATION"*

Static Sensor Data is sent from each **shelf or station** using a JSON object. Below are examples of the shelf and station data:

```
#Shelf Data
{
  "asset_id": "S1",
  "type": "SHELF",
  "item_id": "item_A",
  "stock": 150,
  "unit": "units"
}
```

```
#Station Data
{
  "asset_id": "P1",
  "type": "PACK_STATION",
  "status": "AVAILABLE"
}
```

Simulation Rules. There are 4 AMR models (AMR-1 to AMR-4) and 12 static sensors, now organized by zone as described in Table 1.

Zone ID	Asset ID	Type	item_id	Stock Unit
storage-a	S1-S5	SHELF	item_A, item_B, ...	units | 1 unit = 23 kg
storage-b	S6-S10	SHELF	item_C, item_D, ...	kg
packing_zone	P1-P3	PACK_STATION	N/A	status

Table 1. Static Sensors to Simulate

- The `amr_robot.py` script will be launched 4 times (once for each robot). It takes the `robot_id` as a command-line argument.
  - **Usage:** $>python3 amr_robot.py {GroupID} {robot_id}
  - **Publishes to:** warehouse/{GroupID}/amr/{robot_id}/status
  - **Subscribes to:** warehouse/{GroupID}/amr/{robot_id}/command
  - **Command Processing & Logic**: The script must use the struct library to decode the 3-byte binary payload received on the command topic (see "Warehouse Gateway" section). It must extract two target IDs to manage its deterministic cycle:
    - **Pick Phase:** Upon receiving the command, the robot switches to MOVING_TO_PICK. When the move time (3s) expires, it updates its location_id to SHELF-S{Byte2} (e.g., if Byte 2 is 0x0A, the location becomes "SHELF-S10").
    - **Drop Phase:** After the pick wait time (1s) and the move-to-drop wait time

(2s), the robot updates its location_id to STATION-P{Byte3} (e.g., if Byte 3 is 0x01, the location becomes "STATION-P1").

- The `shelves.py` script will be launched 10 times. It now takes the `zone_id` and `asset_id` as arguments.
  - **Usage:** $>python3 shelves.py {GroupID} {zone_id} {asset_id} {update_time}
  - **Example:** $ python3 shelves.py MyGroup storage-a S1 10
  - **Publishes to:** warehouse/{GroupID}/locations/{zone_id}/{asset_id}/status
- Each robot must also process the FORCE_CHARGE override command sent by the System Monitor. On receiving this command, the robot will drop its current task and immediately change its status to MOVING_TO_CHARGE.

6.2. MQTT Broker

A Raspberry Pi running Mosquitto v2.0.11 MQTT broker at IP 10.6.1.9 (srsa-pi-8.dei.uc.pt), port 1883, will be available to students. This broker will handle all students' topics. **Students may use any other MQTT broker available for their tests.**

6.3. Warehouse Gateway (`warehouse_gateway.py`)

This agent runs on the students' computers. It is the central data hub (InfluxDB interaction) and the only component that encodes and sends byte commands to the robots. It has the following functionalities:

- Subscribes to all raw sensor topics using a multi-level wildcard: `warehouse/{GroupID}/#`
- Communicates with the Fleet Coordinator using MQTT (receives JSON task commands on `{GroupID}/internal/tasks/dispatch`).
- Receives high-priority alert messages from the System Monitor by running a UDP Server (on port **9090**).
- The Gateway is responsible for **extracting** and **standardizing** data from all sensors. For example, it must convert all stock from kg to a common unit (e.g., units, or vice-versa) based on Table 1.
- Once the data has been standardized and processed, the Gateway stores it in InfluxDB.
- It also forwards the clean, standardized data to internal MQTT topics (`{GroupID}/internal/amr/AMR-1/status`) for other services to consume.
- The Gateway is responsible for formatting and sending all control commands. It translates JSON commands (received from MQTT or UDP) into the precise byte format the robots expect.
- Instructions are sent to the robots as a 3-byte binary payload, rather than using JSON. The `Warehouse Gateway` is responsible for encoding JSON commands into this 3-byte binary format. Conversely, the `amr_robot` uses the `struct` library (https://docs.python.org/3/library/struct.html) to decode the binary payload back into commands. The precise structure of this 3-byte command, including the meaning, size, and position of each byte, is detailed in Table 2.

Bytes	Meaning	Example Value
1	Command Type	0x01: **EXECUTE_TASK** (Triggers the Move → Pick → Drop cycle) 0x03: FORCE_CHARGE (Override command from Monitor)
2	Target Shelf ID	0x0A (Integer 10) represents Shelf S10
3	Target Station ID	0x01 Parking_Zone Station P1

Table 2. Command format

6.4. Fleet Coordinator (`fleet_coordinator.py`)

The Fleet Coordinator serves as the brain of the warehouse, overseeing all tasks. It maintains a real-time world state by listening to all internal sensor data and receives new orders via its UDP server. Based on this information, it decides when and where to dispatch an available robot by sending a JSON task command to the `Warehouse Gateway` via MQTT.

- It subscribes to the internal, clean data topics from the Gateway (`{GroupID}/internal/+/+/status`).
- It maintains a real-time world state (e.g., *a Python dictionary*) of all robot states, shelf stock, and station availability.
- It runs a UDP Server (on port **9091**) that listens for new order connections from the `client_order_injector.py` client.
- An order is a JSON string, e.g.: {"item": "item_A", "quantity": 10, "pack_station": "P1"}.
- The main function will be a while True loop that does:
  1. **On MQTT event:** Calls MQTT client to process incoming sensor messages and update its `world_state`.
  2. **On UDP event:** Handle new order connections and add them to a pending list.
- When an order is received, it analyzes its `world_state` to find an `IDLE` robot and a shelf with the required item and stock.
- When a match is found, it generates a JSON task command and sends it to the Warehouse Gateway via the internal MQTT topic: via the internal MQTT topic: `{GroupID}/internal/tasks/dispatch`. If no match is found, the order remains in the "Pending Orders" queue until all conditions are met.

```
# Match command example

{
  "robot_id": "AMR-3",
  "command": "EXECUTE_TASK",
  "target_shelf_id": "S1",
  "target_station_id": "P1"
}
```

6.5. System Monitor

This module is responsible for analyzing the health of the fleet, independent of the `Fleet Coordinator's` tasks.

- It subscribes to the Gateway's internal, clean robot data topics (`{GroupID}/internal/amr/+/status`).
- It keeps a record of robot states over time to detect anomalies.
- It establishes the definition of robot health using two (2) levels: NORMAL and CRITICAL.
- If a `CRITICAL` status is detected, it sends a JSON alert message (e.g., `{"robot_id": "AMR-2", "level": "CRITICAL", "override_task": "FORCE_CHARGE", ...}`).
- Each group of students must define how to assess the health of a robot.
- Example of how to assess robot health:
  - **STALLED:** Track the `location` and `status` of each robot. If `status` is `MOVING` but its location has not changed for 30 seconds, its health becomes `CRITICAL`.
  - **LOW BATTERY:** If a robot's `battery` is below 15% and its status is not `CHARGING` or `MOVING_TO_CHARGE`, its health becomes `CRITICAL`.
- These actions are independent of those performed by the `Fleet Coordinator`.

6.6. MQTT Debugger (`mqtt_debugger.py`)

Writes in the console every message that is transferred through the relevant MQTT topics. With the new structure, it should subscribe to `warehouse/{GroupID}/#` and `{GroupID}/internal/#`. The information should be displayed as: `[time]: [topic]:[message]`.

6.7. Storage

An InfluxDB database will be used to store data. This database must be created in InfluxDB Cloud (https://www.influxdata.com/). The database should hold all necessary values, including: Robot status (location, battery, state), Shelf status (stock levels), Task commands, and Alert messages.

6.8. Control Dashboard

Grafana will be used to create a control dashboard. This dashboard must be created in Grafana Cloud (https://grafana.com). The data analysis should include two separate sections:

- **Section 1:** Fleet Overview (Summary Dashboard)
  - **Robot Status Panel:** A table or list showing all four (4) robots, their current `status`, and `battery` level. This should allow an operator to easily understand the fleet's performance.
  - **Active Alerts:** A panel that shows the latest `CRITICAL` alerts from the `System Monitor`.

- **Section 2:** Detailed Dashboard
  - A dashboard variable (dropdown menu) to select a `robot_id`.
  - **Gauges/Graphs for selected robot**: `Battery` level, `Status Timeline` (using a State timeline panel to show `IDLE`, `MOVING`, etc. over time).
  - **Task History Panel:** A table panel that queries and displays the history of all *Task commands* (e.g., `MOVE`, `PICKUP`) sent to the selected robot, showing the command and its timestamp.
  - A dashboard variable to select a `zone_id` and an `asset_id`.
  - **Stock History Graph**: A time-series graph showing the `stock` for the selected shelf.

7. Project Delivery

This section describes the delivery conditions and submission policy for the 2 project parts.

- **Delivery conditions.** The project should be completed in pairs of two students. Although it is a group project, each student will receive their own grade based on their individual performance during the final defense. The Python programming language must be used.
- **Project submission.** Each group of students must create a `.zip` file containing all the necessary files for their project to work.
  - **Submissions via email will not be accepted!**
  - Include all source and configuration files needed. Do not include any files not needed for the execution of the programs.
  - For the second delivery also add a report (**maximum 3-page A4 in .pdf format**) that includes information about the data sent to the MQTT broker, stored in the database, and the analysis made to the data (dashboard screenshots are required).
  - The final `.zip` file must be submitted to **Inforestudante** (only one `.zip` file per group).
- **Submission dates:**
  - 02/Dec (First part)
  - 14/Dec (Second part)

8. Evaluation

This practical assignment is worth **eight (8) points** of the final grade. The evaluation will consist of two parts.

First part (2 points – 25%):

The first part will include the simulation of devices from the warehouse and the MQTT debugger that will monitor all MQTT transfers (Table 3). The devices should send all the information to the MQTT server.

Module	Value
Warehouse Simulators (AMR + Shelves)	10%
MQTT Debugger	5%
Storage (InfluxDB)	10%
Total	**25%**

Table 3. Evaluation criteria

Second part (6 points – 75%):

The second part will include all the remaining modules of the project (Table 4). Take into account that **the modules for the second part of the project require those from the first part**. Even if they are not ready for the first defence, they must be for the second.

The final grade will be weighted according to the individual defence of the project. **Not being present for the final defence will result in a grade of 0 for that student.**

Students may be asked during the defence to modify some of the data-processing features or the graphics displayed on the dashboard.

Module	Value
Fleet Coordinator (Task Logic & UDP Server)	20%
Warehouse Gateway (Aggregation & Encoding)	20%
Control Dashboard (Grafana)	20%
UDP Socket (Server/Client & Injector)	10%
System Monitor (Health Logic & UDP Client)	5%
Total	**75%**

Table 4. Evaluation criteria