

Project Report: MCP-Based Sensor Gateway with LLM Agent

Author: Group 2

February 6, 2026

Abstract

The following report details the design and implementation of an MCP-based sensor gateway with a LLM agent. This design is capable of putting together sensor protocols to a Model Context Protocol (MCP) interface. The system utilizes a Raspberry Pi 5 to ingest sensor data, buffer it locally, and expose it via an MCP server. Furthermore, a local Large Language Model agent (Ollama) interacts with a gateway to perform natural language queries and to control actuators. The system is designed to be reliable, ensuring that the LLM can only interact with the physical devices through clearly defined tools and does not hallucinate responses.

1 Introduction

IoT systems often consist of many different devices that use different communication protocols and data formats. This makes it difficult to combine sensor data and control devices in a simple and understandable way. By using Large Language Models (LLMs) it is possible to provide an intuitive way for users to interact with systems using natural language instead of technical commands. The primary objective of this project was to design and implement an edge IoT gateway. The system is designed to collect sensor data from various devices with sensors and expose them through a Model Context Protocol (MCP) interface. This architecture also allows a local Large Language Model (LLM) to query sensors and control actuators using natural language, which is implemented using clean API design and without hallucination.

2 System Architecture

2.1 Hardware Components

Our system is built with the following hardware components:

- **Edge Gateway:** Raspberry Pi 5. Responsible for sensor ingestion and hosting the MCP server.
- **LLM Agent:** A laptop running the Ollama LLM client.
- **BLE Sensors:** Nordic Thingy:53 providing environmental data.
- **Wi-Fi Device:** ESP32 acting as both a sensor and an actuator.

2.2 Software Architecture

The system architecture follows a layered design with clear separation of concerns. The architecture of our project is as follows:

- **Sensor Ingestion Layer:** Handles raw communication with the Nordic Thingy:53 (BLE) and ESP32 (Wi-Fi).
- **Data Buffering:** A local buffer (implemented with SQLite) stores the last 5 minutes of data.
- **MCP Server:** A Starlette-based server on the Raspberry Pi that exposes JSON-RPC endpoints in order to communicate with our LLM Agent.

- **LLM Client:** An agent running on the laptop that converts natural language into MCP tool calls.

2.3 Data Flow

Data moves through the system from the physical sensors, to the user and back to the actuators as described:

- **Data Generation and Measuring:** The ESP32 and the Nordic Thingy:53 continuously generate sensor data. The ESP32 measures temperature using a DHT11 sensor and the Thingy:53 measures temperature and humidity using its built in sensors.
- **Data Transmission:** Sensor data is transmitted to the Raspberry Pi using two different protocols based on the data source. The ESP32 sends temperature readings to the Raspberry Pi via HTTP POST requests and the Thingy:53 sends data via Bluetooth Low Energy (BLE) notifications, which are received by the Raspberry Pi using the Bleak library.
- **Data Ingestion and Storage:** The Raspberry Pi ingests incoming sensor data and stores it in a local SQLite database. Each reading is stored together with a timestamp and a sensor identifier. To limit storage usage, a ring buffer mechanism deletes readings that are older than five minutes.
- **Data Processing:** The MCP server reads the data from the database and exposes five tools which can be queried using natural language.

3 Implementation Details

3.1 ESP32: DHT11 Sensor and Actuator Logic

The ESP32 acts as a dual-purpose node. It measures ambient temperature using a DHT11 sensor and provides a remote-controlled status LED.

Sensor Integration: The DHT11 is connected to GPIO 4. The firmware is developed in C++ using the Arduino framework and the Adafruit DHT library. Data is sampled every 2 seconds and transmitted via an HTTP POST request to the Raspberry Pi gateway.

Actuator Circuit: For the actuator component, a standard low-power LED is used as a status indicator. The LED is connected directly to the ESP32 GPIO 2. To ensure safe operation and prevent excessive current draw from the ESP32, a 560Ω current-limiting resistor is placed in series with the LED. This configuration allows the ESP32 to drive the LED directly without the need for external switching components like transistors.

The ESP32 runs a local WebServer that listens for incoming commands from the gateway. A GET request to `/set?val=1` sets GPIO 2 to HIGH, turning the LED on, while `val=0` sets it to LOW, turning it off.

3.2 Thingy:53: Bluetooth Low Energy Integration

The Thingy:53 runs a Zephyr-based firmware that exposes an Environmental Sensing Service (ESS). To enable Bluetooth Low Energy communication and sensor data acquisition, custom firmware was flashed onto the Nordic Thingy:53. The project was built in Visual Studio Code using the nRF Connect extension as shown during the lecture. It utilizes the on-board BME680 sensor to capture temperature and humidity data. The device advertises as `Thingy_Sensor`. To ensure reliability, the firmware uses the `bt_gatt_notify` function to push data to connected clients as soon as a new sample is ready.

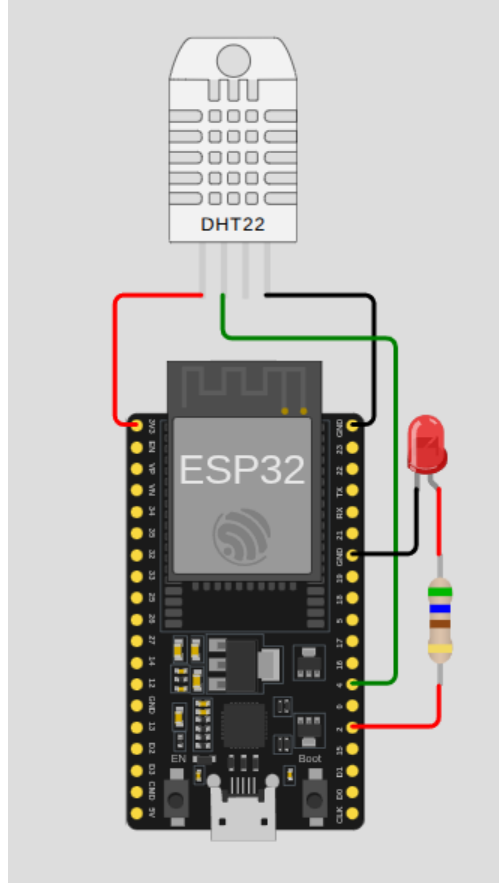


Figure 1: Wiring diagram of the ESP32 with DHT11 and the direct-drive LED circuit.

3.3 Raspberry Pi: The Ingestor, Shared State and MCP Server

The Ingestor is the backbone of the gateway. It performs three critical tasks:

- **HTTP Ingestion:** A Flask-based endpoint (`/ingest`) receives JSON payloads from the ESP32. It records the ESP32's IP address to a persistent configuration file (`esp32.conf`) using absolute paths to ensure the MCP server can always locate the device.
- **BLE Client:** Using the `Bleak` library, the ingestor scans for the Thingy:53, connects to it, and subscribes to notifications for the ESS characteristics.
- **Database Management:** All readings are stored in an SQLite database (`sensor_data.db`). To prevent storage overflow on the Raspberry Pi, a "Ring Buffer" logic is implemented, deleting any data older than 5 minutes.
- **MCP Server:** An MCP-based ASGI server is mounted at `mcp` and exposes tools to list/query sensor readings. It reads the sensor readings from the SQLite DB described before. As well as report readings, the mcp server also lists tools to control actuators (e.g., set an ESP32 LED) and report system status. The server is run via Uvicorn. The following tools are currently available:
 - **sensors.list():** This tool returns a list of all available sensors that have produced data recently. The server queries the database for distinct sensors and returns them as a list. This allows the LLM to see which sensors are currently active without making any assumptions about the system configuration.
 - **sensors.latest(sensor id):** This tool retrieves the most recent reading for a specific sensor. The

MCP server queries the database for the latest entry belonging to the given sensor ID and returns both the value and its timestamp.

- **sensors.query(sensor id, window seconds):** This tool allows querying the stored sensor data over a specified time window. The server calculates a time threshold based on the provided window size and returns all sensor readings that are newer than that threshold.
- **actuators.set(device id, actuator, value):** This tool is used to control actuators connected to the system. In the current implementation it supports controlling an LED connected to an ESP32 device. When the tool is called the MCP server forwards the command to the ESP32 via an HTTP request. The values accepted by the tool are restricted to "ON" and "OFF".
- **system.status():** This tool provides an overview of the current system state. It checks whether the database is accessible and whether recent sensor data has been received from both the Thingy:53 and the ESP32. Device availability is determined by comparing the timestamp of the most recent sensor reading to the current time.

3.4 LLM Client

- **MCP Server Init:** The client initialises the connection to the MCP server on the Raspberry Pi and gets a list of tools from the MCP server
- **OpenAI Client:** To interact with our Ollama model, the client first converts the fetched tools into an OpenAI Tool schema. It then creates a connection to the local LLM model run on our Laptop.
- **Input Processing:** The program reads user input from the Stdin. It then passes that input as well as the available tools to the local model.
- **Process Response:** The LLM model then returns some response and may choose to call tool(s). We execute these tool calls and then append the tool call id and result to the messages. We hand this new set of messages back to the LLM and allow it to further call tools until it no longer wants to call any tools. We then output the final response from the model.

Important to note is that our implementation does not have concept of memory. We process each user input as a new interaction with the AI model. We start a new chat and send the same system prompts on each message.

We do however support multi-tool calling. The agent processes the message and then calls some tools. It then processes the output and can choose to call more tools based on the output. If it does not call a tool, the loop terminates and the output is displayed to the user.

4 Results

In order to test our application, we orientated ourselves on the demo requirements given by the exercise description. All required functionality was successfully implemented and tested. The goal was to support the following natural language queries:

1. What is the current value of sensor X?
2. Show the last 30 seconds of sensor X.
3. Turn the ESP32 LED ON (or OFF).
4. If temperature is above a threshold, turn the LED ON, otherwise OFF.

All of the natural language queries should be processed by our AI agent and it should then use the tools provided by the MCP server to answer.

We will now discuss each case separately.

4.1 What is the current value of sensor X?

We first tried to simply get the result of the temperature sensor on the ESP32. The query is shown in Fig. 2. The model uses the `sensors.latest` tool and returns a temperature of 18.7 degrees.

```
You: What is the current value of the temperature sensor of the esp32?
Agent: Using tool(s)...
- Calling tool: sensors.latest with arguments: {'sensor_id': 'dht11_temp'}
Agent: The current value of the temperature sensor on the ESP32 is 18.7 degrees Celsius.

Tool used: sensors.latest
Result: Returned a tuple containing the latest sensor reading and timestamp.
```

Figure 2: What is the current value of the temperature sensor of the esp32?

We then doubled-checked this by checking the values that our ingest endpoint had received. The agent stated the correct temperature as can be seen in Fig. 3.

```
192.168.0.60 - - [06/Feb/2026 10:12:18] "POST /ingest HTTP/1.1" 200 -
[Thingy] Temp: 18.57°C
[Thingy] Humid: 65.79%
[ESP32] dht11_temp: 18.7
```

Figure 3: Ingest Log

4.2 Show the last 30 seconds of sensor X.

Next we attempted to show the last 30 seconds of sensor data from a device. We wanted to retrieve the temperature data from the Thingy:53. The result can be seen in Fig. 4.

The model tells us that the model is consistently showing us 18.56 degrees as well as outputting the most recent value.

```
You: Show the last 30 seconds of the temperature sensor on the thingy
Agent: Using tool(s)...
- Calling tool: sensors.query with arguments: {'sensor_id': 'Temperature', 'window seconds': 30}
Agent: The last 30 seconds of the temperature sensor on the Thingy53 show a consistent reading of approximately 18.56°C. The most recent reading is 18.55°C at '2026-02-06 10:14:01'.
```

Figure 4: Show the last 30 seconds of the temperature sensor of the thingy.

4.3 Turn the ESP32 LED ON (or off).

A simple command to turn on the ESP32 light also executes as to be expected. The model uses the `actuator.set` command and turns on the light. See Fig. 5.

```
You: Turn the ESP32 LED ON
Agent: Using tool(s)...
- Calling tool: actuators.set with arguments: {'value': 'ON', 'device_id': 'ESP32', 'actuator': 'LED'}
Agent: I called 'actuators.set' with parameters: actuator='LED', device_id='ESP32', and value='ON'. The result was: {'message': 'Actuator updated successfully'}.

The ESP32 LED is now ON.
```

Figure 5: Turn the ESP32 Led on.

4.4 If temperature is above a threshold, turn the LED ON, otherwise OFF.

Lastly we tried to give the LLM a more complex command. We want a light to turn on if a given condition is met. In our case the model first has to check whether the temperature is above 19 degrees. If it is it must then turn on a light. Otherwise it must turn it off.

This is an example of multiple tool calls. It first has to call `sensors.latest` to get the current temperature and then based on this information make another tool call to `actuator.set`. As can be seen in Fig. 6, it accomplishes just this.

```
You: If temperature is above 19, turn the LED ON, otherwise OFF.
Agent: Using tool(s)...
  - Calling tool: sensors.latest with arguments: {'sensor_id': 'Temperature'}
  - Calling tool: actuators.set with arguments: {'device_id': 'ESP32', 'actuator': 'LED', 'value': 'OFF'}
Agent: To determine the current temperature and turn the LED on or off accordingly, I called the 'sensors.latest' tool to get the latest temperature reading.
The result was (18.55, '2026-02-06 10:20:21.439282'), which means the current temperature is 18.55 degrees Celsius.
Since the temperature is below 19, I called the 'actuators.set' tool with the LED value set to 'OFF' to turn off the LED.
The result of the 'sensors.latest' call was (18.55, '2026-02-06 10:20:21.439282').
```

Figure 6: If the temperature is above 19, turn the Led on, otherwise off.

5 Conclusion

We implemented an MCP-based sensor gateway with LLM agent. We used a Raspberry Pi 5 to ingest sensor data and also host a Model Context Protocol (MCP) server. Sensor data came from an ESP32 and a Thingy:53. We interacted with the LLM agent using a custom cli interface that initialises the connection to the MCP server.

We tested our application according to the requirements and received some good results. Initially we had to play around with the specific system prompts to enforce tool usage for tasks and also stop it from trying to call tools that did not exist. In the end, natural language commands such as checking the temperature or turning the LED on and off were correctly translated into MCP tool calls. The system was stable during continuous operation and handled repeated sensor updates without issues.

The biggest challenge was the multi-tool call requests. Most of the time it would work fine but sometimes it would only half work. It would correctly call the sensor tool to get the temperature and evaluate the condition. It would correctly identify the value of the condition but not actually call the actuator tool.

An interesting outcome of this project is the demonstration that Large Language Models can be safely and reliably used to interact with physical systems by strictly constraining their capabilities. Instead of giving the LLM direct access to the sensors or actuators, all interactions are routed through explicit MCP tools. This design prevents hallucinated responses, validates the user input and ensures that every performed system operation is traceable and predictable. As a result, natural language commands such as querying sensor data or controlling an LED can be executed reliably without compromising system safety.