

Data Pipelines and Logging

- [MQTT to InfluxDB and Grafana](#)
- [MeshCore Sensor Nodes](#)
- [Field Sensor Deployment Guide](#)

MQTT to InfluxDB and Grafana

Architecture Overview

The full telemetry pipeline runs: **Meshtastic node** → **MQTT broker** → **Telegraf (or Python subscriber)** → **InfluxDB 2.x** → **Grafana dashboard**. Each component is independently replaceable, so you can swap Telegraf for a custom Python script without touching InfluxDB or Grafana. This pipeline requires the gateway to publish decoded JSON (`mqtt.json_enabled true`); note that JSON-over-MQTT is not supported on nRF52 gateways (RAK4631, etc.) - use an ESP32 gateway.

InfluxDB 2.x Setup

Install InfluxDB 2.x on a server or Raspberry Pi:

```
# Debian / Ubuntu / Raspberry Pi OS
wget https://dl.influxdata.com/influxdb/releases/influxdb2-2.7.1-amd64.deb
sudo dpkg -i influxdb2-2.7.1-amd64.deb
sudo systemctl enable --now influxdb
```

After installation, visit `http://<host>:8086` to complete the setup wizard. Create an **organisation** (e.g. `meshamerica`), a **bucket** (e.g. `meshtastic`), and generate an **API token** with write access to that bucket. Store the token - you will need it for both Telegraf and the Python client.

Telegraf MQTT Consumer

Telegraf is a convenient path from MQTT to InfluxDB. Install it, then add a stanza to `/etc/telegraf/telegraf.conf`. Subscribe only to the JSON subtree - the bare `msh/#` wildcard also catches the binary protobuf `2/e/` topics, which the JSON parser cannot decode:

```

[[inputs.mqtt_consumer]]
  servers = ["tcp://localhost:1883"]
  topics = ["msh/+ /2/json/#"]
  qos = 0
  data_format = "json_v2"

  [[inputs.mqtt_consumer.json_v2]]
  [[inputs.mqtt_consumer.json_v2.object]]
  path = "payload"
  tags = []
  # flattens the nested payload object (temperature, relative_humidity, ...)

[[outputs.influxdb_v2]]
  urls = ["http://localhost:8086"]
  token = "YOUR_INFLUXDB_TOKEN"
  org = "meshamerica"
  bucket = "meshtastic"

```

Meshtastic's JSON nests the numeric metrics under a `payload` object, so a plain `data_format = "json"` will not reach temperature/humidity by default - you must use the `json_v2` parser (or a path/query) to flatten `payload.*`, or use the Python subscriber below for full control. Ensure `mqtt.json_enabled true` is set on the gateway. Restart Telegraf after any config change: `sudo systemctl restart telegraf`.

Python Subscriber (Alternative)

If you need to apply custom logic, use a Python subscriber instead. For a JSON workflow (`mqtt.json_enabled true`) the gateway already publishes decoded JSON, so no protobuf module is needed - parse the JSON directly. (If you ever do need the protobuf `ServiceEnvelope`, the current import path is `from meshtastic.protobuf import mesh_pb2`.)

```
pip install paho-mqtt influxdb-client
```

```

import paho.mqtt.client as mqtt
from influxdb_client import InfluxDBClient, Point
from influxdb_client.client.write_api import SYNCHRONOUS
import json, os

INFLUX_URL = "http://localhost:8086"

```

```

INFLUX_TOKEN = os.environ["INFLUX_TOKEN"]
INFLUX_ORG = "meshamerica"
INFLUX_BUCKET = "meshtastic"

client_db = InfluxDBClient(url=INFLUX_URL, token=INFLUX_TOKEN, org=INFLUX_ORG)
write_api = client_db.write_api(write_options=SYNCHRONOUS)

# Optional: map node IDs to long names by caching NODEINFO_APP messages
# (payload.longname keyed by "from"); Meshtastic JSON has no top-level "fromName".
node_names = {}

def on_message(client, userdata, msg):
    try:
        envelope = json.loads(msg.payload)
        node_id = envelope.get("from", "unknown")
        # NODEINFO packets carry the long name in payload.longname
        if envelope.get("type") == "nodeinfo":
            ln = envelope.get("payload", {}).get("longname")
            if ln:
                node_names[node_id] = ln
        return
        if envelope.get("type") != "telemetry":
            return
        node_name = node_names.get(node_id, node_id)
        tel = envelope.get("payload", {}) # telemetry fields are flat, snake_case
        point = (
            Point("telemetry")
            .tag("node_id", node_id)
            .tag("node_name", node_name)
            .field("temperature", tel.get("temperature"))
            .field("humidity", tel.get("relative_humidity"))
            .field("battery_voltage", tel.get("voltage"))
        )
        write_api.write(bucket=INFLUX_BUCKET, record=point)
    except Exception as e:
        print(f"Parse error: {e}")

mqttc = mqtt.Client()
mqttc.on_message = on_message
mqttc.connect("localhost", 1883)

```

```
mqttc.subscribe("msh/+2/json/#")
mqttc.loop_forever()
```

Note: Meshtastic's JSON envelope has **no** top-level `rxSnr`/`rxRssi` (or `snr`/`rssi`) fields, and telemetry metrics are flat snake_case under `payload` (e.g. `relative_humidity`, not `relativeHumidity`). Add signal metrics only if a downstream processor explicitly supplies them.

InfluxDB Line Protocol

Whether you use Telegraf or Python, each telemetry reading lands in InfluxDB as a **measurement** with tags and fields:

```
telemetry,node_id=!abc12345,node_name=ridge-repeater
temperature=22.4,humidity=61.0,battery_voltage=3.91 1746230400000000000
```

- **Measurement:** `telemetry`
- **Tags:** `node_id`, `node_name` (indexed, low-cardinality)
- **Fields:** `temperature`, `humidity`, `battery_voltage` (add `snr`/`rssi` only if a downstream processor supplies them; they are not in the gateway's JSON)

Grafana Setup

Install Grafana on the same host or a separate machine, then add InfluxDB as a data source using the **Flux** query language:

1. Configuration → Data Sources → Add data source → InfluxDB
2. Set Query Language to **Flux**
3. URL: `http://localhost:8086`
4. Paste your InfluxDB org, bucket, and token

Recommended dashboard panels:

- Temperature over time (line graph per node)
- Battery voltage trends (multi-node time series)
- Node count (stat panel - unique `node_id` values in last hour)
- Signal quality map (table of last RSSI/SNR per node)

Example Flux query - temperature trend for last 24 hours:

```
from(bucket: "meshtastic")
  |> range(start: -24h)
  |> filter(fn: (r) => r._measurement == "telemetry" and r._field == "temperature")
  |> aggregateWindow(every: 5m, fn: mean, createEmpty: false)
```

Retention Policies and Downsampling

In InfluxDB 2.x, retention is set per bucket. For raw telemetry data set the bucket retention to **90 days**. Create a second bucket (e.g. `meshtastic_hourly`) with indefinite retention and use a Flux task to downsample:

```
// Task: run every 1h - downsample telemetry to hourly averages
option task = { name: "downsample_telemetry", every: 1h }

from(bucket: "meshtastic")
  |> range(start: -2h)
  |> filter(fn: (r) => r._measurement == "telemetry")
  |> aggregateWindow(every: 1h, fn: mean, createEmpty: false)
  |> to(bucket: "meshtastic_hourly", org: "meshamerica")
```

Alerting

Grafana alerting fires when conditions are met. Useful rules for mesh deployments:

- **Low battery:** `battery_voltage < 3.5` for any node - triggers SMS or email alert.
- **Temperature exceeded:** `temperature > 50` - useful for enclosure health monitoring in summer.
- **Node offline:** no telemetry from a given `node_id` for more than N hours - check the "Last seen" derived field using a Flux `last()` query against the current time.

MeshCore Sensor Nodes

MeshCore Sensor Support

MeshCore does **not** ship a separate "Sensor" firmware variant. MeshCore device firmware is built around three roles - **Companion**, **Repeater**, and **Room Server**. Environmental-sensor support is a **compile-time feature**: it is enabled when the firmware is built with the relevant `ENV_INCLUDE_*` flags (the environment-sensor manager, demonstrated by example builds such as `simple_sensor` / `SensorMesh`). It is an evolving, limited capability, not a polished deployable role you "flash and go."

Because sensor support is selected at build time, there is no stock "RAK4631 SENSOR" release asset to download. To use it you compile a sensor-capable build for your board. nRF52 boards (RAK4631, T-Echo) are flashed via UF2 drag-and-drop or the web flasher; `esptool` applies only to ESP32 boards.

Supported Sensors

When a sensor-capable build is compiled in, the environment-sensor manager probes the I2C bus to detect supported sensors. Commonly referenced hardware includes:

- **BME280** - temperature, humidity, barometric pressure (I2C). A common choice for weather monitoring.
- **BME680** - adds a gas/VOC sensor to the BME280 feature set, useful for indoor air-quality indication. (IAQ requires the Bosch BSEC library and a calibration period to be meaningful.)
- **INA219** - bus voltage and current measurement over I2C. Useful for monitoring battery banks, solar panels, or load/charge current.
- **Custom sensors** - additional inputs require *modifying and recompiling the firmware* (e.g. the `onSensorDataRead` hook plus the appropriate `ENV_INCLUDE_*` flags in `EnvironmentSensorManager`). This is a build-time change, not a runtime configuration option.

How Sensor Telemetry Is Delivered

A common misconception is that MeshCore "broadcasts" sensor readings in its adverts. It does not. A MeshCore **advert** announces a node's presence and routing information only - its name, optional position, and signed public key. Adverts do **not** carry temperature, humidity, or any other sensor

value.

Sensor data travels instead over MeshCore's **binary request/response protocol**. A client explicitly requests telemetry from a node (`REQ_TYPE_GET_TELEMETRY_DATA`); the node replies with a **CayenneLPP**-encoded payload. This exchange is addressed and permission-gated - it is a pull (request/response), not a broadcast that every node receives and forwards. (Adverts themselves are either zero-hop or flooded for routing purposes, but in neither case do they contain sensor readings.)

Power Profile

A telemetry-reporting MeshCore node can run at a low duty cycle, which makes solar or long-term battery deployment feasible. Concrete figures depend heavily on TX power, reporting frequency, whether Bluetooth is disabled, and the board's sleep current, so the numbers below are **rough estimates, not specifications**:

- An nRF52840 node (e.g. RAK4631) with a BME280, reporting infrequently with BLE off and low TX power, can draw on the order of a few mAh/day. Measure your own node's consumption before sizing a battery - do not treat any single figure as a firm spec.
- Battery runtime estimates must also account for self-discharge and capacity derating over time, so a nominal pack capacity does not translate directly into a guaranteed number of days.
- A small (~1 W) solar panel can offset consumption at a site that *reliably* gets 4+ peak-sun-hours, but this varies by season and location and will not keep a node running "indefinitely" - lithium cells still age out over calendar years.
- For the lowest power, disable Bluetooth and set LoRa TX power to the minimum needed to reach the nearest repeater.

Outdoor lithium safety: never charge a lithium cell (including LiFePO4) below 0 C / 32 F. For any outdoor node deployed where winter temperatures drop below freezing, use a charger/controller with a low-temperature charge cutoff, and add an in-line fuse on the battery positive lead.

Configuration

MeshCore is configured over USB/serial or over BLE (via the companion app or the MeshCore CLI). The actual sensor CLI is a small generic key/value store - there is **no** `set sensor type`, `set sensor addr`, `set sensor interval`, or `set sensor enabled` command. The real commands are:

- `sensor list [start]` - list the sensor custom variables (printed as `key=value` lines).
- `sensor get <key>` - read a sensor custom variable.
- `sensor set <key> <value>` - set a sensor custom variable.

Which sensors are present is determined at **compile time** (the `ENV_INCLUDE_*` build flags) and by I2C auto-detection at startup - not by a runtime enable/disable or type-selection command. I2C addresses are detected during the environment sensor manager's bus probe; they are not set from the CLI. Advert cadence is governed by the real interval prefs (e.g. `set flood.advert.interval <hours>` / `set advert.interval <minutes>`), which control advert timing - they do not broadcast sensor readings.

Use Cases

- **Remote weather station:** BME280 on a ridge or mountain peak, reporting temperature, humidity, and pressure on request.
- **Water level monitoring:** an ultrasonic or pressure sensor at a stream crossing or water tank. Treat this as **best-effort situational awareness only** - mesh packets can drop, and a hobbyist sensor must *not* replace official NWS/USGS flood warnings for any safety decision (see the Water Quality and Flood Monitoring page).
- **Air quality indication:** BME680 in an urban or wildfire smoke corridor (low-cost sensors are indicative, not reference-grade).
- **Soil temperature for agriculture:** BME280 buried at root depth in a remote field.
- **Power system monitoring:** INA219 across the shunt resistor of a solar-charged battery bank, reporting **bus voltage and load/charge current**. Note: state of charge is not measured directly - it must be derived (e.g. coulomb counting) from voltage and current.

Integration with Data Pipelines

Capturing MeshCore sensor data is not automatic - readings do not appear by themselves in the app's node list. You need a node connected to a host (a room server, or a serial/USB/BLE bridge) running code that requests telemetry and logs it. Use the real async **meshcore_py** library: create a client, subscribe to telemetry events, and issue telemetry requests through `commands.*` - do not parse adverts for sensor values.

```
import asyncio, sqlite3, datetime
from meshcore import MeshCore, EventType

async def main():
    # create_serial(port), create_tcp(host, port), or create_ble(address)
    mc = await MeshCore.create_serial("/dev/ttyUSB0")

    db = sqlite3.connect("sensors.db")
    db.execute("CREATE TABLE IF NOT EXISTS readings "
              "(ts TEXT, node TEXT, payload TEXT)")
```

```
def on_telemetry(event):
    # event.payload is the decoded CayenneLPP telemetry response
    db.execute("INSERT INTO readings VALUES (?, ?, ?)", (
        datetime.datetime.utcnow().isoformat(),
        str(event.payload.get("from", "")),
        str(event.payload),
    ))
    db.commit()

mc.subscribe(EventType.TELEMTRY_RESPONSE, on_telemetry)

# Pull telemetry from a known contact (request/response, not broadcast):
contacts = await mc.commands.get_contacts()
for contact in contacts.values():
    await mc.commands.req_telemetry(contact)

# keep the asyncio loop running to receive responses
await asyncio.Event().wait()

asyncio.run(main())
```

There is no `MeshCore("/dev/ttyUSB0")` constructor, no `mc.on_advertisement = ...` callback, no `mc.run()`, and no `advert.get("temperature")` - adverts do not carry sensor values. MeshCore has **no built-in MQTT module**; MQTT export is done through an external bridge (meshcore_py-based, or a third-party tool such as MeshMonitor), not by the room server.

A MeshCore Home Assistant integration does exist (github.com/meshcore-dev/meshcore-ha, domain `meshcore`), built on meshcore_py.

Comparison with Meshtastic Telemetry

Feature	Meshtastic Telemetry module	MeshCore sensor support
BME280 / BME680 support	Yes (auto-detected on I2C)	Yes, via built-in environmental telemetry (compiled in)
INA219 support	Yes (via power metrics)	Yes

Feature	Meshtastic Telemetry module	MeshCore sensor support
Data propagation	Mesh packet (Telemetry protobuf)	CayenneLPP telemetry over the binary request/response protocol (pull, not broadcast)
MQTT output	Yes (via Meshtastic MQTT module, ESP32 gateways)	No built-in MQTT; requires an external bridge (e.g. meshcore_py or MeshMonitor)
Dedicated sensor firmware	No (runs on standard node firmware)	No separate variant - sensor support is compiled into the node firmware via <code>ENV_INCLUDE_*</code> flags
Power profile	Low - dominated by TX duty cycle and sleep config	Low - likewise dominated by TX duty cycle and sleep config

Field Sensor Deployment Guide

Site Selection

Place sensors where you need data - not where it is convenient to access. Ideal sites are often inconvenient: a peak for a weather station, a stream bank for water level, a crop row for soil temperature. Choose the site first, then engineer the power and connectivity to support it.

Weatherproofing Sensors

Temperature and humidity sensors require a **radiation shield** (white louvered housing) for accurate readings. In direct sunlight a bare sensor's error can exceed 10 °C, sometimes much more, depending on wind and the sensor (see weather-station siting references). Heat trapped inside a sealed enclosure will do the same. Rules:

- Never seal a BME280 or BME680 inside a closed waterproof enclosure - humidity will read 100 % and temperature will reflect enclosure heat, not ambient air.
- Mount the sensor in a louvered radiation shield. A hobby plastic louvered shield runs roughly \$10-25 (price varies; check a current product listing), while a full traditional Stevenson screen is considerably more expensive.
- If you cannot use a radiation shield, at minimum shade the sensor from direct sun and allow free airflow.

Enclosure Strategy

Keep electronics and sensors in separate compartments:

- Main board, battery, and solar charge controller in an **IP67 sealed enclosure** (ABS or polycarbonate, UV-rated).
- Run sensor wiring through a cable gland or a small hole sealed with self-amalgamating tape.

- BME280 / BME680: mount in the radiation shield outside the enclosure and run I2C wiring inside. Keep I2C cable runs short - under ~50 cm is a useful rule of thumb, ultimately governed by the 400 pF total bus-capacitance limit in the I2C specification (NXP UM10204). For longer runs use an active I2C bus extender/repeater rather than a simple buffer.
- For insect protection, cover any ventilation holes with fine stainless mesh - spiders love warm enclosures.

Power Sizing

Sensor node consumption is low with the right hardware and firmware. The figures below are an idealized best case (they exclude regulator quiescent draw and wake/active current); real nodes often run somewhat higher:

Component	Average current (10-min TX interval)
nRF52840 MCU (sleep)	~2 μ A
BME280 (sleep / active)	~0.1 μ A sleep; ~3.6 μ A active at 1 Hz
LoRa TX burst (10 s/day total)	~0.1 mA averaged (TX current $\times$ airtime $\div$ 86400 s; e.g. ~118 mA at +22 dBm $\times$ ~10 s/day $\div$ 86400 s $\approx$ 0.014 mA — adjust for your actual TX power and airtime)
Total daily	< 5 mAh/day (idealized best case; excludes regulator quiescent and wake/active current)

- **Battery-only:** 3 000 mAh LiPo $\rightarrow$ ~600 days as a theoretical maximum. Derate for LiPo self-discharge and regulator quiescent draw - real runtime will be shorter.
- **Solar-maintained:** a 1 W (6 V) panel can keep a 3 000 mAh pack topped up at sites that reliably get ~4+ peak-sun-hours, but this does **not** hold in every climate - high-latitude winters and shaded/canopy sites can fall short for extended periods. Size conservatively for worst-case winter insolation rather than assuming indefinite operation. Also ensure the battery is not charged below 0 $^{\circ}$ C: use a charge controller with a low-temperature charge cutoff (charging any lithium cell, including LiFePO4, below freezing causes plating and permanent damage).
- For critical sensors in low-light environments (north-facing, dense canopy), upsize to 2 - 3 W and add a 5 000 - 6 000 mAh pack. Tie panel/battery sizing to your actual load budget and local peak-sun-hours (e.g. via PVWatts/ NREL insolation data) rather than fixed numbers.

Connectivity Range

Sensor nodes use the same LoRa mesh relay infrastructure as every other node. A sensor 20 km from the nearest internet gateway can deliver data with low latency when the relay path is healthy, but mesh delivery is best-effort: expect dropped readings and gaps whenever any hop fails (see Data Gaps below). Do not rely on near-real-time delivery for time-critical or safety-of-life monitoring. When planning a sensor deployment, map out the relay chain first:

1. Identify the target sensor location.
2. Verify line-of-sight or near-LOS to at least one repeater.
3. Trace that repeater's path to a node with internet/MQTT uplink.
4. Add intermediate repeaters if any hop is marginal.

Data Gaps and Local Storage

If the mesh path to a gateway is down, sensor readings are lost - sensor nodes have no local storage. Mitigation options:

- **Store-and-Forward (Meshtastic):** the Store & Forward module requires a dedicated ESP32 node with PSRAM acting as a S&F server on a private channel, and it primarily reserves *text-message* history on request. It is **not** a transparent telemetry buffer that automatically backfills sensor data across gateway outages. For sensor-data gap recovery, prefer local SD logging (below).
- **MeshCore room servers:** a Room Server is a store-and-forward BBS that holds room *chat* history for clients on request - it is not a sensor-telemetry buffer that flushes accumulated readings across a gateway outage. See MeshCore docs; do not rely on it to recover lost telemetry.
- **Local SD card logging:** for critical sensors add an SD card module and log locally in CSV format. This is the recommended way to recover from gateway outages. A recovery script can push historical data to InfluxDB when connectivity is restored.

Maintenance Planning

Remote sensor nodes require infrequent but non-zero maintenance:

- BME280 radiation shield accumulates dust, pollen, and spider webs over time - clean annually or after wildfire smoke events.
- **Fit an in-line fuse (or PTC/polyfuse) on the battery positive lead of every field node.** Outdoor wiring is exposed to corrosion, abrasion, and water, and an unfused lithium pack can start a fire on a short. Inspect the fuse and connections during the annual maintenance visit.
- INA219 shunt connections can corrode in marine environments - inspect annually and apply dielectric grease.
- Battery capacity degrades over 2 - 4 years - plan for a pack swap.

- **Label every enclosure** with the node name, deployment date, battery install date, and a contact name/number. Future you (or a search and rescue volunteer) will be grateful.
- Design for access: if a node is on a 3-hour hike, make the enclosure tool-free to open (quarter-turn latches rather than screws).