Data Integration

- MQTT Integration for Sensor Data
- MeshCore Sensor Data Integration
- Building a Mesh Weather Station Network

MQTT Integration for Sensor Data

MQTT Integration for Sensor Data

Meshtastic includes a built-in MQTT bridge that can publish all mesh traffic - including telemetry sensor packets - to an MQTT broker. This enables real-time integration with Home Assistant, InfluxDB, Grafana, and other data platforms without any custom firmware modifications.

Enabling the MQTT Bridge on a Gateway Node

The MQTT bridge runs on any Meshtastic node with a direct internet connection. **JSON output over MQTT is only supported on ESP32 gateways** (such as the T-Beam or Heltec V3); it is **not** supported on the nRF52 platform (RAK4631, etc.), so an nRF52 node cannot itself be the JSON-MQTT gateway - use an ESP32 gateway or a host-side bridge. Configure the bridge via the Meshtastic CLI or the Python API:

```
meshtastic --set mqtt.address mqtt.example.com
meshtastic --set mqtt.username meshuser
meshtastic --set mqtt.password s3cr3t
meshtastic --set mqtt.root msh/region/us-east
meshtastic --set mqtt.enabled true
meshtastic --set mqtt.json_enabled true # publish decoded JSON, not raw protobuf
meshtastic --set mqtt.tls_enabled true # strongly recommended for production
```

Uplink and downlink are configured per-channel, e.g. `meshtastic --ch-index 0 --ch-set uplink_enabled true`.

The gateway node will now publish received packets to the topic `msh/<REGION>/2/json/<CHANNEL_NAME>/<USER_ID>` (the protobuf equivalent is `msh/<REGION>/2/e/<CHANNEL_NAME>/<USER_ID>`). Note that the 4th segment is the **channel name** and the final segment is the gateway's **user/node ID** - the packet type (e.g. telemetry) is **not** a topic segment; it is carried in the `type` field inside the JSON body.

JSON Packet Format

A typical decoded telemetry JSON payload looks like this. Telemetry fields are flat, snake_case, under `payload`:

```
{
 "from": 1234567890,
 "to": 4294967295,
 "type": "telemetry",
 "payload": {
 "temperature": 22.4,
 "relative_humidity": 58.2,
 "barometric_pressure": 1013.7,
 "air_util_tx": 0.4
 },
 "timestamp": 1714500000,
 "channel": 0,
 "sender": "!499602d2"
}
```

The documented top-level fields are `id`, `channel`, `from`, `payload`, `sender`, `timestamp`, `to`, and `type`. There are **no** top-level `rssi`/`snr` fields in Meshtastic's JSON output - if you need signal metrics, add them with a downstream processor.

Node-RED Integration

Node-RED provides a low-code flow editor ideal for parsing and routing Meshtastic telemetry. A typical flow consists of:

1. **MQTT In node** - subscribe to `msh/+/2/json/#` (do not subscribe to bare `msh/#`, which also catches the binary `2/e/` topics that the JSON parser chokes on), then filter on the JSON `type` field == `"telemetry"` in a function node, since the portnum is not a topic level
2. **JSON node** - parse the payload string to a JavaScript object
3. **Function node** - extract fields, add tags (node name, location)
4. **InfluxDB Out node** or **Home Assistant node** - write the data to your chosen store

Home Assistant MQTT Sensor Configuration

Add the following to your Home Assistant `configuration.yaml` to create sensors for a Meshtastic node named `SensorNode-01`. Set the `state_topic` to match your actual `msh/<REGION>/2/json/<CHANNEL_NAME>/<USER_ID>` topic, or use a wildcard such as `msh/+/2/json/+/+` and filter in the `value_template`:

```
mqtt:
 sensor:
 - name: "SensorNode-01 Temperature"
 state_topic: "msh/+/2/json/+/+"
 value_template: "{{ value_json.payload.temperature }}"
 unit_of_measurement: "°C"
 device_class: temperature
 - name: "SensorNode-01 Humidity"
 state_topic: "msh/+/2/json/+/+"
 value_template: "{{ value_json.payload.relative_humidity }}"
 unit_of_measurement: "%"
 device_class: humidity
 - name: "SensorNode-01 Pressure"
 state_topic: "msh/+/2/json/+/+"
 value_template: "{{ value_json.payload.barometric_pressure }}"
 unit_of_measurement: "hPa"
 device_class: pressure
```

Restart Home Assistant after editing the configuration. The sensors will populate with live data as new telemetry packets arrive via the MQTT bridge.

Grafana Dashboard for Long-Term Trending

For historical analysis, write telemetry data to InfluxDB (v2 recommended) and visualise with Grafana:

1. Create an InfluxDB bucket named `mesh_telemetry`.
2. Use the Node-RED InfluxDB Out node (or the Python `influxdb-client` library) to write measurements with tags `node_id` and `node_name`.
3. In Grafana, add InfluxDB as a data source and create a dashboard with time-series panels for temperature, humidity, and pressure. Use a 7-day or 30-day range to identify trends, calibration drift, or equipment failure.
4. Configure Grafana alerting to notify via email or Slack when temperature exceeds a threshold or a sensor node stops reporting (absence of data alert).

MeshCore Sensor Data Integration

MeshCore Sensor Data Integration

MeshCore supports environmental sensors, but the capability is more limited than a polished, deploy-and-forget telemetry system. Sensor support is a **compile-time feature** (the firmware's environment-sensor manager, enabled by build flags and example builds such as `simple_sensor` / `SensorMesh`) on standard MeshCore firmware roles (Companion, Repeater, Room Server) — there is no separate "SENSOR" firmware variant or role. Telemetry is delivered through MeshCore's **binary request/response protocol** (CayenneLPP-encoded, permission-gated): a client asks a node for its telemetry and the node replies. Readings are **not** broadcast on a schedule as free-floating packets, and the room server does **not** log structured sensor lines. This page describes the real capture path using the `meshcore_py` Python library, then how to write the decoded values to InfluxDB or CSV for analysis.

Note: parts of MeshCore's sensor/telemetry support are evolving and build-dependent. Confirm a sensor-capable build and the telemetry permission settings for your node before relying on the workflow below. If your firmware build does not include sensor support, telemetry capture is **not currently available** on that node.

How Sensor Nodes Behave

A MeshCore node running standard firmware *built with sensor support enabled* reads its attached I2C sensors (for example a BME680 or SHTC3) and exposes the readings as **telemetry**. Rather than waking on a timer and broadcasting a reading to everyone, the node serves telemetry on request: a connected or in-range client issues a telemetry request and the node returns a CayenneLPP-encoded response. Access is permission-gated, so only authorised clients receive the data.

Because telemetry is request/response traffic, it is **addressed** rather than broadcast. It travels to and from the node over learned routes (falling back to flooding only when a path is not yet known), routed like other directed MeshCore messages — not as unsolicited broadcasts that "any node can decode." To collect readings continuously, you run an always-on client (a host PC or single-board computer connected to a MeshCore node over USB serial, TCP, or BLE) that polls each sensor node and logs the responses.

Sensor Data Flow Through the Mesh

1. A sensor node reads its BME680 (illustrative values: T=21.8°C, H=62.3%, P=1011.4 hPa, IAQ=42) and holds the latest reading as telemetry.
2. A collector client sends a telemetry **request** to the node (in `meshcore_py`, via `commands.req_telemetry()` for a remote contact, or `commands.get_self_telemetry()` for the locally attached node).
3. The request and the response are routed hop-by-hop over the mesh as addressed traffic between the collector and the sensor node.
4. The node replies with a **CayenneLPP-encoded telemetry response** (channel + LPP data type + value per metric), which the client decodes.
5. The collector writes the decoded values to a time-series store (InfluxDB) or CSV.

Aggregation and visualisation are done by these external tools (for example a custom Python collector, or community projects such as MeshMonitor) — the stock room server is a store-and-forward BBS, not a per-node sensor dashboard.

Capturing Telemetry with meshcore_py

The supported way to capture MeshCore telemetry programmatically is the `meshcore_py` library, running on a host connected to a MeshCore node over serial, TCP, or BLE. You connect, request telemetry (or subscribe to telemetry responses), and decode the CayenneLPP payload. There is no structured `SENSOR ...` stdout log on the room server to scrape — capture happens through the protocol, not by parsing console text.

```
import asyncio
from meshcore import MeshCore, EventType

# Connect to a locally attached MeshCore node (also: create_tcp, create_ble)
async def main():
    mc = await MeshCore.create_serial("/dev/ttyUSB0")

    # Subscribe to telemetry responses as they arrive
    def on_telemetry(event):
        # event.payload contains the decoded CayenneLPP telemetry
        # (channel + LPP type + value per metric)
        print("telemetry:", event.payload)

    mc.subscribe(EventType.TELEMETRY_RESPONSE, on_telemetry)

    # Read the locally attached node's own sensors
```

```
    await mc.commands.get_self_telemetry()

    # Or request telemetry from a remote contact by key/name
    contacts = await mc.commands.get_contacts()
    # await mc.commands.req_telemetry(some_contact)

    await asyncio.sleep(60)

asyncio.run(main())
```

The decoded telemetry arrives as structured CayenneLPP fields (temperature, humidity, pressure, battery voltage, etc.), keyed by LPP channel and data type — not as a regex match over a text log line. Map those decoded fields to your own record dict before writing to a store. (On a node, the device-side CLI exposes only `sensor list`, `sensor get <key>`, and `sensor set <key> <value>` — a generic key/value store, not a live `sensor read` command.)

Writing to InfluxDB

Once you have a record dict of decoded telemetry values, write it to InfluxDB using the line protocol. Compute an epoch-nanosecond timestamp from the time the reading was captured (or omit the explicit timestamp and let InfluxDB assign the write time):

```
import time
from influxdb_client import InfluxDBClient, WriteOptions

INFLUX_URL = "http://localhost:8086"
INFLUX_TOKEN = "your-token-here"
INFLUX_ORG = "meshamerica"
INFLUX_BUCKET = "mesh_sensors"

client = InfluxDBClient(url=INFLUX_URL, token=INFLUX_TOKEN, org=INFLUX_ORG)
write_api = client.write_api(write_options=WriteOptions(batch_size=1))

def write_record(rec):
    # rec holds decoded telemetry values from meshcore_py
    # InfluxDB line protocol; timestamp in epoch nanoseconds
    ts_ns = int(time.time() * 1e9)
    point = (
        "environment"
        f",node_id={rec['node_id']}"
```

```
        f" temperature={rec['temperature']},"
        f"humidity={rec['humidity']},"
        f"pressure={rec['pressure']},"
        f"iaq={rec['iaq']},"
        f"battery_mv={rec['battery_mv']}"
        f" {ts_ns}"
    )
    write_api.write(bucket=INFLUX_BUCKET, record=point)
```

Follow the InfluxDB line protocol rules for escaping tag and field values, and make sure the values feeding this function come from a real telemetry source (the `meshcore_py` path above).

Writing to CSV for Data Science Workflows

For ad-hoc analysis with pandas or R, a CSV sink is often more convenient than a full time-series database:

```
import csv, pathlib

CSV_PATH = pathlib.Path("/var/log/mesh_sensors.csv")

def append_csv(rec):
    write_header = not CSV_PATH.exists()
    with CSV_PATH.open("a", newline="") as f:
        w = csv.DictWriter(f, fieldnames=rec.keys())
        if write_header:
            w.writeheader()
        w.writerow(rec)
```

Load into pandas for analysis: `df = pd.read_csv("/var/log/mesh_sensors.csv", parse_dates=["time"])`. Standard pandas operations (resampling, rolling averages, correlation with external weather data) apply directly.

Integration with Python Data Science Tools

- **pandas** - Resample to hourly/daily averages, detect anomalies, compute sensor drift over time.
- **matplotlib / seaborn** - Plot temperature and humidity heatmaps across a sensor grid.

- **scikit-learn** - Train a simple regression to predict indoor temperature from outdoor readings. Useful for HVAC optimisation use cases.
- **Jupyter Notebook** - Combine data ingestion, analysis, and visualisation in a reproducible, shareable format ideal for community reporting.

Building a Mesh Weather Station Network

Building a Mesh Weather Station Network

A neighbourhood weather monitoring grid built on mesh radio nodes provides hyper-local environmental data at a fraction of the cost of commercial weather station networks. This page presents a deployment blueprint: hardware selection, node placement strategy, data aggregation, and community value proposition.

A note on telemetry capture. The realistic data path for an automated mesh weather grid today is **Meshtastic**, whose Telemetry module reports environment metrics (temperature, humidity, barometric pressure, gas resistance/IAQ, voltage, current) that an ESP32 gateway can publish to MQTT and into a database. MeshCore sensor support exists but is build-time and request/response only (see the MeshCore Sensor Data Integration page); it has no scheduled-broadcast sensor packets and no room-server sensor log to scrape. The hardware list below works for either firmware, but choose your aggregation method (below) to match the firmware you actually flash.

Use Case and Goals

The target deployment is a 5-node network covering a suburban neighbourhood approximately 2 km in diameter, providing temperature, humidity, barometric pressure, and rainfall (with optional tipping-bucket rain gauge) at 15-minute intervals. The base station node aggregates data and pushes to a local dashboard and optionally to Weather Underground as a Personal Weather Station (PWS) network contribution.

Hardware List (5-Node Network)

Item	Qty	Notes
RAK19007 Base Board	5	One per node
RAK4631 Core Module	5	nRF52840 + SX1262

Item	Qty	Notes
RAK1906 BME680 Sensor	5	Temp/humidity/pressure/IAQ
0.5 W solar panel	4	Remote nodes; base station uses mains power. 0.5 W is marginal in low-sun regions — size to local insolation and duty cycle (see note below).
3000 mAh LiPo battery	4	Remote node backup power. Add an in-line fuse on the positive lead and a charge controller with low-temperature charge cutoff (see safety note below).
Weatherproof enclosure (e.g. RAK Unify or equivalent IP-rated box)	5	Use an outdoor-rated enclosure with mounting bracket; verify the exact SKU's IP rating against the manufacturer datasheet before buying.
915 MHz fiberglass antenna (3 dBi)	5	~3 dBi over isotropic — a modest gain over a true half-wave dipole (~2.15 dBi), but a worthwhile upgrade over a stock rubber-duck whip.
Raspberry Pi 4 (base station)	1	Runs the data pipeline (MQTT broker / collector + database + dashboard)

The RAK4631/RAK19007 is configured over its onboard USB-C connector, which already provides a serial console — no separate USB-C-to-UART adapter cable is required for normal setup.

Battery safety (outdoor LiPo): outdoor nodes need an in-line fuse (or polyfuse) on the battery positive lead, and a charge controller/PMIC with a **low-temperature charge cutoff** so the pack is never charged below 0 °C (32 °F). Charging a lithium cell below freezing causes lithium plating — permanent capacity loss and a fire risk. A bare TP4056 has no such cutoff. This applies to LiFePO4 as well.

Solar sizing: a 0.5 W panel may be insufficient in low-sun months or cloudy climates. Size the panel and battery to your worst-month insolation and the node's actual duty cycle rather than assuming a fixed wattage carries a node year-round.

Node Placement Strategy

LoRa range depends heavily on terrain and obstructions. For a neighbourhood grid targeting 1 - 2 km node spacing:

- **Hilltops and ridge lines** - Priority placement. Raising an antenna extends the radio horizon: the line-of-sight distance to the horizon is roughly 4.12 × √h km for antenna height h in metres (about 13 km at 10 m, 16 km at 15 m for a flat earth), and the usable

link is the sum of both ends' horizons. A rooftop-mounted node on a two-storey building often outperforms a ground-level hilltop node.
- **Open areas** - Parks, schoolyards, and sports fields with no obstructions in the LoRa Fresnel zone are ideal secondary sites.
- **Avoid dense urban canyons** - Buildings attenuate 868/915 MHz signals significantly, from a few dB for a single drywall partition to 20+ dB for reinforced concrete per wall. Place nodes at building edges or on roof parapet walls rather than interior courtyards.
- **1 - 2 km spacing** - With omnidirectional 3 dBi antennas and SF10 spreading factor, links around 1.5 km are achievable in favourable suburban line-of-sight conditions; obstructions, foliage, and terrain reduce this. Test with field RSSI measurements before finalising locations.

Use RF planning tools such as HeyWhatsThat or Radio Mobile to model coverage before physically deploying hardware.

Data Aggregation at the Base Station

The aggregation method depends on the firmware you flash:

- **Meshtastic (recommended for automated weather telemetry):** an ESP32 gateway node publishes telemetry to an MQTT broker (JSON output is supported on ESP32 gateways, not on nRF52). A collector subscribes to the broker and writes the readings to InfluxDB. Note the RAK4631 is an nRF52 board, so it cannot itself emit JSON-MQTT — pair it with an ESP32 gateway or a host-side bridge for the base-station role.
- **MeshCore:** there is no room-server sensor log to parse. Use the MeshCore Sensor Data Integration path — a host running `meshcore_py` requests telemetry from each node (CayenneLPP response) and writes the decoded values to InfluxDB.

A Grafana instance on the same Raspberry Pi can provide the neighbourhood dashboard, accessible via a local web browser or optionally published to the internet via a Cloudflare Tunnel for remote access without port-forwarding.

Sample Grafana panels for the weather station dashboard:

- Map panel showing node locations with colour-coded current temperature
- Time-series panel with all 5 node temperatures overlaid (last 24 hours)
- Humidity and pressure time-series with storm-front detection annotation
- Battery voltage panel to flag nodes needing maintenance
- Uptime table showing last-seen timestamp per node

Comparison with Weather Underground PWS Network

Weather Underground's Personal Weather Station programme allows individuals to contribute data to a public map. A mesh weather station network is complementary rather than competing:

- **Mesh advantage** - No internet connectivity required per node; data flows over radio. A single internet-connected base station is sufficient for the whole neighbourhood.
- **PWS contribution** - Optionally forward base station data to Weather Underground using the WU API to contribute to the public network and gain access to WU dashboard tools.
- **NOAA CoCoRaHS comparison** - CoCoRaHS focuses on manual rain gauge readings reported daily. An automated mesh network provides sub-hourly data and adds temperature, humidity, and pressure. The two approaches are complementary; mesh data can supplement manual CoCoRaHS reports for the same location.

Community Value Proposition

A neighbourhood mesh weather station network provides tangible community benefits beyond individual weather curiosity:

- **Urban heat island mapping** - Identify which streets or parks run significantly hotter in summer, informing tree-planting and shade-structure decisions.
- **Frost and freeze alerts** - Gardeners and small farmers can get a useful hyperlocal indication of frost or freeze from the nearest sensor node, which may differ from official forecasts based on airport weather stations kilometres away. Treat a single uncalibrated node as supplementary — cross-check against the National Weather Service forecast before acting on it.
- **Flood and drainage monitoring** - Nodes near drainage channels can trigger alerts on rapid barometric pressure drops correlated with heavy rain events.
- **Resilience during grid outages** - Solar-powered mesh nodes continue operating when mains power fails, providing situational awareness during severe weather events precisely when it is most needed.
- **Educational resource** - Open data from a neighbourhood sensor grid makes a compelling school science project, with real local data available for analysis.