IoT & Sensors

LoRa mesh for remote sensor networks, telemetry, and automation.

□□ Start Here — IoT & Sensors Guide

This book covers using LoRa mesh as the communication layer for sensor networks - environmental monitoring, smart agriculture, asset tracking, and data pipeline integration. Most of the detailed sensor-telemetry workflows here are built on **Meshtastic**; MeshCore's sensor support is newer and more limited (see the MeshCore Sensors section below for an honest summary of what it can and can't do).

□□ Quick Start by Goal

- **Just want temperature/humidity from my nodes:** Meshtastic Telemetry Module (in Meshtastic book)
- **Build a custom sensor node:** Building an Environmental Sensor Node
- **Get data into Grafana:** MQTT to InfluxDB and Grafana
- **Water quality / flood monitoring:** Water Quality and Flood Monitoring Networks
- **Air quality monitoring:** Air Quality and Environmental Monitoring Networks

□□ What's In This Book

Sensor Node Hardware

- Sensor Node Hardware Selection
- Building an Environmental Sensor Node
- Solar-Powered Sensor Node Deployment

Data Pipelines

- MQTT to InfluxDB and Grafana

- MQTT Integration for Sensor Data
- MeshCore Sensor Data Integration
- Home Assistant and Node-RED Integration

Application Guides

- Environmental Monitoring with Meshtastic
- Remote Asset Tracking
- Building a Mesh Weather Station Network
- Field Sensor Deployment Guide

Real-World Projects

- Water Quality and Flood Monitoring Networks
- Air Quality and Environmental Monitoring Networks

MeshCore Sensors

MeshCore sensor support is an **evolving, compile-time** capability - there is no standalone "Sensor" firmware variant, adverts do not carry sensor readings, and telemetry is pulled via a request/response protocol. Read these pages for an honest, scoped overview rather than a turnkey workflow:

- MeshCore Sensor Nodes - what compile-time sensor support actually is, how telemetry is delivered, and the real `meshcore_py` data path
- MeshCore Sensor CLI Reference - the real `sensor get` / `sensor set` / `sensor list` commands

➡ Related Books

- Meshtastic - Telemetry module and MQTT integration
- Room Servers & Gateways - MQTT gateways and Node-RED
- Solar & Power Systems - Powering remote sensor nodes

IoT Applications

Introduction to LoRa Mesh for IoT

LoRa mesh networks provide a compelling platform for IoT sensor deployments, especially where WiFi doesn't reach, cellular is too expensive, and wired connections are impractical.

When LoRa mesh is the right choice for IoT

Scenario	LoRa mesh advantage
Remote sensors (field, barn, remote cabin)	No cellular or WiFi needed; solar-powered nodes transmit data back to base
Large properties (farms, ranches, campuses)	Single gateway + relay nodes covers miles; WiFi would require many access points
Emergency/event temporary deployment	No infrastructure setup; nodes self-organize; deploy in minutes
Low-bandwidth telemetry (weather, soil, water)	LoRa's low data rate matches sensor data volumes perfectly
Deep sleep battery operation	Sensors sleep between readings; nRF52 nodes draw only microamps in deep sleep, so the radio's duty cycle dominates run time

LoRa mesh vs. standalone LoRaWAN for IoT

LoRaWAN (The Things Network, Helium) requires fixed gateways with internet uplinks. LoRa mesh (MeshCore, Meshtastic) self-organizes and works in areas with no internet or gateway infrastructure. Tradeoffs:

	LoRaWAN	LoRa Mesh (MeshCore)
Infrastructure required	Yes - gateway needed	No - self-organizing
Range extension	Gateway-only (no repeating)	Multi-hop relay through mesh
Data rate	Higher (ADR)	Lower (fixed preset)
Cloud integration	Built-in (TTN, Helium)	Manual (MQTT bridge)
Best for	Fixed sensor fields near gateways	Remote, no-infrastructure, or mobile IoT

Typical IoT payload sizes

LoRa mesh is suitable for low-bandwidth sensor data. Typical packet sizes:

- Temperature + humidity: ~10 - 20 bytes
- GPS position: ~20 - 30 bytes
- Multi-sensor (temp + humidity + pressure + battery): ~40 bytes
- Short text alert: ~50 - 100 bytes

Long Fast is the faster of the common presets (higher data rate, shorter airtime); Medium Slow and Long Slow are progressively slower (lower data rate, longer airtime) in exchange for more range and sensitivity. A 40-byte sensor reading transmits in a fraction of a second on Long Fast and takes longer on the slower, longer-range presets. Either way, IoT use cases are generally not limited by data rate.

Battery life for IoT sensor nodes

With the Heltec T096 (nRF52840, ~13 µA deep sleep on the bare board, around $30 as of 2026-06-08) and a 1000 mAh LiFePO4 cell:

```
Bare-board deep-sleep current: ~13 µA
Wake + measure + transmit: ~25 mA for ~0.5 seconds every 15 minutes

Average current ≈ 13 µA + (25,000 µA × 0.5s / 900s) ≈ 27 µA average

Battery life (radio/MCU only) ≈ 1000 mAh / 0.027 mA ≈ ~37,000 hours ≈ ~4 years (theoretical)
```

This is a best-case theoretical figure for the bare board: a real enclosed node adds sensor and quiescent draws, and LiFePO4 calendar aging plus self-discharge mean the cell will not actually deliver four full years of capacity. A small solar cell can keep the battery topped up across most of

the year, but size for your worst-month sun and expect seasonal limits - "maintenance-free for 5+ years in any climate" is an over-promise. Never charge LiFePO4 below 0 °C (32 °F); use a charger/controller that blocks charging below freezing.

Remote Sensor Deployment Guide

A practical guide for deploying LoRa mesh sensor nodes in the field for environmental monitoring, agriculture, and infrastructure monitoring.

Example use cases from the community

Weather station network

Multiple BME280-equipped nodes reporting temperature, humidity, and pressure at regular intervals. A gateway node (with internet uplink) receives all reports and logs them to a database. The mesh provides coverage even when nodes are miles apart across a farm or forest.

Soil moisture monitoring

Capacitive soil moisture sensors (e.g. STEMMA Soil Sensor) connected to a RAK4631 WisBlock. Node wakes every 30 minutes (this interval is user-configurable), reads moisture level, transmits if below threshold (irrigation alert), returns to sleep. With a typical pack (e.g. a 3000 mAh cell at well under 1 mA average) the battery can last many months on a single charge with occasional solar topping.

Water level monitoring

Ultrasonic or pressure transducer water level sensors for creek gauging, tank monitoring, or flood early warning. The mesh allows data to reach a gateway even when the sensor is in a remote canyon without cellular coverage. Note: a hobbyist mesh is a supplementary indicator only - official NWS/USGS warnings remain the authoritative flood-warning source.

Gate and door alerts

Reed switch or Hall effect sensor triggers a mesh message when a gate, shed door, or access point is opened. Uses a latching trigger architecture (interrupt-driven, not polling) for maximum battery life.

Choosing hardware for IoT sensor nodes

Board	MCU	Sleep current	Sensor interfaces	Best for
Heltec T096	nRF52840	~13 µA (per Heltec spec, bare board)	I²C, SPI, UART, ADC	Ultra-low power remote sensors
RAK4631 WisBlock	nRF52840	~2 µA (MCU only)	WisBlock sensor slot ecosystem	Modular sensor builds with WisBlock sensors
Heltec V4	ESP32-S3	~10 µA (bare-MCU deep-sleep figure; verify against Heltec V4 spec - typical whole-board deep-sleep current is higher)	I²C, SPI, UART, ADC, WiFi	Gateway nodes that also serve WiFi
XIAO nRF52840	nRF52840	~5 µA (see Seeed XIAO nRF52840 power spec)	I²C, SPI, UART	Compact DIY sensor builds

The RAK WisBlock sensor ecosystem

RAKwireless produces a modular ecosystem (WisBlock) where sensor modules snap directly onto the RAK4631 base board without soldering (see each module's RAKwireless datasheet page):

- RAK1901: Temperature + humidity (SHTC3)
- RAK1902: Barometric pressure (LPS22HB)
- RAK1904: 3-axis accelerometer (LIS3DH) - vibration / motion detection
- RAK12010: Ambient light sensor (see RAK12010 datasheet)
- RAK12019: UV index sensor (see RAK12019 datasheet)

- RAK12500: GPS module (u-blox ZOE-M8Q)
- RAK13010: SDI-12 interface for agricultural soil/water sensors (see RAK13010 datasheet)

This ecosystem significantly reduces build complexity - no custom PCB or wiring required for common sensor types.

Deployment checklist

- Enclosure rated IP65+ with UV-resistant housing
- Cable glands on all penetrations (sensor cable, antenna, power)
- Desiccant pack inside enclosure; replace annually
- Solar panel and MPPT charge controller for remote sites
- **In-line fuse (or PTC/polyfuse) on the battery positive lead**, sized just above peak load, to protect against a short-circuit fire if outdoor wiring chafes, floods, or is damaged. A lithium pack can source tens of amps into a fault.
- LiFePO4 battery (not LiPo) for outdoor/temperature-variable deployments - but note that LiFePO4, like all lithium chemistries, must **not be charged below 0 °C (32 °F)**; cold-charging causes lithium plating and is hazardous. Use a charge controller with a low-temperature charge cutoff. (LiFePO4 tolerates cold *discharge* better than LiPo, which is why it is preferred outdoors.)
- Antenna mounted outside enclosure (even 1 meter above the enclosure adds range)
- GPS coordinates recorded and logged - for the network map and maintenance records
- Label the enclosure with node name and maintainer contact

Getting data off the mesh

Sensor data on the mesh is useful only if someone can read it. Options for data collection:

1. **Manual retrieval:** A person with a phone and the app periodically reads nodes. Simple but not real-time.
2. **Room server (MeshCore):** A MeshCore Room Server (running on dedicated nRF52840 or ESP32 hardware) is a store-and-forward BBS for room chat history, not an MQTT/telemetry gateway. Verify the Room Server's role against the MeshCore Room Server documentation; MQTT bridging is a separate gateway component, not a built-in Room Server feature.
3. **Python API script:** For MeshCore, the meshcore-py library is scoped to message/contact logging - use a script on a server-connected node to log incoming messages/contacts to a database (see the MeshCore Python API page). For logging actual structured *sensor telemetry*, use the Meshtastic Telemetry + MQTT path instead.
4. **Internet-bridged gateway:** A MeshCore node with WiFi (Heltec V4) connected to a home network; the Python library reads data and forwards to any cloud service.

IoT Integration Guides

Step-by-step guides for integrating LoRa mesh with sensors, home automation, and asset tracking systems.

Environmental Monitoring with Meshtastic

Use Case Overview

LoRa mesh networking enables remote environmental monitoring in locations without cellular coverage or WiFi infrastructure. A solar-powered Meshtastic node with an attached sensor can transmit its natively-supported telemetry - temperature, humidity, barometric pressure, gas resistance (air-quality estimate), plus voltage and current - across the mesh to a gateway, which forwards it to a database or home automation system. Other sensor types (for example soil moisture) are **not** part of Meshtastic's native environment telemetry and require a separate microcontroller with a custom port number to publish onto the mesh.

Common applications include:

- Weather station on a remote property or off-grid cabin
- Soil moisture monitoring for irrigation management across a farm (requires a custom integration - see note above)
- Air quality monitoring at a remote site or community location
- Freezer/cold storage temperature monitoring with alerts

Supported Sensor Modules

Meshtastic's Telemetry module supports a range of **I2C digital** sensor types natively (supported sensors on the I2C bus are auto-detected at startup):

- **BME280 / BME680** - temperature, relative humidity, barometric pressure; BME680 also provides a gas resistance reading useful for air quality estimates
- **INA219 / INA260** - DC voltage and current monitoring; useful for monitoring solar panel output, battery charge state, or load current

Note: **MQ-series analog gas sensors** (MQ-2, MQ-135, etc.) are **not** natively supported by Meshtastic's Telemetry module, which reads I2C digital sensors only - there is no native ADC analog-sensor telemetry path. Using an MQ-series sensor requires a separate MCU/ADC and a

custom integration to publish the reading onto the mesh.

Hardware Setup: BME280 Wiring

The BME280 is a commonly used environmental sensor with Meshtastic. It connects via I2C. **I2C pins vary by board** - look up your exact board's default I2C (Wire) pins before wiring. As a starting point, on the classic ESP32 T-Beam the defaults are GPIO21 (SDA) / GPIO22 (SCL); the Heltec V3 (ESP32-S3) and nRF52 boards (e.g. RAK4631) use different I2C pins, so verify your specific board's pinout.

- **VCC** → 3.3V
- **GND** → GND
- **SDA** → board default SDA pin (GPIO21 on classic ESP32 T-Beam - verify your board)
- **SCL** → board default SCL pin (GPIO22 on classic ESP32 T-Beam - verify your board)

Caution - 3.3 V only: power the BME280 from the board's **3.3 V** rail, NOT 5 V. Most BME280/BME680 breakouts and the ESP32/nRF52 I2C pins are 3.3 V devices; applying 5 V, or feeding 5 V logic levels onto the 3.3 V SDA/SCL lines, can permanently damage the sensor or the MCU. If you use a 5 V-only sensor module, add a logic-level shifter on the I2C lines. (BME280 default I2C address is 0x76, or 0x77 if SDO is tied to VCC.)

After wiring, enable the sensor in the Meshtastic app or CLI: navigate to **Telemetry → Environment** and enable the module. The node will begin broadcasting environment telemetry on the mesh.

Reading Sensor Data

Telemetry data is accessible through multiple paths:

- **Meshtastic app (phone):** telemetry appears in the node info panel when you tap on a node - shows last-received temperature, humidity, pressure, and other available metrics
- **MQTT gateway:** a Meshtastic node with internet access forwards telemetry packets to an MQTT broker. **JSON output is only available on ESP32 gateways** - JSON MQTT is **not supported on the nRF52 platform** (RAK4631, etc.). Since this book's build pages center on the RAK4631 (nRF52), a RAK4631 cannot itself emit JSON to MQTT; use a separate ESP32 gateway, or parse the protobuf `ServiceEnvelope` instead. This enables integration with databases and dashboards.

Sample MQTT Telemetry JSON Structure

When a telemetry packet is forwarded via MQTT as JSON (ESP32 gateway only), the fields are **flat `snake_case` under a `payload` object**, with the envelope fields at the top level of the message. (Note: this is the JSON-MQTT format - it is *not* the protobuf `decoded.telemetry.environmentMetrics` camelCase structure; parsers built against that path will get nothing.)

- **from** (top-level, `value_json.from`) - the unique decimal node ID of the sending device
- **to** (top-level, `value_json.to`) - the destination node ID; a broadcast is `-1` (decimal of 0xFFFFFFFF)
- Other top-level envelope fields include **id**, **channel**, **type**, **sender**, and **timestamp**
- **payload** - the data object; for an environment telemetry packet it contains the flat snake_case keys:
  - **payload.temperature** - degrees Celsius (float)
  - **payload.relative_humidity** - percentage (float, 0 - 100)
  - **payload.barometric_pressure** - hPa (float)
  - **payload.gas_resistance** - BME680 only, correlates with VOC concentration (the firmware emits ohms; note the Home Assistant example labels this in MOhms, so reconcile units in your parser)
  - **payload.voltage** / **payload.battery_level** - when device-metrics telemetry is present

This JSON structure can be consumed directly by InfluxDB (via Telegraf or a Node-RED flow), Home Assistant MQTT sensors (key off `value_json.payload.*`), or any custom application that subscribes to the MQTT topic. Subscribe with a JSON-specific topic filter such as `msh/+/2/json/#` rather than a bare `msh/#` (which also catches binary `2/e/` topics the JSON parser cannot read).

Integration Targets

- **InfluxDB + Grafana:** store time-series telemetry and visualize on dashboards; well-suited for long-term environmental data logging
- **Home Assistant:** configure MQTT sensors using the official `value_json.payload.*` templates (e.g. `value_json.payload.temperature`, `value_json.payload.relative_humidity`); trigger automations on temperature or humidity thresholds
- **Node-RED:** flexible pipeline for transforming, filtering, and routing telemetry data to multiple destinations simultaneously

Deployment Considerations

- **Enclosure:** use a weatherproof IP65+ enclosure for outdoor deployments; mount the sensor in a vented radiation shield for accurate temperature readings - direct sunlight on the enclosure will give artificially high temperature readings
- **Solar power:** a small 1 - 5W solar panel with a LiPo battery can power most low-duty-cycle LoRa sensor nodes, and the low transmit duty cycle makes solar viable even in partial sun. Note that the common TP4056 module is a bare single-cell linear charger (not MPPT) with **no low-temperature charge cutoff** - it will happily charge a LiPo below 0 °C (32 °F), which damages the cell and risks a fire. For cold-climate outdoor use, add an NTC-based low-temp cutoff (or use a charger that supports it), and add an in-line fuse on the battery positive lead.
- **Sensor venting:** drill small holes or use a membrane vent in the enclosure so humidity and temperature readings reflect ambient conditions, not the trapped air inside the box

Power Impact of Environmental Sensors

The BME280 draws only a few microamps when sampling at low rates (≈3.6 µA at 1 Hz for humidity, pressure, and temperature) and well under 1 µA (≈0.1 µA) in sleep mode - negligible relative to the LoRa radio and microcontroller. At a 10-minute telemetry interval on a T-Beam or similar device, sensor power consumption has minimal impact on overall battery life. The dominant power draw remains the ESP32 or nRF52840 idle current and LoRa transmit bursts.

Home Assistant and Node-RED Integration

Integration Architecture

The standard integration path from a Meshtastic mesh to Home Assistant or Node-RED is:

1. **Meshtastic MQTT gateway** - a mesh node with WiFi (or a dedicated gateway device) publishes all received mesh packets to an MQTT broker as JSON
2. **MQTT broker (Mosquitto)** - runs on the local network (often on the same machine as Home Assistant); receives all packets from the gateway
3. **Home Assistant or Node-RED** - subscribes to the MQTT topics and processes the incoming data

Home Assistant Setup via HACS

The Meshtastic integration for Home Assistant is community-maintained (not an official HA core integration) and is available via HACS by adding `github.com/meshtastic/home-assistant` as a custom repository (Integration category), then installing it:

1. Install HACS in Home Assistant if not already present
2. In HACS, add the Meshtastic integration's GitHub URL (`github.com/meshtastic/home-assistant`) as a custom repository: HACS > Integrations > three-dot menu > Custom repositories, category "Integration". (It is not in the default HACS catalog, so a plain search will not find it until the repo is added.)
3. After adding the repo, install the Meshtastic integration from HACS
4. Add the integration in Home Assistant Settings → Integrations → Add Integration → Meshtastic (see the integration's README for the exact config-flow steps)
5. Configure with your MQTT broker address, port, and topic prefix. The default root topic is `msh` (the full path is `msh/REGION/...`); see the integration's config options.
6. Nodes appear as Home Assistant devices with individual sensors for position, battery level, and telemetry values

What You Can Do in Home Assistant

- **Geofencing automations:** trigger actions when a node's GPS position enters or leaves a defined zone - useful for "person arriving home" automations using a mesh device as a low-cost tracker
- **Telemetry alerts:** send a notification when a remote temperature sensor exceeds a threshold (e.g., greenhouse overheating, freezer failure)
- **Battery low notifications:** alert when a field node's battery level drops below a set percentage
- **Dashboard visualization:** display node positions on a map card alongside other Home Assistant entities

Node-RED Alternative

Node-RED provides more flexibility for complex processing pipelines than the Home Assistant integration:

- The simplest, most portable approach is to use Node-RED's standard **MQTT-in** node plus a **JSON** node to consume the gateway's raw MQTT JSON from the broker. A community Meshtastic contrib node for direct serial/TCP connection may also exist - verify the exact package name on flows.nodered.org before relying on it.
- Build custom routing logic: filter messages by node ID, apply transformations, forward to multiple destinations simultaneously
- Suitable for multi-destination forwarding - e.g., send alerts to both Home Assistant and a Telegram bot at the same time

Example Node-RED Flow

A typical data logging flow:

1. **MQTT In node** - subscribes to the Meshtastic MQTT topic
2. **JSON parse node** - parses the raw MQTT payload
3. **Switch node** - filters for a specific node ID (e.g., only process packets from your sensor node)
4. **Function node** - extracts the desired telemetry fields (snake_case under `payload`, e.g. `payload.temperature`) and formats a row

5. **Google Sheets node** - appends the row to a Google Sheet for long-term logging

MeshCore and Home Assistant

MeshCore does not have a native MQTT gateway equivalent to Meshtastic's built-in WiFi gateway. A real community Home Assistant integration does exist (`github.com/meshcore-dev/meshcore-ha`, HA domain `meshcore`, built on the meshcore-py library). Integration options include:

- **meshcore-ha integration:** the community-maintained `meshcore-ha` integration (domain `meshcore`) connects to a MeshCore node via the meshcore-py library and exposes it to Home Assistant
- **Serial/USB bridge script:** connect a MeshCore node to a Raspberry Pi or other Linux machine via USB, use the MeshCore Python library (meshcore-py) to read incoming data, and publish it to an MQTT broker with a custom script
- This approach requires more custom development compared to the Meshtastic HACS integration

Privacy Considerations

Position and telemetry data from your nodes is visible to *anyone* who has access to your mesh channel. By default, Meshtastic devices transmit on the public LongFast channel, meaning nearby nodes operated by others can receive your telemetry.

- Configure a private channel with a unique name and PSK for nodes you don't want public network participants to observe
- Consider whether you want GPS position data from home-based sensor nodes to be visible on public MQTT bridges. The public default Meshtastic broker is `mqtt.meshtastic.org` (verify the current host against Meshtastic docs; as of 2026-06-08).
- Disable MQTT uplink on nodes with sensitive data if you use the public broker

Remote Asset Tracking

Use Case Overview

LoRa mesh networking provides a low-cost option for tracking assets in areas with no cellular coverage: vehicles, heavy equipment, livestock, shipping containers, or packages moving through rural or remote areas. Unlike cellular asset trackers that require a SIM and data plan for *every* asset, LoRa-based trackers report over your own mesh, avoiding per-asset SIM costs. Note that this does not eliminate recurring cost entirely: to deliver positions off the local mesh to a server you still need at least one gateway with internet backhaul, which may itself be a cellular/fixed connection with its own monthly cost. The savings come from not needing a SIM per tracked asset, not from eliminating connectivity costs altogether.

How It Works

1. The asset carries a LoRa node configured to broadcast GPS position packets at regular intervals
2. Any mesh node or gateway within range receives the position packet and propagates it through the mesh via multi-hop routing
3. A gateway node with internet access (WiFi or cellular backhaul) forwards the position via MQTT to a server
4. The server stores the position in a database and displays it on a web map or dashboard

Meshtastic MQTT + Cloud Visualization

Position packets forwarded via Meshtastic's MQTT gateway can be ingested by a variety of tools:

- **Grafana map panel:** InfluxDB stores the time-series position data; Grafana's Geomap panel displays asset trails and current positions
- **Custom Leaflet application:** a lightweight web app using the Leaflet.js mapping library can display live asset positions from a database backend

- **Home Assistant:** device tracker entities can be updated from MQTT position packets, integrating asset positions into your existing HA setup

MeshCore Asset Tracking

MeshCore advertisement packets can **optionally** include the sending node's GPS position - this is not unconditional. The advert location mode controls it: **share** broadcasts the live GPS location, **prefs** advertises a location stored in preferences, and **none** (the default) includes no position in adverts. So position is only present in adverts when the node is configured to share it. Where positions are being advertised, a repeater that hears an advert and logs the contact (node ID and time) can make it possible to reconstruct asset movement through a coverage area by analyzing which repeaters logged contact with a given node ID and when. This is useful for:

- Understanding which parts of a property or trail system an asset has visited
- Detecting when an asset enters or leaves a coverage zone
- Post-event reconstruction of movement without requiring a centralized database during the event

Update Interval Trade-off

Position update frequency involves a trade-off between tracking resolution, battery life, and mesh traffic:

- **Every 1 minute:** good resolution for fast-moving vehicles, but significantly higher battery drain and substantial added mesh traffic. On a shared LoRa channel, 1-minute position broadcasts are aggressive - they can quickly eat into practical airtime/duty-cycle budgets on a busy mesh and crowd out other nodes. Meshtastic deliberately defaults position/telemetry to much longer intervals (30 minutes). Reserve sub-minute or 1-minute rates for a very small number of nodes or short tests, not a fleet.
- **Every 10 - 30 minutes:** appropriate for most slow-moving assets (livestock, equipment); much better battery life, and roughly in line with Meshtastic's 30-minute default
- **Motion-triggered:** the most power-efficient approach - configure the node to transmit on movement detected via accelerometer, and go silent when the asset is stationary. Ideal for parked equipment or containers that move infrequently.

Hardware Options

- **RAK4631 + RAK1910 GPS module:** excellent for asset tracking - the RAK4631's nRF52840 has very low sleep current, and the modular WisBlock system allows GPS to be

powered down when not sampling. Best-in-class for long-term battery-powered deployments.
- **T-Echo:** integrated GPS and ePaper screen for field checking the last known position; good for assets that someone will physically inspect periodically (e.g., a piece of equipment on a large property)
- **T-Beam:** has built-in GPS and is popular for vehicle tracking; higher power consumption than the RAK solution but easier to integrate with vehicle 12V power

Coverage Considerations

Asset tracking over LoRa mesh requires mesh coverage in the asset's operating area. Without a mesh node or gateway within range, the asset simply does not report until it re-enters coverage:

- For wide-area tracking across a large property, plan repeater placement to improve coverage of the operating area
- A gateway with cellular backhaul at the edge of coverage can ensure position data reaches a server even if the local mesh is isolated
- Assets that travel outside the mesh coverage area will have gaps in their track. The mesh does **not** fill these gaps: positions a node broadcasts while it is out of range are lost entirely - there is no retroactive delivery once it returns. The server can only ever display the last position that was received while the asset was in range, so store the last known position and timestamp to show "last seen at X location at Y time."

Regulatory and Privacy Considerations

Transmitting GPS coordinates of people or assets has legal and ethical implications. The points below are general guidance, not legal advice - consult local law for your situation:

- **Consent:** ensure anyone being tracked (employees, family members) is aware of and consents to the tracking. Covert GPS tracking of people may be illegal depending on your jurisdiction - laws vary widely by state and country, so consult local law and obtain consent before tracking any individual.
- **Employees:** employer tracking of company vehicles during work hours is generally permissible in the US but varies by jurisdiction; consult applicable state employment law (this is general guidance, not legal advice)
- **Livestock and equipment:** tracking your own property is generally straightforward; tracking assets on shared or public land may have additional considerations
- **Channel access:** as with all mesh data, position packets are visible to anyone with access to the same channel. Use a private channel with a unique PSK for asset tracking

deployments to prevent third parties from observing your assets' movements.

Data Pipelines and Logging

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.TELEMETRY_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
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 `ENV_INCLUDE_*` 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 µA
BME280 (sleep / active)	~0.1 µA sleep; ~3.6 µA active at 1 Hz
LoRa TX burst (10 s/day total)	~0.1 mA averaged (TX current × airtime ÷ 86400 s; e.g. ~118 mA at +22 dBm × ~10 s/day ÷ 86400 s ≈ 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 → ~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 °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 re-serves *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).

MeshCore Sensor Configuration

MeshCore Sensor CLI Reference

Overview

This page describes MeshCore's **actual** sensor support, which is deliberately minimal and differs substantially from the richer Meshtastic telemetry model. The most important facts to understand up front:

- **Sensor support is a compile-time feature.** Which sensor (if any) a node uses is selected when the firmware is built — there is no runtime command to pick or configure a sensor driver. A build only has sensor support if it was compiled with it (e.g. the `ENV_INCLUDE_*` build flags and example/sensor builds such as `simple_sensor` / SensorMesh). There is no separate deployable "SENSOR firmware variant" or role; the firmware roles are **Companion**, **Repeater**, and **Room Server**.
- **The entire sensor CLI is three commands:** a generic key/value store — `sensor list [start]`, `sensor get <key>`, and `sensor set <key> <value>`. These are only available when sensor support is compiled in.
- **Sensor readings are NOT broadcast in adverts.** A MeshCore advert carries only the node's name, optional position, and signed public key. Live sensor data is delivered on demand through the binary request/response protocol (`REQ_TYPE_GET_TELEMETRY_DATA`, CayenneLPP-encoded and permission-gated) — a client requests it; the node does not push it to the mesh.

Rich per-sensor runtime configuration is not currently available in MeshCore. There are no commands to select a sensor type, set its I2C address, set a per-sensor read/broadcast interval, enable/disable a sensor, trigger an immediate read, or apply a temperature offset at runtime. If you need those behaviors, they must be handled at compile time in the firmware or in the host application that consumes the telemetry. Always check the current documentation at docs.meshcore.io/cli_commands, because MeshCore is evolving and sensor support is expanding.

Accessing the CLI

MeshCore Repeater, Room Server, and sensor-capable nodes are configured over a USB/serial connection (using a terminal app, PuTTY, or the Arduino Serial Monitor) or over BLE via the MeshCore companion app or the `meshcore-cli` tool. Serial and BLE access to a node is at `115200 baud`. There is no documented "commands are case-insensitive" behavior — use the documented lowercase command names. Whether a given change takes effect immediately or requires a reboot is per-command (for example, `set radio` requires a reboot to apply), so do not assume every setting applies instantly.

The Sensor CLI (compile-time sensor builds only)

When a build includes sensor support, MeshCore exposes a small generic key/value store for sensor-related custom variables. These are the only sensor commands that exist:

Command	What it does
`sensor list [start]`	Lists the sensor custom variables as `key=value` lines (optionally starting from an index).
`sensor get <key>`	Prints the current value of one sensor variable.
`sensor set <key> <value>`	Sets the value of a sensor variable.

The keys available depend on what the compiled-in sensor build defines; this is a generic custom-variable mechanism, not a fixed set of sensor settings.

The following commands do **not** exist in MeshCore and will not work — do not use them (they are Meshtastic-style or invented): `set sensor type`, `set sensor addr`, `set sensor interval`, `set sensor enabled`, `set sensor temp_offset`, `sensor read`, `sensor status`, and `i2c scan`. The sensor (and its I2C address) is fixed at compile time; there is no runtime sensor-type table, enable/disable toggle, immediate-read command, status dump, or I2C scanner.

Reading Sensor Data (telemetry protocol)

There is no `sensor read` command and no human-readable console-output block. To obtain live sensor values, a client requests telemetry over the binary protocol: `REQ_TYPE_GET_TELEMETRY_DATA`. The node returns the data encoded as **CayenneLPP** (channel/type/value triplets), and access is permission-gated. In the Python library this is the `get_self_telemetry` path rather than a CLI

command. Decoding the CayenneLPP response into named values (temperature, humidity, pressure, etc.) is done by the requesting client, not printed by the node.

MeshCore Device CLI Reference

The real MeshCore node configuration commands (for Repeater / Room Server / sensor builds) include the following. See docs.meshcore.io/cli_commands for the authoritative and current list.

Command	Purpose
`set name <name>`	Set the node name.
`set tx <dbm>`	Set transmit power in dBm.
`set radio <freq>,<bw>,<sf>,<cr>`	Set radio parameters. Requires a reboot to apply.
`get <key>`	Read a device setting.
`advert`	Send an advert immediately.
`set flood.advert.interval <hours>`	Set the flood-advert interval, in hours (default 12 for Repeater, 0 for sensor).
`gps on` / `gps off`	Toggle GPS power state.
`set powersaving on` / `set powersaving off`	Change the power-saving mode (Repeater, persisted to prefs).
`reboot`	Reboot the node.
`erase`	Erase stored configuration.

Advert cadence is controlled by `set flood.advert.interval` (hours) — there is no per-second "sensor broadcast interval." Note again that adverts do not contain sensor readings.

Sensor Hardware and Wiring

Because the sensor driver is fixed at build time, the wiring below is about physically connecting a supported I2C sensor to the board, not about telling MeshCore which sensor to use. These I2C facts are accurate regardless of firmware.

- **BME280 / BME680 default I2C address:** `0x76` when SDO is tied low (to GND), or `0x77` when SDO is tied high (to VCC). The BME680 uses the same addressing as the BME280.
- **BME280 measures temperature, humidity, and pressure only** (no gas / air quality). The **BME680** adds a VOC gas / IAQ sensor.
- **Power the sensor from 3.3 V, not 5 V.** The BME280/BME680 and the nRF52/ESP32 I2C pins are 3.3 V devices; applying 5 V (or driving 5 V logic onto the 3.3 V SDA/SCL lines) can

permanently damage the sensor or the MCU.

For the cleanest integration on the RAK4631, use the **RAK1906 WisBlock Environmental Sensor module** (a Bosch BME680). It plugs directly into a WisBlock Base Board sensor slot with no soldering and no I2C wiring — the base board routes power and the shared I2C bus automatically. The RAK1906 is an I2C module and can go in any of the I2C sensor slots; slot A is just a convenient default.

If wiring a bare BME280/BME680 breakout to the RAK4631 GPIO header instead, use the WisBlock I2C1 bus:

Sensor pin	RAK4631 pin
VCC	3V3 (3.3 V — never 5 V)
GND	GND
SDA	WB_I2C1_SDA
SCL	WB_I2C1_SCL
SDO	GND (selects I2C address 0x76; tie to 3.3 V for 0x77)
CSB	3V3 (selects I2C mode)

Keep I2C wires short. If you must run the sensor on an extended cable, add 4.7 kΩ pull-up resistors on SDA and SCL.

Where to Go Next

For the authoritative, current CLI command list see docs.meshcore.io/cli_commands. Sensor support is a compile-time firmware feature, so which sensors and keys are available depends on the specific build. For host-side integration, the `meshcore_py` library and the MeshCore Home Assistant integration (`github.com/meshcore-dev/meshcore-ha`, domain `meshcore`) consume telemetry through the binary request/response protocol described above.

Sensor Node Hardware

Sensor Node Hardware Selection

Sensor Node Hardware Selection

Choosing the right sensor hardware determines the long-term reliability, accuracy, and maintainability of your mesh monitoring deployment. This page compares the two dominant approaches: RAK WisBlock modular sensor boards and Meshtastic telemetry running on commodity hardware such as the TTGO T-Beam.

RAK WisBlock Sensor Modules

WisBlock is RAK Wireless's modular ecosystem: a WisBlock Core module (e.g., the RAK4631 = Nordic nRF52840 / SX1262) pairs with a WisBlock Base board such as the RAK19007 (other bases exist, e.g. RAK5005-O and RAK19003). Sensor modules snap onto the base board's sensor slots with no soldering required, making field assembly and repair straightforward.

- **RAK1906 (BME680)** - Measures temperature (±1°C), relative humidity (±3% RH), barometric pressure (±0.6 hPa), and volatile organic compound (VOC) air quality index. The BME680 gas sensor requires a burn-in period of roughly 48 hours before IAQ readings stabilise (IAQ also depends on the Bosch BSEC library and ongoing calibration). Current draw: ~0.15 µA sleep; the standard temperature/humidity/pressure measurement is in the microamp range (a few µA at 1 Hz), while the integrated gas (VOC) heater draws several mA in bursts only when VOC sensing is enabled. Ideal for indoor air quality and outdoor environmental monitoring.
- **RAK12500 (u-blox ZOE-M8Q GPS)** - Adds GNSS positioning for mobile or asset-tracking nodes. Cold-start TTFF ~29 s, hot-start ~1 s. Active current ~17-18 mA; disable when stationary to preserve battery. Compatible with external active antenna via U.FL connector.
- **RAK12004 (MQ-2 Gas Sensor)** - Detects LPG, propane, hydrogen, methane, and smoke. The MQ-2 heater (~150 mA) cannot be put to sleep and must run continuously for valid readings; duty-cycling it to save power makes gas readings unreliable until the heater re-stabilises (tens of seconds to minutes after warm-up). This ~150 mA continuous draw (~3.6 Ah/day) dominates the power budget, so the MQ-2 is poorly suited to small solar/battery nodes and should not be relied on as a life-safety combustible-gas detector.

- **RAK1901 (SHTC3)** - Dedicated temperature/humidity sensor with ±0.2°C and ±2% RH accuracy. Lower-power alternative to the BME680 when pressure and air quality are not needed. Current: ~0.5 mA during measurement, <1 µA idle.

Meshtastic Telemetry on T-Beam / Generic Boards

Meshtastic supports telemetry from I2C sensors wired to the GPIO header of ESP32-based boards. Common pairings include:

- **BMP280 / BME280** - Temperature, pressure, and (BME280) humidity. Widely available and inexpensive. Direct I2C wiring to SDA/SCL pins. The BME280 draws only ~3.6 µA active (typical, @ 1 Hz, humidity+pressure+temperature) and well under 1 µA in sleep per the Bosch datasheet - negligible relative to the radio and MCU.
- **SHT31** - High-accuracy temperature and humidity (±0.3°C, ±2% RH). More robust against contamination than the cheap capacitive temperature/humidity modules often used on hobbyist nodes.
- Enable the Telemetry module in Meshtastic and set the sensor type in the module config. Data is broadcast on the mesh as Protobuf telemetry packets at the configured interval.

Power Consumption Comparison

Component	Active Current	Sleep Current
RAK4631 base node (LoRa TX)	10 - 50 mA (lower-power TX); SX1262 reaches ~118 mA at +22 dBm	2.5 µA
BME680 (RAK1906)	+microamps (T/RH/P); gas heater several mA in bursts only when enabled	+0.15 µA
SHTC3 (RAK1901)	+~0.5 mA (during measurement)	+0.5 µA
ZOE-M8Q GPS (RAK12500)	+~17-18 mA	+7.5 µA (backup)
MQ-2 heater (RAK12004)	+150 mA	Cannot sleep heater
T-Beam + BME280 (Meshtastic)	~80 mA (board-level)	~500 µA

Note: the T-Beam figures are board-level (ESP32 + GPS + peripherals). Stock T-Beam deep-sleep current is frequently in the low-mA range unless peripherals are disabled, and is often higher than 500 µA in practice; the BME280's own contribution is microamp-level.

For battery-constrained outdoor deployments the RAK WisBlock platform with BME680 or SHTC3 is strongly preferred. Base sleep current below 5 µA enables multi-month operation on a modest LiPo without solar, assuming moderate temperatures - cold significantly reduces usable battery

capacity. **Outdoor lithium-powered nodes (including LiFePO4) must not be charged below 0°C (32°F);** see the deployment pages for low-temperature charge-cutoff guidance when a node is solar-equipped.

Form Factor and Weatherproofing

Outdoor sensor nodes must be rated for the deployment environment. Common IP ratings relevant to mesh sensor nodes:

- **IP65** - Dust-tight, protected against low-pressure water jets. Minimum for exposed outdoor use.
- **IP67** - Dust-tight, temporary immersion to 1 m. Suitable for ground-level or flood-risk sites.
- **IP68** - Continuous submersion rated. Required near water crossings or in humid tropical climates.

Membrane vents (Gore-Tex or equivalent) are essential for enclosures containing humidity sensors. A sealed enclosure traps heat and distorts readings. In the Northern Hemisphere, mount the enclosure on a north-facing surface to minimise solar heating effects on temperature sensors, or use a radiation shield (Stevenson screen style) for meteorological-grade accuracy.

Building an Environmental Sensor Node

Building an Environmental Sensor Node

This guide walks through assembling a production-ready environmental sensor node using the RAK WisBlock platform with the BME680 sensor, then configuring it for MeshCore firmware deployment.

Bill of Materials

- RAK19007 WisBlock Base Board (v2)
- RAK4631 WisBlock Core (nRF52840 + SX1262)
- RAK1906 WisBlock Sensor (BME680)
- LoRa antenna (868 MHz or 915 MHz depending on region), SMA-to-IPEX pigtail
- 3.7 V LiPo battery (1000 - 3000 mAh) with JST 2.0 connector
- In-line fuse or polyfuse for the LiPo positive lead (outdoor/lithium safety)
- Enclosure: Hammond 1591XXFLBK (approx. 120×65×40 mm) or RAK Unify Enclosure (approx. 100×75×38 mm) - verify exact dimensions against the manufacturer datasheet
- 2× M16 IP68 cable glands
- Silica gel desiccant packet (2 g)
- Gore-Tex membrane vent plug (optional but recommended)

Safety note: A single-cell LiPo must never be charged below 0 °C (32 °F) - doing so plates lithium and risks a cell fault or fire. If this node will be deployed where winter temperatures drop below freezing, use a charger/PMIC with a low-temperature charge cutoff (NTC-based), and verify the RAK19007's cold-charge behavior before cold-weather deployment. Always include an in-line fuse on the battery positive lead.

Assembly Steps

1. **Prepare the base board.** The RAK19007 features four sensor slots (A, B, C, D); slots A-C take 10 mm modules and slot D takes larger 23 mm modules (and supports GNSS). No soldering is required - all WisBlock modules use proprietary 40-pin or 24-pin board-to-board connectors. Press the RAK4631 core firmly onto the core slot until it clicks.
2. **Attach the BME680 sensor module.** The RAK1906 is an I2C sensor and can go in any of the I2C sensor slots (A-C for 10 mm modules); Slot A is just a convenient default. It uses the 24-pin connector. Align the gold contacts and press until seated. The module sits flush with the base board surface.
3. **Connect the LoRa antenna.** Attach the IPEX end to the U.FL connector on the RAK4631 (labelled LoRa). Ensure the antenna is fully connected before powering on to avoid PA damage. Route the SMA pigtail through a cable gland in the enclosure wall.
4. **Connect the battery.** Plug the JST 2.0 LiPo connector into the battery port on the RAK19007 - verify connector polarity first, as RAK battery-connector polarity is not universal across LiPo vendors. The base board includes an onboard USB-C LiPo charger that stops charging once the battery is full. Over-discharge protection generally comes from the battery pack's own protection circuit (PCM) rather than the base board, so use a protected cell.
5. **Flash the firmware.** Download the appropriate RAK4631 MeshCore firmware `.uf2` (e.g. a Companion or Repeater build) from the MeshCore GitHub releases page. MeshCore has no dedicated "SENSOR" firmware variant - sensor support is a compile-time feature selected at build time, so confirm a sensor-capable build before relying on one. (Alternatively you can flash standard Meshtastic firmware, whose Telemetry module supports the BME680 directly.) Put the RAK4631 into bootloader mode (double-tap reset - the drive `RAK4631` should appear), then drag-and-drop the `.uf2` file.

Firmware Configuration

After flashing, connect over USB serial or BLE (via the companion app / meshcore-cli) to configure the node. The MeshCore device CLI uses commands such as:

```
set name "SensorNode-01"   # Set the node name
gps off                    # Disable GPS for indoor/static nodes
set advert.interval 15     # Advert cadence in minutes (or set flood.advert.interval in hours)
reboot
```

Note: MeshCore has no `set telemetry_interval`, `set gps_enabled`, `set power_save`, or `set node_name` command - those are Meshtastic-style names that do not exist in MeshCore. Power saving (`powersaving on`) is documented as a repeater-role feature. If you are running Meshtastic instead, set the environment telemetry interval with `meshtastic --set telemetry.environment_update_interval 900`.

In Meshtastic, a supported BME680 on the I2C bus is auto-detected at startup; MeshCore selects sensor hardware at compile time rather than by an I2C auto-scan. Once configured, confirm telemetry is being produced on a listener node or via the telemetry request protocol. On the

BME680's 0-500 IAQ scale (which requires the Bosch BSEC library and a calibration period): 0-50 good, 51-100 moderate, 101-150 little bad / unhealthy for sensitive groups, 151-200 bad, 201-300 worse, 301-500 very bad.

Enclosure and Weatherproofing

Drill two M16 cable gland holes in the bottom face of the enclosure - one for the LoRa antenna SMA cable and one for a USB-C charging cable if periodic wired charging is desired. Thread the cable glands and tighten until the rubber grommet compresses around the cable. Place the desiccant packet inside and install the Gore-Tex vent plug on a side wall to allow pressure equalisation without moisture ingress. Apply silicone sealant around any remaining penetrations.

Mount the PCB assembly on the supplied standoffs inside the enclosure using M3 screws. Ensure the BME680 sensor on the RAK1906 faces toward the vent plug - airflow across the sensor significantly improves humidity response time and reduces self-heating error.

Label the outside of the enclosure with the node name, installation date, frequency region, and an emergency contact. Photograph the final assembly before sealing for maintenance records.

Commissioning Checklist

- Antenna connected before power-on
- Firmware version confirmed in boot log
- Sensor readings visible in serial output
- Node appearing in MeshCore network map within expected hop count
- Telemetry packets visible at expected interval on a listener node
- Enclosure sealed, cable glands tightened
- In-line fuse fitted on the battery positive lead
- Node name and installation date label affixed externally

Solar-Powered Sensor Node Deployment

Solar-Powered Sensor Node Deployment

A well-designed solar sensor node can run for years with minimal maintenance in favorable climates, but no solar node runs indefinitely: batteries degrade with calendar aging, panels accumulate dirt and snow, and winter insolation at high latitudes or under dense canopy can fall short. Size for the worst-case (winter) month for your site, plan for periodic battery replacement and seasonal checks, and aim for an average current consumption well below 1 mA so that even a small panel can replenish the battery during short winter days.

Power Budget Design

Start with a current budget before selecting hardware. Charge per event is current × time (for example 5 mA × 1.5 s = 7.5 mAs = 2.08 µAh). A 15-minute telemetry cycle on a RAK4631 + BME680 node, transmitting at full +22 dBm output, breaks down as follows:

Event	Duration	Current	Charge (µAh)
Deep sleep	898 s	3 µA	0.75
Wake + sensor read	1.5 s	5 mA	2.08
LoRa TX (1 packet, +22 dBm)	0.5 s	118 mA	16.4
Total per 15-min cycle	900 s	-	**≈ 19.2 µAh**
Average current	-	-	**≈ 0.077 mA**

At ~0.077 mA average, daily consumption is ~1.85 mAh (≈ 19.2 µAh × 96 cycles/day). Note the SX1262 draws ~118 mA at +22 dBm, so TX dominates the budget — if you transmit at a lower output power the average drops further, but never assume the 40 mA figure sometimes quoted for low-power transmit. Size the panel against the worst-case (winter) month rather than an annual

average: a small 0.5 W panel can comfortably cover this load on short overcast winter days at mid-latitudes, but always confirm against the December peak-sun-hours for your target latitude, panel voltage, and conversion losses, and derate for soiling and snow.

Sleep/Wake Cycle Design

The nRF52840 on the RAK4631 supports deep sleep with RAM retention at 2.5 µA. Choose the minimum useful reporting interval for your application:

- **Weather monitoring:** 15 - 30 minutes is typically sufficient. Temperature and humidity change slowly outdoors.
- **Air quality / smoke detection:** a shorter interval (e.g. 5 minutes or less) is advisable. VOC spikes and smoke events evolve faster than weather parameters.
- **Asset tracking with GPS:** 1 - 5 minutes when moving, 30 minutes when stationary (detected via onboard accelerometer). GPS adds ~15 - 25 mA active depending on the module - budget accordingly.

Avoid waking more frequently than necessary. Each LoRa transmission occupies shared airtime. At a 15-minute interval a single node has a very low duty cycle (~0.5%), which is comfortably within EU 868 MHz duty-cycle limits. US/Canada 915 MHz operation is governed instead by FCC Part 15 rules, which impose **no duty-cycle limit** on digital-modulation LoRa — only a 400 ms maximum dwell time per channel (with frequency hopping). Verify the rules for your specific region and band.

Solar Panel Selection

Match panel output to your deployment's worst-case (winter) solar insolation. A conservative rule of thumb: the panel's short-circuit current (I_{sc}) should be at least 10× the node's average current draw.

- **0.5 W, 5 V panel** (~100 mA I_{sc}) - Sufficient for a basic BME680 node at 15-minute intervals. Physically small (~80×55 mm), suitable for fence-post or junction-box mounting.
- **1 W, 6 V panel** (~165 mA I_{sc}) - Comfortable margin for nodes with GPS enabled part-time, or deployments above 50° latitude where winter insolation is poor.
- **MQ-2 / continuous-heater gas-sensor nodes** - An MQ-2 runs its heater continuously at ~150 mA (~3.6 Ah/day) and cannot sleep, so a 2 W panel is *not* sufficient — that load far exceeds a 2 W panel's winter harvest, and battery sizing dominates. The 10× I_{sc} rule would demand ~1.5 A I_{sc}. Either duty-cycle the heater, or budget a substantially larger panel and battery; a continuous-heater gas sensor is generally not well suited to a small solar node.

To stop the battery from discharging back through the panel at night, use a **series blocking diode** (a low-drop Schottky type) in line between the panel and the battery, or a charge controller

with built-in reverse-current/night protection — most MPPT and PWM charge controllers already block reverse current, making an external diode unnecessary. (A *bypass* diode is a different component, wired in parallel across panel cells to route current around partial shading, and does *not* stop night reverse current.) An MPPT charge controller (e.g., CN3791) improves harvest efficiency 15 - 30% over simple PWM controllers and is worthwhile for any deployment intended to last more than one year. Add an inline fuse (or polyfuse) on the battery positive lead for any outdoor lithium node.

Battery Selection

Never charge any lithium cell — LiPo, Li-ion, or LiFePO4 — below 0°C (32°F). Charging a sub-freezing lithium cell causes lithium plating, permanent capacity loss, and a risk of internal short and fire. This matters for solar nodes specifically: they will attempt to charge on cold, sunny winter mornings exactly when the battery is below freezing. Deployments where temperatures drop below 0°C must use a charge controller/PMIC with a low-temperature charge cutoff (NTC thermistor), or site/insulate the battery so it stays above freezing while charging. A bare TP4056 module has no low-temperature cutoff. Discharge is acceptable down to about −20°C, but charging is not.

- **LiPo (3.7 V, 2000 - 5000 mAh)** - Best energy density, wide availability, integrates directly with the RAK19007 PMIC. Never charge below 0°C (lithium plating, permanent damage). Discharge capacity also drops sharply below −20°C. Replace every 3 - 5 years.
- **18650 Li-ion** - More robust mechanically, with better low-temperature *discharge* performance — but the same sub-0°C charge prohibition still applies. Requires a separate holder and protection circuit unless using pre-protected cells. Useful when cylindrical cells fit the enclosure better.
- **AA lithium primary (e.g., Energizer L91)** - For locations where charging is infeasible. Rated to −40°C. A 4×AA pack in series is ~6 V (~3000 mAh); on a regulated rail this can run a sub-milliamp node for many months without any solar input (at 0.56 mA, ~220 days).

Size the backup battery for at least 7 days of autonomy without solar input. For a 0.56 mA node: 7 × 24 × 0.56 mA = 94 mAh minimum. A 2000 mAh LiPo provides roughly 100 days of reserve at 0.56 mA in ideal conditions, but cold reduces usable capacity and can block charging entirely below 0°C — so factor in winter derating rather than assuming the battery covers any overcast period.

Mounting and Deployment Best Practices

- Orient solar panels within 30° of due south (Northern Hemisphere) or due north (Southern Hemisphere) at an angle matching the site's latitude for optimal year-round harvest.
- Mount the enclosure in shade where possible (under eaves, north-facing surface) while keeping the panel in direct sun. High ambient temperature degrades LiPo capacity over time.

- Use stainless steel hose clamps for pole mounting. UV-resistant zip ties degrade within 2 - 3 years outdoors and are not adequate for permanent installation.
- Route cables with a drip loop before entering the enclosure cable gland to prevent water wicking along the cable jacket into the enclosure.
- Record GPS coordinates, orientation, panel angle, and photos of each node at installation. This data is invaluable for remote troubleshooting and future maintenance visits.

Data Integration

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

Item	Qty	Notes
RAK4631 Core Module	5	nRF52840 + SX1262
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.

Real-World IoT Applications

Water Quality and Flood Monitoring Networks

Environmental monitoring is one of the most compelling applications for LoRa mesh networks: sensors can be deployed in remote, power-limited locations that are difficult or expensive to reach with traditional wired or cellular connectivity.

Water Quality Monitoring

Mesh-connected water quality sensors provide real-time data for rivers, lakes, reservoirs, and municipal water systems. Note that water-quality sensors (pH, turbidity, dissolved oxygen, conductivity) require regular calibration to stay accurate:

Parameters to Monitor

- **pH** - Atlas Scientific EZO pH circuit (~$46) plus probe (~$85), roughly $130 total (probe sold separately; prices as of 2026-06-08). pH outside roughly 6.5-8.5 (the EPA secondary drinking-water range) may warrant investigation rather than proving contamination on its own.
- **Turbidity** - Measures water clarity; spikes indicate sediment runoff or contamination events
- **Dissolved Oxygen** - Critical for aquatic life; low DO indicates algal bloom or organic pollution
- **Conductivity/TDS** - Total dissolved solids; elevated levels indicate industrial runoff or saltwater intrusion
- **Water temperature** - DS18B20 waterproof probe (~$5); temperature affects biological and chemical processes
- **Water level** - Ultrasonic or pressure transducer sensor; critical for flood warning systems

Hardware Architecture

A complete water quality node:

- RAK4631 base (ultra-low power nRF52840)
- Atlas Scientific EZO carrier board (I2C) for multi-parameter sensing - e.g. the Whitebox Tentacle carrier
- DS18B20 waterproof temperature probe
- IP68 fiberglass enclosure with cable glands for sensor probes
- 10W solar panel + 10Ah LiFePO4 battery. This supports autonomous operation across most of the year in favorable sun, but do **not** assume unconditional year-round operation: LiFePO4 cells must **never be charged below 0°C (32°F)** - charging a frozen LiFePO4 cell causes lithium plating and permanent damage - so a charge controller / BMS with a low-temperature charge cutoff is required for winter water-side deployments.
- Transmit interval: 15-30 minutes (low duty cycle saves power and channel bandwidth)

Flood Early Warning Systems

The National Weather Service (NWS), working with USGS streamgages, is the authoritative source for flood warnings. A community LoRa mesh is a **supplementary, non-authoritative** monitoring aid - it does not replace NWS/USGS warnings or official alerts. Mesh delivery is best-effort and unacknowledged; packets can drop. Use a local mesh to monitor your own property as an early local indicator, but make life-safety and evacuation decisions based on official NWS/USGS warnings and local emergency alerts.

As a supplement, a creek or river monitoring network may give downstream residents some local advance notice. Any lead time (sometimes cited as 30-120 minutes) is watershed-dependent and not a guaranteed property - flash-flood watersheds can give far less. Treat it as a best-effort early indicator, not a guaranteed warning window:

1. Deploy water level sensors at upstream monitoring points (2-3 sensors per watershed)
2. Set alert thresholds. The percentages below are illustrative only, not a safe universal recipe: trigger on **rate-of-rise** (e.g., cm per minute) as well as absolute level, set conservative early thresholds, and align cut points with the official NWS flood-stage definitions for that specific gauge (per-site calibration with the local NWS/USGS gage). A 90%-of-flood-stage "Emergency" trigger can fire too late to evacuate, especially in flash-flood-prone watersheds. Example illustrative levels: "Advisory" ~50% of flood stage, "Warning" ~75%, "Emergency" ~90% - but do not hard-code these as universal.
3. Gateway node at the monitoring station forwards alerts to community mesh
4. Room server stores alerts and delivers to all connected community members
5. Integration with community alerting: Telegram bot, email, or siren activation

Case study framework: A network of 5 sensors along a 20-mile creek watershed, each transmitting hourly with a gateway node at the nearest road bridge, can give the downstream community a supplementary local indicator of rising water. This is an early local signal only - it does not replace NWS/USGS flood warnings, and residents should always treat official warnings and alerts as primary.

Data Management and Visualization

```
# Simple data pipeline for water monitoring:
# 1. Sensor node transmits JSON over LoRa mesh
# 2. Gateway node receives and publishes to MQTT
# 3. InfluxDB stores time-series data
# 4. Grafana displays dashboard with:
# - Current readings per sensor
# - Historical trend charts
# - Alert status indicators
# - Map overlay with sensor locations

# Sample query for a flood alert (threshold is illustrative only).
# Note: InfluxQL syntax shown below applies to InfluxDB 1.x;
# InfluxDB 2.x/3.x use Flux or SQL instead.
# SELECT last("level_cm") FROM water_sensors
# WHERE "location" = 'upstream_north'
# AND "level_cm" > 180 # illustrative alert threshold - calibrate per site
```

Air Quality and Environmental Monitoring Networks

Urban air quality monitoring is an underserved application for community mesh networks. Low-cost sensor nodes can build hyperlocal air quality maps that government monitoring stations - typically spaced miles apart - cannot provide. Keep in mind throughout that low-cost air-quality sensors are **not regulatory- or reference-grade**: their readings are indicative and supplementary, not authoritative.

Why Low-Cost Sensors Matter

Regulatory air quality monitors are sparse - often only a handful per metropolitan area - so a single station may represent tens of square miles (see EPA AQS network design criteria, 40 CFR Part 58 App. D). Regulatory reference monitors cost roughly $15K-$40K each, and a fully-equipped multi-pollutant station including siting and maintenance can run into the six figures; each represents a wide catchment area. Community mesh nodes with low-cost sensors (roughly $50-200 per node as a BOM estimate) can provide neighborhood-level data that reveals hotspots, traffic corridors, and industrial emission events invisible to the regional monitoring network. Community low-cost-sensor networks such as PurpleAir have demonstrated this hyperlocal value, but their readings are indicative rather than reference-grade.

Sensor Options for Air Quality

Parameter	Sensor	Cost	Accuracy	Notes

PM2.5 / PM10	Plantower PMS5003 or SDS011	$15-25	SDS011 ≈ max(±15%, ±10 µg/m³)	Most important for health; needs temperature correction. PMS5003 has no published absolute-accuracy spec, only consistency. Both are UART sensors - see the build note below on Meshtastic support.
CO_2	Sensirion SCD40 or SCD41	$35-50	±(50 ppm + 5% of reading)	True NDIR sensor; factory-calibrated, but relies on periodic fresh-air exposure (automatic self-calibration) to hold long-term accuracy.
VOCs	SGP30 or SGP41	$15-20	Semi-quantitative	Good for trend and event detection; not precise absolute levels
NO_2 / O_3	SPEC Sensors electrochemical	$50-100 each	±20 ppb (see SPEC Sensors datasheet)	Higher cost; good for near-road monitoring. Electrochemical sensors drift over time and need temperature/humidity compensation.
Temp + Humidity	BME280	$3-8	±0.5°C, ±3% RH	Essential for correcting PM sensor readings
Temp + Humidity	SHT31	$3-8	±0.3°C, ±2% RH	More accurate alternative for PM correction

Building a PM2.5 Monitoring Node

The following is a **design sketch, not a turnkey recipe**. The PMS5003 and SDS011 are UART laser particulate sensors, and stock Meshtastic's Telemetry module does **not** support them. For particulate matter in stock Meshtastic, use the I2C **PMSA003I** (I2C address 0x12), which the Telemetry module's air-quality metrics path supports. A UART PMS5003 requires custom firmware or a separate MCU/co-processor that injects readings into the mesh.

```
# Hardware: RAK4631 + RAK1906 (BME680) for temp/humidity correction
# For stock Meshtastic PM support, use the I2C PMSA003I (addr 0x12),
# NOT the UART PMS5003/SDS011 (those are not natively supported).

# Path A (recommended, no custom firmware):
#   PMSA003I (I2C) on a Meshtastic node -> AirQualityMetrics telemetry

# Path B (UART PMS5003): requires a co-processor or custom firmware.
#   Meshtastic environment telemetry uses fixed protobuf fields, so a
#   raw PMS5003 needs either:
#   - an ESP32 (e.g. T-Beam) running custom Arduino firmware that reads
#     PMS5003 + BME280 over UART/I2C and formats the data, or a custom
#     portnum, or
#   - a Raspberry Pi co-processor reading PMS5003 via UART and injecting
#     messages into the mesh via the Meshtastic Python API (see the
#     Meshtastic Python API send-message docs; clarify whether you send
#     plain text or structured telemetry).
# MeshCore note: there is no deployable "SENSOR firmware" that broadcasts
# PM telemetry; treat the working paths above as the only buildable ones.
```

Community Air Quality Network Design

For a neighborhood-scale (10-50 node) air quality network:

- **Spatial coverage:** 1 node per 0.5-1 km² in residential areas; denser near industrial sources and major roads
- **Transmission interval:** a recommended 5-15 minutes (PM sensors need averaging to reduce noise; see EPA sensor-performance guidance on averaging)
- **Data aggregation:** Central room server or MQTT gateway with InfluxDB backend
- **Public dashboard:** Grafana public dashboard showing real-time map (use Leaflet.js for geographic visualization)
- **Calibration:** Collocate 2-3 nodes with the nearest EPA monitoring station to develop calibration coefficients. Per EPA Air Sensor Toolbox collocation guidance, recommended collocation can run weeks to months and ideally spans multiple seasons.

Community Value and Partnerships

- Share data with PurpleAir, AirNow, or OpenAQ platforms for broader visibility
- Partner with local universities for calibration studies and data analysis
- Provide data to neighborhood environmental justice organizations
- Submit findings to local air quality district as citizen science data

Important: low-cost PM and gas sensors are not regulatory-grade. Readings require collocation/calibration and should always be presented as indicative, supplementary data - not as authoritative air-quality measurements. Do not use uncalibrated node readings as the basis for individual health decisions; defer to official AirNow/EPA data for that.