

Room Servers & Gateways

MeshCore room servers, Meshtastic MQTT gateways, and internet-bridging infrastructure.

- [MeshCore Room Servers](#)
 - [What is a Room Server?](#)
 - [Room Server Setup & Configuration](#)
 - [Dual Deployment: Repeater + Room Server](#)
- [Meshtastic MQTT Gateways](#)
 - [MQTT Overview & Setup](#)
 - [Home Assistant & Grafana Integration](#)
 - [Meshtastic MQTT Overview](#)
 - [Setting Up a Meshtastic MQTT Gateway](#)
 - [Running a Self-Hosted MQTT Broker](#)
 - [MQTT to Node-RED Integration](#)
- [Advanced Room Server Topics](#)
 - [Running Multiple Rooms](#)
 - [Internet Bridging and MQTT](#)
- [Gateway Setup Guides](#)
 - [Raspberry Pi MQTT Gateway Setup](#)
 - [Setting Up a Meshtastic MQTT-to-Internet Gateway](#)
- [Room Server Administration](#)

MeshCore Room Servers

What is a Room Server?

What is a MeshCore Room Server?

A **Room Server** is a store-and-forward bulletin board that runs directly on a LoRa node running MeshCore firmware. It acts as a shared message board for the mesh community - any node that can reach the room server can post and read messages without needing internet access or a central server. The whole point of a room server (versus an ordinary channel) is that it **retains message history**: roaming users who were out of range can come back later and pull the messages they missed, much like an email server.

Key Characteristics

- **Storage:** The room server **retains** message history and pushes it to clients when they connect - a client receives the previous 32 unseen messages on login. (Persistence across a reboot/power cycle depends on your firmware and hardware; do not assume messages survive a power loss without verifying on your firmware version.)
- **Access control:** Two passwords - an **admin** password and a **guest** password - plus a per-companion ACL (`setperm <pubkey> <level>`, with levels Guest / Read-only / Read-write / Admin) and a separate `set allow.read.only` flag for unauthenticated read access.
- **Repeater hybrid:** A room server can also be configured as a repeater with `set repeat on`, but this is a trade-off - room servers benefit from an indoor location for USB/Bluetooth administration, whereas repeaters perform best with a high external antenna.
- **Radio requirements:** The room server's radio preset (frequency/bandwidth/spreading factor/coding rate) must match the rest of the network; otherwise its packets and adverts cannot be received by other nodes.

Access Control

MeshCore room servers use two passwords plus a per-companion ACL - not a single "three-tier" password ladder:

Mechanism	Default	Permissions
Admin password	<code>password</code>	Full control: manage permissions, delete messages, change all settings
Guest password	<code>hello</code>	Read and write messages once authenticated as a guest

Mechanism	Default	Permissions
Per-companion ACL (<code>setperm</code>)	per node	Levels: Guest / Read-only / Read-write / Admin
<code>set allow.read.only</code> flag	off	When enabled, allows unauthenticated read-only access; off by default (no automatic no-password read)

Warning: Both default passwords must be changed immediately after first login. The default admin password `password` and guest password `hello` are publicly known.

Why Room Servers Matter

Room servers enable asynchronous community communication entirely over the mesh. A traveler passing through can read recent area messages, a neighborhood can coordinate during a power outage, or volunteers can leave notes for the next person who checks in - all without any internet connection or cellular infrastructure. Because the server stores and re-pushes missed messages, participants do not all need to be in range at the same time.

Room Server Setup & Configuration

Step 1 - Flash Room Server Firmware

Use the MeshCore web flasher at flasher.meshcore.io. Select your hardware (e.g., Heltec V3) and choose the **Room Server** firmware variant.

Step 2 - Connect via Serial or Bluetooth

Connect via USB serial at **115200 baud** using a terminal emulator (PuTTY, screen, minicom) or via Bluetooth using the MeshCore companion app.

Step 3 - Immediate Security: Change Default Passwords

Before anything else, change both default passwords. The defaults are publicly documented and must not remain in production:

```
password yourSecureAdminPassword
set guest.password yourSecureGuestPassword
```

These are the two passwords a room server uses (admin and guest). For finer control you can also assign per-companion permissions with `setperm <pubkey> <level>` (Guest / Read-only / Read-write / Admin), and enable unauthenticated read-only access with `set allow.read.only on` (default off). This is a two-password-plus-ACL model, not a single three-tier password ladder.

Step 4 - Set Node Identity and Location

```
set name MyRoomServer
set lat 46.879682
set lon -96.789803
```

Accurate coordinates allow the room server to appear correctly on network maps and help users gauge how far they are from it.

Step 5 - Verify Radio Preset

The room server's radio settings must match the network. The MeshCore USA/Canada preset (as of 2026-06-08) for most North American MeshCore networks ([NoDakMesh](#), [RegionMesh](#), etc.) is:

- Frequency: 910.525 MHz
- Bandwidth: 62.5 kHz
- Spreading Factor: SF7
- Coding Rate: CR5

Prefer applying the named USA/Canada preset from the firmware menu or companion app rather than hand-entering these values - that sets them automatically and avoids landing off the network if the preset has changed. Verify against the current MeshCore preset for your region before relying on the numbers above.

Step 6 - Optional: Enable Repeat Mode

If there is no separate repeater at this location, you can enable packet relay on the room server:

```
set repeat on
```

See [Dual Deployment: Repeater + Room Server](#) for why a dedicated repeater device is preferred over this option.

Step 7 - Broadcast Presence

Trigger an immediate advertisement so other nodes discover the room server without waiting for the next scheduled broadcast:

```
advert
```

Complete Configuration Reference

```
set guest.password yourGuestPassword
password yourAdminPassword
set name MyRoomServer
set lat 46.879682
```

```
set lon -96.789803
```

```
set repeat on # optional: also relay packets
```

```
advert # broadcast presence immediately
```

Dual Deployment: Repeater + Room Server

Dual Deployment: Separate Repeater and Room Server

The recommended best practice for any fixed site is to deploy **two separate devices** rather than enabling repeat mode on the room server.

Why Two Devices?

A repeater and a room server have conflicting placement requirements:

Device	Ideal Placement	Why
Repeater	Outdoors, elevated, high external antenna	Maximum RF range in all directions
Room Server	Indoors, near power and USB/BT access	Easy administration, configuration, and monitoring

Forcing one device to do both jobs means compromising on placement - either the repeater antenna is too low, or the room server is too hard to access for administration.

Recommended Hardware

- **Device 1 (Repeater):** Heltec V3 flashed with Repeater firmware + high-gain external antenna (fiberglass omni or directional yagi)
- **Device 2 (Room Server):** Heltec V3 flashed with Room Server firmware + stock indoor antenna

Cost

Two Heltec V3 units cost approximately **\$54** total (roughly \$27 each, as of 2026-06-08; pricing is volatile - verify current street price). This is a minimal investment for a significant capability improvement at a fixed site.

When Single-Device is Acceptable

If budget or space constraints make dual deployment impossible, enabling `set repeat on` on the room server is workable - it simply won't perform as well as a dedicated repeater at height. (`set repeat on` is the documented MeshCore CLI command for the repeater-hybrid mode; confirm against your firmware's CLI help output.) In that case, do your best to place the device near a window or in an elevated location to improve the antenna's line of sight.

Meshtastic MQTT Gateways

MQTT Overview & Setup

Meshtastic MQTT Gateway Overview

MQTT allows Meshtastic nodes to bridge the local LoRa mesh to the internet, enabling mesh messages to travel between physically separated mesh islands and feeding data into monitoring dashboards.

Hardware Requirements

A device can gateway to MQTT in several ways:

- **Onboard Wi-Fi (ESP32 only)** - the device connects to a Wi-Fi network directly. ESP32 boards with Wi-Fi include:
 - Heltec V3 / V4
 - T-Beam
 - T-Deck
 - Station G2
- **Ethernet** - the nRF52-based RAK4631 with the RAK13800 Ethernet module.
- **MQTT Client Proxy** - the node uses the connected phone app's internet connection, so it needs no onboard internet hardware at all. This works on any node, including nRF52 boards.

nRF52-based devices (most RAK Wireless WisBlock cores, e.g. the RAK4631, and some Lilygo variants) lack onboard Wi-Fi, so they cannot gateway over their own Wi-Fi. They **can** still use MQTT via Ethernet (RAK4631 + RAK13800) or via MQTT Client Proxy. The one caveat is that **nRF52 platforms cannot emit JSON over MQTT** - JSON output requires an ESP32 gateway.

Public MQTT Broker

The Meshtastic project operates a free public broker:

`mqtt.meshtastic.org`

The public broker is **not anonymous**. It requires credentials - the built-in defaults are username `meshdev` / password `large4cats`. The firmware auto-fills these when the fields are left blank, but external clients (MQTT Explorer, mosquitto_sub, Node-RED, Home Assistant) must supply them explicitly. The public broker also enforces restrictions (e.g., zero-hop, no broad # subscriptions, location-precision limits).

Topic Structure

```
msh/REGION/2/e/CHANNELNAME/USERID
```

The `2` segment is a fixed value in the Meshtastic MQTT protocol and does not change per channel. The encoding segment is the literal value `e` for encrypted protobuf (the default) or `json` for human-readable JSON. (Firmware before 2.3.0 used `/c/` instead of `/e/`.) The final segment is the gateway node's USERID. For example, a US node publishing JSON on the LongFast channel:

```
msh/US/2/json/LongFast/!a1b2c3d4
```

Configuration via Meshtastic CLI

Enable JSON encoding (human-readable messages on the broker):

```
meshtastic --set mqtt.json_enabled true
```

Enable uplink on channel 0 (mesh → MQTT). Uplink and downlink are per-channel settings and default to off:

```
meshtastic --ch-index 0 --ch-set uplink_enabled true
```

Enable downlink on channel 0 (MQTT → mesh):

```
meshtastic --ch-index 0 --ch-set downlink_enabled true
```

Best practice: Enable downlink on only **ONE** node on your local mesh. Because multiple gateway nodes can be connected to a single mesh, the same MQTT message can be injected into the mesh multiple times if several nodes have downlink enabled, causing duplicate messages and unnecessary channel congestion.

App Configuration Path

In the [Meshtastic app](#): **Settings > MQTT** (Android) or **Settings > Module Configuration > MQTT** (Apple)

- Enable MQTT: on
- Address: `mqtt.meshtastic.org`
- Root topic: `msh/US` (replace US with your region code)
- Username / Password: for the public broker, leave blank to use the built-in defaults (`meshdev` / `large4cats`), or set them explicitly. The public broker is not anonymous.
- TLS: enabled recommended

Home Assistant & Grafana Integration

Home Assistant & Grafana Integration

Once your Meshtastic node is publishing to an MQTT broker, that data stream can feed home automation and monitoring platforms.

Home Assistant Integration

Home Assistant has a built-in MQTT integration that can subscribe to Meshtastic topics.

1. Navigate to **Settings → Devices & Services**
2. Click **Add Integration** and search for **MQTT**
3. Configure with broker address: `mqtt.meshtastic.org`. The public broker is **not** anonymous - supply username `meshdev` and password `large4cats`. It also enforces restrictions (zero-hop, location-precision and topic limits), so for full data a self-hosted broker is recommended (see below).
4. Subscribe to your node's topic tree, e.g., `msh/US/2/json/LongFast/#`. Note the `/json/` topic only carries data when `mqtt.json_enabled` is true on the gateway (and JSON output is not available on nRF52 gateways); the public broker also filters JSON output.
5. Create template sensors to parse position, telemetry, and message payloads from the JSON. The fields live under `value_json.payload.*` in snake_case (e.g. `value_json.payload.temperature`, `value_json.payload.relative_humidity`, `value_json.payload.text`).

With location data flowing into Home Assistant, you can create dashboards showing node positions, battery levels, and last-seen times, or trigger automations when specific nodes report in.

Grafana Integration

Grafana is commonly used for time-series visualization of mesh telemetry (signal strength, battery voltage, node uptime).

- Community projects exist that expose Meshtastic MQTT data as Prometheus metrics. Because Prometheus scrapes HTTP endpoints rather than MQTT directly, such a project typically bridges MQTT to a metrics endpoint that Prometheus then scrapes.
- Search GitHub for **meshtastic prometheus exporter** for current community projects.
- A typical stack: Meshtastic node → MQTT broker → Node-RED or custom MQTT-to-metrics bridge → Prometheus → Grafana.

Self-Hosted MQTT Broker

For production or privacy-sensitive deployments, run your own MQTT broker (Mosquitto is the standard choice) rather than relying on the public broker. This also enables TLS with your own certificates and username/password authentication.

```
# Mosquitto install (Debian/Ubuntu)
sudo apt install mosquitto mosquitto-clients

# Mosquitto 2.0+ defaults allow_anonymous to false and binds localhost-only,
# so add a listener and auth before remote clients can connect, e.g. in
# /etc/mosquitto/conf.d/local.conf:
#   listener 1883 0.0.0.0
#   password_file /etc/mosquitto/passwd    (with: mosquitto_passwd -c /etc/mosquitto/passwd
#   <user>)
#   allow_anonymous false
# Configure TLS (listener 8883 with cafile/certfile/keyfile) and ACLs as needed.
```

Meshtastic MQTT Overview

Meshtastic's **MQTT module** extends a LoRa mesh network over the internet by bridging packets between the radio air interface and an MQTT message broker. A gateway node - a Meshtastic device with a working Wi-Fi or Ethernet connection - *uplinks* packets it hears over LoRa to the broker and optionally *downlinks* packets received from the broker back to the local mesh. This page explains the architecture, the public broker, and the trade-offs of the approach.

What MQTT Does in Meshtastic

MQTT (Message Queuing Telemetry Transport) is a lightweight publish-subscribe protocol designed for constrained devices and unreliable networks. In the Meshtastic context it serves as the internet backbone that connects isolated mesh islands:

- **Uplinking** - the gateway node listens for LoRa packets from nearby nodes, re-serialises them as MQTT payloads, and publishes them to the broker under a well-known topic hierarchy. The canonical topic is `msh/REGION/2/e/CHANNELNAME/USERID` for raw/encrypted protobuf, or `msh/REGION/2/json/CHANNELNAME/USERID` for JSON. The `2/e` (or `2/json`) segment is the protocol version and encoding; it is not optional.
- **Downlinking** - the gateway subscribes to the same topic hierarchy; when the broker delivers a message published by a gateway in another location, the local gateway broadcasts it over LoRa to its nearby mesh.

The net effect: two mesh islands in different cities share a common channel as if they were on the same local RF network - provided both have an internet-connected gateway and use the same channel encryption key.

Public MQTT Server: `mqtt.meshtastic.org`

The Meshtastic project operates a community MQTT broker at `mqtt.meshtastic.org` on port `1883` (unencrypted) and port `8883` (TLS). This server is free to use for experimentation and for joining the global mesh.

- **Requires credentials** - the public broker is *not* anonymous. Connect with the default username `meshdev` and password `large4cats` (pre-filled by the firmware/CLI). These are

shared community credentials, not per-user accounts; external clients (mosquitto_sub, MQTT Explorer, Node-RED, Home Assistant) must supply them explicitly or the connection is rejected.

- **No privacy guarantee** - any message published to the public broker is visible to anyone who subscribes to the same topic.
- **Unencrypted at the MQTT layer** - if you use the default channel key (`AQ==`, which is the single byte `0x01` - not all-zeros), your messages are readable by any MQTT subscriber. Always configure a custom channel key for any sensitive or private use.

“ **Privacy note:** Even with a strong channel key, MQTT metadata (sender node ID, timestamp, GPS coordinates if position sharing is enabled) may be visible to the broker operator and any subscriber on the same root topic. The public broker also enforces restrictions (zero-hop, location-precision limits, no broad # subscribe). Use a self-hosted broker for sensitive deployments.

Self-Hosted Broker Option

For deployments requiring privacy, control, or reliability guarantees, run your own MQTT broker (Eclipse Mosquitto is the standard choice). A self-hosted broker lets you:

- Require authentication (username + password or TLS client certificates).
- Restrict topic access with ACLs.
- Enable TLS on port 8883 with a Let's Encrypt or self-signed certificate.
- Bridge your private broker to the public one selectively.
- Log and retain messages for audit or replay.

MQTT Topic Hierarchy

Meshtastic publishes to a structured topic hierarchy:

```
msh/<region>/2/e/<channel_name>/!<node_id_hex>
```

Examples:

```
msh/US/2/e/LongFast/!abcd1234  
msh/EU_868/2/e/MyCommunity/!1a2b3c4d
```

- `msh` - root topic for all Meshtastic traffic.
- `<region>` - geographic region prefix (US, EU_868, AU_915, etc.).
- `2/e` - protocol version 2, encrypted protobuf payload. (Firmware prior to 2.3.0 published to a topic with `/c/` in place of `/e/`.)

- `<channel_name>` - the channel name configured on the node (e.g. LongFast, MyCommunity).
- `!<node_id_hex>` - the 8-character hexadecimal user/node ID, preceded by `!`. This is the ID of the **gateway node that published the packet**, not necessarily the original sender - in a multi-hop mesh the originating sender and the publishing gateway can differ.

The payload is a binary protobuf `ServiceEnvelope` containing the `MeshPacket` (which may be encrypted). On the wire the protobuf bytes are binary, not base64 (base64 only appears in some viewer UIs like MQTT Explorer). Only devices that share the same channel key can decrypt the inner packet.

Uplink vs. Downlink

Direction	What it does	When to enable
Uplink	Sends LoRa packets heard locally to the MQTT broker	Must be enabled per channel (<code>uplink_enabled</code>), which defaults to off
Downlink	Receives packets from the broker and broadcasts them over LoRa	When you want the local mesh to hear remote traffic

Caution with downlink: enabling downlink on multiple gateways in the same RF coverage area causes duplicate transmissions and wasted air time. In a coverage area with more than one gateway, designate a single gateway for downlink and disable it on the others.

JSON vs. Protobuf Payload Formats

Meshtastic supports two payload formats on MQTT:

- **Protobuf (default)** - binary-encoded, compact, but requires a protobuf decoder to read. This is the format used by actual Meshtastic devices and by the public broker.
- **JSON** - human-readable, easier to consume by Node-RED, Home Assistant, scripts, etc. Published to a parallel topic `msh/<region>/2/json/<channel>/!<node_id>`. Enable **JSON Enabled** in the [Meshtastic app](#) under *Settings > MQTT*. Note: JSON output is only available on ESP32 gateways - nRF52-based devices (RAK4631 and other RAK/WisBlock boards) cannot publish JSON; use the protobuf topic with a decoder instead. Only certain packet types are serialized to JSON (TEXT_MESSAGE, TELEMETRY, NODEINFO, POSITION, WAYPOINT, NEIGHBORINFO, TRACEROUTE, DETECTION_SENSOR, PAXCOUNTER, REMOTE_HARDWARE), not all traffic.

JSON mode is ideal for integration work but produces larger payloads. Disable it on resource-constrained brokers with many active nodes.

Common Use Cases

- **Global channel bridging** - join your local LongFast mesh to the worldwide LongFast MQTT cloud.
- **Remote node monitoring** - a solar-powered node deep in the field uplinks its telemetry (battery voltage, environment sensor data) to MQTT; you query the broker from home.
- **Alert forwarding** - a Node-RED flow subscribes to MQTT, detects specific messages, and forwards them to Telegram or Discord.
- **Mesh analytics** - pipe MQTT JSON output into InfluxDB/Grafana for network coverage and message delivery statistics.

Setting Up a Meshtastic MQTT Gateway

Any Meshtastic device with a direct internet connection can act as an MQTT gateway. This includes ESP32-based boards with onboard Wi-Fi (such as the LILYGO T-Beam or Heltec WiFi LoRa 32) and the nRF52-based RAK4631 paired with the RAK13800 Ethernet module. A node does *not* have to provide its own internet connection: with **MQTT Client Proxy**, any node (including nRF52 boards) can reach the broker through the connected phone app's internet connection. Note that nRF52 boards (e.g., RAK4631) cannot emit JSON over MQTT - JSON output requires an ESP32 gateway.

This guide covers enabling MQTT through the official [Meshtastic app](#) and verifying the connection with MQTT Explorer.

Prerequisites

- A Meshtastic node with a path to the internet: onboard Wi-Fi (ESP32), Ethernet (RAK4631 + RAK13800), or the phone's connection via MQTT Client Proxy.
- The Meshtastic Android / iOS app, or the Meshtastic Web Client.
- For an on-device internet connection, the node already joined to a Wi-Fi network (configure under *Network settings*: Settings > Network in the app, or Radio Config > Network on Web). With MQTT Client Proxy you can skip on-device Wi-Fi entirely.
- An MQTT broker - either the public `mqtt.meshtastic.org` or your own Mosquitto instance. The public broker is **not anonymous**: it requires credentials (default username `meshdev` / password `large4cats`), which the firmware auto-fills when the fields are left blank.

Step 1 - Connect the Node to Wi-Fi

In the Meshtastic app, open the device's config:

1. Navigate to *Network settings* (Settings > Network in the app, or Radio Config > Network on Web).
2. Toggle *Wi-Fi Enabled* on.
3. Enter your SSID and WPA2 passphrase.
4. Tap *Send*. The device will reboot and connect to Wi-Fi; confirm by checking the IP address shown on the device's OLED display or in the app's *Node Info* panel.

Alternatively, skip on-device Wi-Fi and use **MQTT Client Proxy** so the node connects to the broker through the connected phone app's internet connection.

Step 2 - Enable the MQTT Module

1. In the app, open *Settings > MQTT* (on Apple: *Settings > Module Configuration > MQTT*).
2. Toggle **MQTT Enabled** on.
3. Set **MQTT Server Address**:
 - Public broker: `mqtt.meshtastic.org`
 - Self-hosted: `192.168.1.50` or `mqtt.mynetwork.local`
4. For the public broker, leave **Username** and **Password** blank - the firmware will automatically use the built-in default credentials (`meshdev` / `large4cats`). This is **not** anonymous access. For an authenticated self-hosted broker, fill in your own credentials.
5. Set **Root Topic** - this prefixes all published/subscribed topics. Default: `msh`. Change only if you are segregating traffic (e.g., a private regional prefix like `msh/myregion`).
6. Toggle **JSON Output Enabled** if you intend to consume messages from Node-RED, Home Assistant, or custom scripts. Leave off for device-to-device bridging. (JSON is not supported on nRF52 gateways.)
7. Toggle **TLS Enabled** on if connecting to port 8883 (required for TLS brokers).
8. Tap *Send*. The device will apply the config and initiate MQTT connection.

Note: **Uplink Enabled** and **Downlink Enabled** are *per-channel* settings (configured under *Settings > Channels*), not toggles on the MQTT module screen. They default to *off*. See "Advanced: Uplink/Downlink per Channel" below - MQTT only carries traffic when the module is enabled *and* a channel has uplink (or downlink) enabled.

Step 3 - Configure Channel Encryption Key Handling

Messages are encrypted with the channel key on the LoRa air interface. The same encrypted payload is published to MQTT. The gateway does *not* decrypt messages before uplinking - it forwards the ciphertext as-is. This means:

- Subscribers on the MQTT broker who do not know the channel key see only opaque binary blobs.
- Remote Meshtastic devices with the same channel key can decrypt messages downlinked from the broker.
- For JSON mode, the gateway *does* decrypt before serialising to JSON. JSON output therefore exposes plaintext to whoever can subscribe on the broker - use a private broker with ACLs for JSON mode. Note that JSON only serializes a subset of packet types, is **not available on nRF52 boards**, and is published *alongside* (not instead of) the encrypted

protobuf topic - enabling JSON does not stop the encrypted `/e/` topic.

To share a channel key with a remote user, use the channel QR code or URL share feature in the app: *Channels* → *Share* produces a deep link containing the encoded key.

Step 4 - Verify with MQTT Explorer

MQTT Explorer is a free, cross-platform GUI client for exploring broker traffic.

1. Download from mqtt-explorer.com.
2. Create a new connection:
 - Protocol: `mqtt://` (port 1883) or `mqtts://` (port 8883).
 - Host: `mqtt.meshtastic.org` (or your broker).
 - For the public broker, enter Username `meshdev` and Password `large4cats`. These are required - unlike the Meshtastic firmware, a third-party client such as MQTT Explorer will be rejected by the broker if they are left blank.
3. Connect and expand the `msh` topic tree. Within seconds you should see topics appearing as your gateway uplinks packets.
4. Click a leaf topic to see the raw payload (raw/binary protobuf, which some clients display as base64) and, for JSON topics, the decoded JSON object.

```
# Alternatively, use the mosquitto_sub CLI tool:
```

```
mosquitto_sub -h mqtt.meshtastic.org -p 1883 -u meshdev -P large4cats -t "msh/#" -v
```

Advanced: Uplink/Downlink per Channel

Meshtastic 2.x allows per-channel MQTT uplink/downlink settings, so you can uplink a private community channel while keeping a sensitive channel (e.g., encrypted tactical) RF-only:

1. In the app, navigate to *Settings* > *Channels*.
2. Edit the channel you want to configure.
3. Under *Uplink Enabled* / *Downlink Enabled*, set the per-channel overrides. These default to *off* for all channels.
4. MQTT requires *both* the MQTT module to be enabled *and* at least one channel to have uplink (or downlink) enabled. The module toggle alone does not uplink any channel - if the module is on but no channel is enabled, nothing flows.

Common Issues and Fixes

Symptom	Cause	Fix
Device shows "MQTT connecting" indefinitely	Wrong broker address or firewall blocking port 1883	Ping broker from device's LAN; check firewall rules
No topics appear in MQTT Explorer	Uplink not enabled, or no LoRa traffic to uplink	Confirm Uplink is on; transmit a message from another node
Remote nodes not seen locally after downlink	Channel key mismatch between islands	Share channel URL; confirm both ends use identical key bytes
Duplicate messages on the mesh	Multiple gateways have downlink enabled in same RF area	Disable downlink on all but one gateway per coverage area
TLS handshake failure	Clock skew on device; certificate expired	Enable NTP on node; renew/replace TLS cert on broker

Running a Self-Hosted MQTT Broker

Eclipse **Mosquitto** is the de-facto standard open-source MQTT broker, widely used with Meshtastic for private mesh deployments. This guide installs Mosquitto on a Raspberry Pi or Ubuntu/Debian server, secures it with TLS, adds authentication, and optionally bridges it to the public `mqtt.meshtastic.org` server.

Prerequisites

- A Linux host (Raspberry Pi OS, Ubuntu 22.04, Debian 12) with a static LAN IP or a public IP / DDNS hostname.
- A domain name or static IP for the TLS certificate (required for Let's Encrypt; a self-signed cert works for LAN-only deployments).
- Ports 1883 (MQTT) and/or 8883 (MQTT over TLS) open on the host firewall and your router NAT if internet-accessible.

Step 1 - Install Mosquitto

```
sudo apt update
sudo apt install -y mosquitto mosquitto-clients

# Confirm the service is running
sudo systemctl status mosquitto
```

On Mosquitto 2.0+, a fresh install only accepts connections from the local machine and anonymous access is disabled by default (`allow_anonymous false`) - it will **not** accept remote clients on 1883 until you define a listener and configure authentication. (Older 1.x versions did listen on 1883 with anonymous access, which is the source of the common misconception.) In all cases, explicitly configure a listener, authentication, and TLS before any internet exposure.

Step 2 - Minimal Configuration for Meshtastic

Mosquitto's main config is at `/etc/mosquitto/mosquitto.conf`. The `conf.d/` directory is for drop-in configs. Because a 2.0+ install binds localhost-only with no listener, you **must** define a listener bound to `0.0.0.0` (plus authentication) for other hosts - including your Meshtastic nodes - to connect. Create a Meshtastic-specific config:

```
sudo nano /etc/mosquitto/conf.d/meshtastic.conf
```

```
# /etc/mosquitto/conf.d/meshtastic.conf

# — Plain-text listener (LAN only or behind VPN) —————
# Required: without an explicit listener, Mosquitto 2.0 binds localhost only
listener 1883 0.0.0.0
protocol mqtt

# Require authentication on all listeners
allow_anonymous false
password_file /etc/mosquitto/passwd

# — TLS listener —————
listener 8883 0.0.0.0
protocol mqtt
tls_version tlsv1.2

# For a Let's Encrypt SERVER cert, point certfile at fullchain.pem and
# keyfile at privkey.pem. Do NOT set cafile here unless you are requiring
# CLIENT certificate auth - cafile verifies clients, not the server chain.
certfile /etc/letsencrypt/live/mqtt.example.com/fullchain.pem
keyfile /etc/letsencrypt/live/mqtt.example.com/privkey.pem

# — Persistence —————
persistence true
persistence_location /var/lib/mosquitto/

# — Logging —————
log_dest file /var/log/mosquitto/mosquitto.log
```

```
log_type all
```

Step 3 - Add User Authentication

```
# Create the password file and add a user
sudo mosquitto_passwd -c /etc/mosquitto/passwd meshuser
# Enter password at prompt (choose a strong password)

# Add additional users (omit -c flag to append rather than recreate)
sudo mosquitto_passwd /etc/mosquitto/passwd seconduser
```

Step 4 - TLS Certificate (Let's Encrypt)

If your broker has a public domain name:

```
sudo apt install -y certbot
sudo certbot certonly --standalone -d mqtt.example.com --agree-tos --email admin@example.com

# Certbot stores certs in /etc/letsencrypt/live/mqtt.example.com/
# Mosquitto must be able to read the key
sudo usermod -aG ssl-cert mosquitto # Debian; or adjust file perms directly
```

For LAN-only deployments without a public domain, generate a self-signed certificate:

```
sudo mkdir -p /etc/mosquitto/tls
cd /etc/mosquitto/tls

sudo openssl req -x509 -newkey rsa:4096 -keyout server.key -out server.crt -days 3650 -nodes -
subj "/CN=mqtt.local/O=MeshAmerica/C=US"

sudo chown mosquitto:mosquitto server.key server.crt
sudo chmod 600 server.key

# Update meshtastic.conf to use these paths:
# certfile /etc/mosquitto/tls/server.crt
```

```
# keyfile /etc/mosquitto/tls/server.key
```

Meshtastic's MQTT module exposes only a *TLS Enabled* toggle - there is **no** in-app/firmware option named "TLS Verify Cert" to turn certificate verification off. When TLS is enabled the node validates the server certificate, so a self-signed certificate generally requires the node to trust the issuing CA. The simplest reliable path is to use a publicly-trusted certificate from Let's Encrypt rather than a self-signed cert. Self-signed CA-import support varies by firmware version, so verify current behavior on your build.

Step 5 - Open Firewall Ports

```
# Using ufw
sudo ufw allow 1883/tcp comment "MQTT plain"
sudo ufw allow 8883/tcp comment "MQTT TLS"
sudo ufw reload
sudo ufw status
```

Step 6 - Reload and Test

```
sudo systemctl reload mosquitto
sudo systemctl status mosquitto

# Test from local machine (no TLS)
mosquitto_sub -h localhost -p 1883 -u meshuser -P yourpassword -t "msh/#" -v &
mosquitto_pub -h localhost -p 1883 -u meshuser -P yourpassword -t "msh/test" -m "hello"

# Test TLS from a remote machine (using cert bundle)
mosquitto_sub -h mqtt.example.com -p 8883 --cafile /etc/ssl/certs/ca-certificates.crt -u
meshuser -P yourpassword -t "msh/#" -v
```

Step 7 - Bridging to mqtt.meshtastic.org (Optional)

To merge your private broker's traffic with the public network, configure a bridge. Note that the public `mqtt.meshtastic.org` broker is **not anonymous** - it requires the shared credentials `meshdev` /

`large4cats` - and it now **restricts broad topic subscriptions** (you can no longer subscribe to the whole `#` tree). Review Meshtastic's public-broker policy before bridging, and prefer narrow, channel-specific topics rather than a wide `msh/REGION/#` firehose.

```
sudo nano /etc/mosquitto/conf.d/bridge_meshtastic.conf
```

```
# /etc/mosquitto/conf.d/bridge_meshtastic.conf

connection meshtastic_public
address mqtt.meshtastic.org:1883

# Bridge a narrow, channel-specific topic (the public broker now blocks
# broad subscriptions like msh/US/#). Adjust to your region/channel.
topic msh/US/2/e/LongFast/# both 0

# Bridge credentials - the public broker is NOT anonymous; these shared
# credentials are required or the bridge connection is refused.
remote_username meshdev
remote_password large4cats

# Reconnect after 30 seconds if connection drops
restart_timeout 30
keepalive_interval 60

# Use a persistent queue so messages don't drop if bridge disconnects
cleansession false
```

```
sudo systemctl reload mosquitto
journalctl -u mosquitto -f | grep bridge
```

Access Control Lists (ACLs)

Restrict which users can publish or subscribe to which topics:

```
sudo nano /etc/mosquitto/conf.d/meshtastic.conf
# Add inside the config file:
acl_file /etc/mosquitto/acls

sudo nano /etc/mosquitto/acls
```

```
# /etc/mosquitto/acls
# Format: topic [read|write|readwrite] <topic_pattern>
# Rules placed BEFORE the first "user" line apply to anonymous clients.
# Deny rules are evaluated before grants, so never put a blanket
# "topic deny #" inside a user block - it would override that user's
# own grants and lock them out.

# meshuser can read and write all msh topics
user meshuser
topic readwrite msh/#

# readonly user can only subscribe, not publish
user readonly
topic read msh/#
```

To block anonymous clients entirely, the cleanest approach is simply `allow_anonymous false` (set above). If you instead want an explicit ACL deny for anonymous access, place `topic deny #` **before** any `user` line - not inside a user block.

```
sudo systemctl reload mosquitto
```

Keeping Certificates Renewed

```
# Certbot auto-renewal cron is installed at /etc/cron.d/certbot
# After renewal, Mosquitto must reload to pick up new certs:
sudo tee /etc/letsencrypt/renewal-hooks/deploy/mosquitto.sh <<'EOF'
#!/bin/bash
systemctl reload mosquitto
EOF
sudo chmod +x /etc/letsencrypt/renewal-hooks/deploy/mosquitto.sh
```

MQTT to Node-RED

Integration

Node-RED is a browser-based visual programming tool built on Node.js, ideal for wiring together MQTT data streams from Meshtastic with downstream services such as dashboards, databases, and messaging platforms. This page covers installing Node-RED, subscribing to Meshtastic MQTT topics, parsing protobuf and JSON payloads, building a simple monitoring dashboard, and forwarding mesh alerts to Telegram and Discord.

Installing Node-RED

On Raspberry Pi / Debian / Ubuntu

Use the official Node-RED install script for Debian/Ubuntu (it provisions a supported Node.js version). Review the script before running it, per nodered.org:

```
bash <(curl -sL https://github.com/node-red/linux-installers/releases/latest/download/update-nodejs-and-nodered-deb)

# Enable and start Node-RED as a service
sudo systemctl enable nodered.service
sudo systemctl start nodered.service

# Node-RED web UI is available at http://<IP>:1880
```

On x86 Linux via npm (Node.js v20+ already installed)

The bare npm method is only for environments where a supported Node.js (v20+) is already present. Install the build prerequisites first:

```
sudo apt install build-essential git curl
sudo npm install -g --unsafe-perm node-red
# Start manually:
node-red
# Or install as systemd service manually
```

Installing Required Palettes

Open the Node-RED web UI at `http://<IP>:1880`, then click the hamburger menu → *Manage Palette* → *Install*. Search for and install:

- `node-red-dashboard` - UI widgets (gauges, charts, text displays).
- `node-red-contrib-protobuf` - decode Meshtastic protobuf payloads.
- `node-red-contrib-telegrambot` - send messages to Telegram.

The Meshtastic project also publishes an official decode node, `@meshtastic/node-red-contrib-meshtastic` (linked from meshtastic.org), which you can install the same way.

Alternatively via CLI

```
cd ~/.node-red
npm install node-red-dashboard node-red-contrib-protobuf node-red-contrib-telegrambot
sudo systemctl restart nodered.service
```

Subscribing to Meshtastic MQTT Topics

Add an **mqtt in** node to your flow:

- **Server:** your broker address and port. (For the Meshtastic public broker `mqtt.meshtastic.org`, supply username `meshdev` / password `large4cats` - it is not anonymous.)
- **Topic:** `msh/#` to receive all Meshtastic traffic, or `msh/US/2/json/LongFast/#` for JSON output from the LongFast channel. JSON is only published when JSON mode is enabled on the gateway (`mqtt.json_enabled true`); note that `msh/#` also yields protobuf-encoded `/e/` topics that the json node cannot parse.
- **QoS:** 0 (at most once) is sufficient for mesh monitoring.

- **Output:** for protobuf `/e/` topics set Output to a *buffer*; for JSON topics a *parsed JSON object* / string is fine.

Parsing Protobuf Payloads

The protobuf format requires `.proto` schema files. The MQTT envelope type `ServiceEnvelope` is defined in `mqtt.proto`, so download that plus its imports (`mesh.proto`, `config.proto`, `portnums.proto`, etc.) into the same directory:

```
mkdir -p ~/.node-red/proto
cd ~/.node-red/proto
wget https://raw.githubusercontent.com/meshtastic/protobufs/master/meshtastic/mqtt.proto
wget https://raw.githubusercontent.com/meshtastic/protobufs/master/meshtastic/mesh.proto
wget https://raw.githubusercontent.com/meshtastic/protobufs/master/meshtastic/config.proto
wget https://raw.githubusercontent.com/meshtastic/protobufs/master/meshtastic/portnums.proto
wget https://raw.githubusercontent.com/meshtastic/protobufs/master/meshtastic/telemetry.proto
```

Flow wiring for protobuf decode. The encrypted `/e/` MQTT payload is **raw binary protobuf** (not base64), so feed the buffer straight into the decode node:

```
[mqtt in (Output: a buffer)] → [protobuf decode (ServiceEnvelope)] → [debug / further processing]
```

Set the **mqtt in** node's Output to a *buffer* and add a **function** node before the protobuf decode that points it at the right schema and type (do **not** base64-decode the payload first):

```
// The /e/ payload is already a raw binary protobuf buffer - pass it straight through.
msg.protofile = '/home/pi/.node-red/proto/mqtt.proto';
msg.protobufType = 'meshtastic.ServiceEnvelope';
return msg;
```

Base64 only applies to the *nested* `decoded.payload` bytes (for example when extracting `TEXT_MESSAGE_APP` text after the envelope is decoded) - not to the MQTT payload itself.

The **protobuf decode** node only deserializes the bytes into a JavaScript object representing the `ServiceEnvelope`. On encrypted channels the inner `packet.decoded` field is *absent* - the payload is still AES-encrypted with the channel PSK, which must be decrypted separately (or enable JSON output on the gateway, which decrypts before publishing).

“ **Note on channel decryption:** The official `@meshtastic/node-red-contrib-meshtastic` package provides a Meshtastic protobuf decode node. The `protobuf`

`decode` node by itself only deserializes the envelope; for encrypted channels you must AES-decrypt with the channel PSK, or enable JSON output mode on the gateway (which the gateway decrypts before publishing) for full access to message text and telemetry.

Parsing JSON Payloads

(Recommended for Simplicity)

If JSON output is enabled on your gateway node (`mqtt.json_enabled true`, ESP32 only - nRF52 gateways do not emit JSON), subscribe to the JSON topic and use a **json** node to parse directly - no protobuf library needed:

```
[mqtt in (topic: msh/US/2/json/LongFast/#)] → [json] → [switch] → [handlers]
```

A decoded JSON message is a flat envelope and looks like:

```
{
  "id": 3456789012,
  "channel": 0,
  "from": 2887456789,
  "to": 4294967295,
  "sender": "!abcd1234",
  "type": "text",
  "payload": {
    "text": "Hello from City A!"
  },
  "timestamp": 1714953600
}
```

Note: the JSON envelope does **not** include RF metadata - there is no top-level `rxssi`, `snr`, or `hops_away` field. Those values are not available via gateway JSON output; if you need them, obtain them from a local serial/BLE API or from the protobuf `MeshPacket` (`rx_rssi`/`rx_snr`) instead.

Building a Simple Dashboard

With `node-red-dashboard` installed, add a UI group for "Mesh Monitor":

1. Add a **ui_text** node: label "Last Message", wired from the JSON parse output using a function to extract `msg.payload.payload.text`.
2. Add a **ui_chart** node: "Message Rate / min", use an **rbe** (report-by-exception) node plus a counter function to track message frequency.
3. Add a **ui_table** node: "Recent Messages" - build a rolling array of the last 20 messages using a context store.

RF signal gauges (RSSI/SNR) cannot be driven from the JSON MQTT feed because those fields are not present in the JSON envelope. If you want an RSSI display, source the value from a non-MQTT-JSON path (local serial/BLE API, or the protobuf `MeshPacket.rx_rssi` on the `/e/` decode path).

Access the dashboard at `http://<IP>:1880/ui`.

Forwarding Alerts to Telegram

Pre-requisite: create a Telegram Bot via @BotFather and note the bot token and your chat ID.

Wire a flow like this:

```
[mqtt in] → [json] → [switch: msg.payload.type == "text"] → [function: format alert] → [telegram sender]
```

Function node code to format the Telegram message:

```
const p = msg.payload;
if (!p.payload || !p.payload.text) return null; // skip non-text packets

msg.payload = {
  chatId: "YOUR_CHAT_ID_HERE",
  type: "message",
  content: `[] Mesh Message
` +
  `From: ${p.sender}
` +
  `Text: ${p.payload.text}`
};
return msg;
```

Configure the **Telegram Sender** node with your bot token. (RSSI/SNR/hops are intentionally omitted - they are not present in the JSON envelope.)

Forwarding Alerts to Discord

Use a **http request** node pointed at your Discord webhook URL:

```
[mqtt in] → [json] → [switch: msg.payload.type == "text"] → [function: build webhook body] → [http request]
```

Function node code:

```
const p = msg.payload;
if (!p.payload || !p.payload.text) return null;

msg.method = "POST";
msg.url = "https://discord.com/api/webhooks/YOUR_WEBHOOK_ID/YOUR_WEBHOOK_TOKEN";
msg.headers = { "Content-Type": "application/json" };
msg.payload = JSON.stringify({
  username: "MeshAmerica Bot",
  content: `**Mesh Message** from \`${p.sender}\`
  +
  > ${p.payload.text}`
});
return msg;
```

Example: Full Flow JSON Export

Below is a minimal Node-RED flow (importable via *Import* → *Clipboard*) that subscribes to the JSON MQTT topic, parses messages, displays them in a dashboard, and forwards them to a Discord webhook:

```
[
  {
    "id": "mqtt_in_1",
    "type": "mqtt in",
    "name": "Meshtastic JSON",
    "topic": "msh/US/2/json/LongFast/#",
    "qos": "0",
    "broker": "local_broker",
    "wires": [["json_parse_1"]]
```

```

},
{
  "id": "json_parse_1",
  "type": "json",
  "name": "Parse JSON",
  "wires": [["switch_1", "dashboard_table_1"]]
},
{
  "id": "switch_1",
  "type": "switch",
  "name": "Text only",
  "property": "payload.type",
  "rules": [{"t": "eq", "v": "text"}],
  "wires": [["format_discord_1"]]
},
{
  "id": "format_discord_1",
  "type": "function",
  "name": "Format Discord",
  "func": "const p = msg.payload;
if (!p.payload||!p.payload.text) return null;
msg.method='POST';
msg.url='https://discord.com/api/webhooks/XXX/YYYY';
msg.headers={'Content-Type':'application/json'};
msg.payload=JSON.stringify({username:'MeshBot',content:`**${p.sender}**:${p.payload.text}`});
return msg;",
  "wires": [["http_discord_1"]]
},
{
  "id": "http_discord_1",
  "type": "http request",
  "name": "Discord Webhook",
  "method": "POST",
  "wires": [[]]
}
]

```

Replace the Discord webhook URL placeholder with your real webhook before importing.

Advanced Room Server Topics

Running Multiple Rooms

A MeshCore room server is a single node that hosts a **single room**. There is no multi-room mode: one room server node = one room. To offer several separate spaces across a community (for example, a public room and a private emergency operations room), you deploy **multiple room-server nodes** — one node per room. This page covers how a single room server works and how to run several of them as a coordinated set.

How a room server works

A MeshCore room server is a store-and-forward node configured entirely over its serial/BLE CLI — there is **no configuration file**. A single room server has:

- One advertised **node name**, set with `set name <name>` (this is the name clients see; there is one name per node).
- One **admin password**, changed with `password <new>`, used for administrative control of the node.
- One **guest password**, changed with `set guest.password <password>` (default: `hello` — change it immediately). This gates guest access to the room.
- One per-contact **access control list (ACL)**, managed with `setperm <pubkey> <level>` and viewed with `get acl`. Permission levels are Guest (0), Read-only (1), Read-write (2), and Admin (3).
- A single **read-only flag**, `set allow.read.only on|off` (default: **off**). When on, it permits unauthenticated read-only access; by default unauthenticated read is not granted.
- A single persisted **message history** store. The room server retains posts and pushes the unseen backlog to a client when that client next connects (a client receives the previously unseen messages on login — documented as the previous 32). This persistence is the entire point of a room server versus an ordinary channel.

A client connects to a room server (supplying the guest password if one is required) to access that server's single room. There is no "choose which room to join" selection, because a server hosts only one room. A client can, however, hold multiple room servers as contacts and interact with each separately.

Exact password defaults, ACL levels, and the unseen-message count should be verified against your firmware version and the current MeshCore CLI reference at docs.meshcore.io/cli_commands (as of 2026-06-08).

Deploying multiple room servers

Because one node hosts one room, separate spaces require separate nodes. A community that wants a public room and a restricted emergency room runs two room-server nodes, each configured independently over its own CLI. Each node is a separate piece of hardware (for example a Heltec V3 or an nRF52840 board) with its own name, passwords, ACL, and history store.

For example, to set up three room servers across a region you would, on each node in turn, connect over serial/BLE and configure it:

```
# --- Node 1: public regional room ---
set name RegionMesh-Public
password <newAdminPassword>
set guest.password <publicGuestPassword> # or leave blank for open guest access
set allow.read.only on # optional: allow unauthenticated read-only
advert

# --- Node 2: restricted emergency room (separate hardware) ---
set name EmergencyNet
password <newAdminPassword>
set guest.password <emergencyGuestPassword>
setperm <coordinatorPubkey> 2 # grant a coordinator Read-write
advert

# --- Node 3: club room (separate hardware) ---
set name ClubNet
password <newAdminPassword>
set guest.password <clubGuestPassword>
advert
```

All of these nodes — like every node in the network — must share the same radio preset (frequency, bandwidth, spreading factor, coding rate) as the companions and repeaters around them; otherwise their packets and adverts cannot be received by other nodes. Use the named region preset in the app or web flasher rather than hand-entering values. Give each node a clear, distinct name so users can tell the rooms apart.

Controlling access to each room

Access to a room is governed by the room server's guest password and per-contact ACL, not by a per-room PSK. Anyone who has a room's guest password can join and read that room's messages,

so treat it like a shared secret: distribute it through a separate secure channel (encrypted email, in person, Signal, etc.), not over the mesh itself, and rotate it with `set guest.password <new>` if it may have leaked.

For finer control on a given node, grant individual contacts specific permission levels with `setperm <pubkey> <level>` (Guest/Read-only/Read-write/Admin) and review the current list with `get acl`. For a restricted emergency room, the guest password is typically shared only with vetted local emergency-management personnel, ARES/RACES members, and CERT team leads — not published publicly — and trusted operators can be given Read-write or Admin via the ACL.

Monitoring a room server

A MeshCore room server has no HTTP status endpoint and no management script — it is firmware on a microcontroller (nRF52840/ESP32). You monitor it over the serial/BLE CLI using the built-in stats commands:

```
# System stats: battery, uptime, queue length, debug flags
stats-core

# Radio-layer stats
stats-radio

# Packet-level stats
stats-packets
```

These commands are issued over the serial console (or the companion app's CLI). To monitor several rooms, connect to each room-server node and query it individually.

Internet Bridging and MQTT

A MeshCore room server runs as firmware on a single LoRa node (typically nRF52840 or ESP32 hardware). It is a store-and-forward node on the RF mesh and does **not** have a native internet, TCP, or MQTT bridge. There is no MeshCore feature that lets phone users without LoRa hardware join a room over the internet, and there is no MeshCore MQTT integration. If you want MeshCore-related traffic on the internet or in a monitoring stack, you bridge to a separate Linux host, or use Meshtastic's documented MQTT path on a co-located gateway. This page explains what MeshCore actually supports and how to do internet/monitoring integration correctly.

What "bridge" means in MeshCore

MeshCore's only built-in "bridge" is a **radio-layer link**, compiled into the firmware, used to join two co-located boards so they extend RF coverage. It is configured through the device CLI (over serial/BLE), for example:

- `set bridge.enabled <on|off>` - enable the compiled-in bridge (only present when bridge support is built into the firmware)
- The transport is a radio/serial link (RS-232 serial or ESP-NOW between two boards), not a TCP/internet connection

This bridge does **not** relay between LoRa radio and the internet, and it does not let off-mesh clients participate over TCP. MeshCore companion clients (phone, computer) connect to a node **locally** over BLE, USB, or serial - not over the internet to a remote room server. A person in another city cannot join your local mesh purely through a MeshCore room server; reaching the internet requires a separate gateway host (see below).

Getting mesh data to the internet (the supported paths)

MeshCore room-server firmware does not publish to MQTT. If you need mesh traffic on an MQTT broker, a dashboard, or the wider internet, use one of these real options:

- **Run a separate Linux gateway host.** Connect a MeshCore node by USB/serial to a small Linux machine (Raspberry Pi, mini-PC) and run your own software on that host to read the node's serial output and forward it wherever you like. The microcontroller

running the room server cannot host this software itself - it has on the order of 256 KB of RAM.

- **Use Meshtastic's MQTT path for an internet-connected mesh.** Meshtastic firmware *does* document native MQTT (uplink/downlink to a broker such as `mqtt.meshtastic.org` or a self-hosted Mosquitto). If internet integration is a hard requirement, a Meshtastic gateway node is the documented way to do it. See the Meshtastic MQTT pages in this book.

Do not expect MeshCore CLI keys such as a `mqtt:` config block, a `topic_prefix`, or `meshcore/...` topics - they do not exist in MeshCore firmware.

Monitoring stack (InfluxDB / Telegraf / Grafana) runs on a separate host

A time-series monitoring stack is a legitimate way to visualize mesh data, but every component runs on a **separate Linux host**, never on the nRF52840/ESP32 microcontroller of the room server:

1. Install InfluxDB (time-series database) on a Linux host.
2. Feed it from a real data source - for a Meshtastic mesh, Telegraf's MQTT input plugin can consume the Meshtastic MQTT topics and write to InfluxDB. (There is no MeshCore MQTT feed; for MeshCore you would write your own collector that reads the node's serial output on the gateway host.)
3. Install Grafana on the host and build dashboards (active nodes over time, node battery levels, message rates, coverage from GPS data).

This entire stack lives on the Linux host, not on the MeshCore room-server hardware.

Exposing a Linux gateway host to the internet (TLS)

If - and only if - you are running an actual web/dashboard service on a separate Linux gateway host (for example a Grafana instance or a custom collector's web UI), you can put it behind a reverse proxy with TLS. This nginx example terminates TLS in front of a local service; replace the upstream port with whatever your service actually listens on (e.g. Grafana's default `3000`). A MeshCore room server itself exposes no HTTP service, so there is nothing on the node to proxy to.

```
# Example nginx configuration on a Linux gateway host
# (fronting a real local service, e.g. Grafana on :3000)
server {
    listen 443 ssl;
    server_name mesh.yournetwork.com;

    ssl_certificate /etc/letsencrypt/live/mesh.yournetwork.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/mesh.yournetwork.com/privkey.pem;

    location / {
        proxy_pass http://127.0.0.1:3000; # your service's actual port
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
    }
}
```

Get free TLS certificates from Let's Encrypt using certbot. TLS protects the connection to that gateway host's service; it has nothing to do with the RF mesh, which is encrypted independently at the LoRa layer.

Security considerations for an internet-facing gateway host

Any Linux host you expose to the internet (broker, dashboard, collector) needs proper hardening:

- **TLS/SSL:** Terminate TLS on the gateway host's web/broker service so credentials and data are not sent in plaintext.
- **Authentication:** Require authentication on the service. An open dashboard or broker reachable from the internet attracts abuse.
- **Firewall rules:** Restrict access to only the ports the service needs; block everything else and front the service with a reverse proxy for TLS termination.
- **Rate limiting:** Apply per-client limits so a single client cannot flood the service.

Alerting on node failure (on the gateway host)

If you are collecting *Meshtastic* data via MQTT on your Linux host, you can alert when a node stops checking in using a small Python script (this consumes the Meshtastic MQTT topics, not any MeshCore topic):

```
import paho.mqtt.client as mqtt
import time

nodes = {}
ALERT_TIMEOUT_SECONDS = 3600 # Alert if not heard in 1 hour

def on_message(client, userdata, msg):
    node_id = msg.topic.split('/')[1]
    nodes[node_id] = time.time()

def check_timeouts():
    now = time.time()
    for node_id, last_seen in nodes.items():
        if now - last_seen > ALERT_TIMEOUT_SECONDS:
            print(f"ALERT: {node_id} has not been heard in over 1 hour!")

# Subscribe to the Meshtastic JSON topic tree on your broker
# (requires mqtt.json_enabled on the gateway; not available on nRF52)
client = mqtt.Client()
client.on_message = on_message
client.connect("localhost", 1883)
client.subscribe("msh/+/2/json/+/+")
client.loop_start()

while True:
    check_timeouts()
    time.sleep(300)
```

For a MeshCore-only deployment there is no MQTT feed to subscribe to; you would instead read the node's serial output (and its `stats-core` / `stats-radio` / `stats-packets` CLI output) on the gateway host and build alerting from that.

Gateway Setup Guides

Step-by-step setup guides for Raspberry Pi MQTT gateways and MeshCore Linux room servers.

Raspberry Pi MQTT Gateway Setup

What This Achieves

The Meshtastic node acts as the MQTT gateway, bridging LoRa packets to an MQTT broker (it uplinks/downlinks LoRa<->MQTT using its own network connection or the connected client's). Mosquitto running on the Raspberry Pi is just the broker: it provides a persistent, always-on broker and host so you do not need a phone or laptop running the [Meshtastic app](#). The node must have `mqtt.enabled` set and point at the broker for any bridging to happen - running Mosquitto alone does not bridge the mesh. Once configured it forwards mesh packets to MQTT (locally and/or to a cloud broker), enables remote monitoring of all nodes on your mesh, and allows the node to relay messages between the mesh and internet-connected services.

Hardware Requirements

Component	Notes
Raspberry Pi 3B+, 4, or Zero 2W	3B+ or 4 for comfort; Zero 2W for power-constrained installs. All run Pi OS Lite adequately.
MicroSD card (16 GB+)	Class 10 / A1 rated. Use a quality brand - SD card failures are the #1 Pi reliability issue.
Meshtastic USB node	T-Beam, Heltec V3, RAK WisBlock, or any supported device that presents a USB serial port (native CDC-ACM, or via a CP210x/CH340 USB-UART bridge that appears as a ttyUSB device). See meshtastic.org/docs/hardware for supported devices.
USB cable	Data-capable USB-A to USB-C (or micro, depending on node).
Case and power supply	Official Pi PSU or PoE HAT for rooftop deployments.

Software Setup

Step 1 - Flash the OS

Use **Raspberry Pi Imager** to flash **Raspberry Pi OS Lite (64-bit)** to the SD card. In the imager's Advanced Options, pre-configure:

- Hostname (e.g. `mesh-gw-01`)
- Username and password (since Pi OS Bookworm there is no default `pi` user - the first user is created here; note the name you choose, you will need it for the systemd unit below)
- SSH enabled with your public key
- WiFi credentials (or leave blank if using Ethernet)

Step 2 - Install dependencies

```
sudo apt update && sudo apt upgrade -y
sudo apt install -y mosquitto mosquitto-clients python3-pip
pip3 install meshtastic
```

Step 3 - Configure Mosquitto

Edit `/etc/mosquitto/mosquitto.conf` (or create a file in `/etc/mosquitto/conf.d/`). **Mosquitto 2.0+ (the version shipped by apt) defaults `allow_anonymous` to false and, with no listener defined, binds to localhost only** - a bare install rejects LAN clients until you add a listener and either allow anonymous access or a password file:

```
# Allow anonymous local connections (safe for LAN-only installs)
listener 1883
allow_anonymous true

# For remote access, use authentication instead:
# listener 1883 0.0.0.0
# allow_anonymous false
# password_file /etc/mosquitto/passwd
```

```
sudo systemctl enable mosquitto
sudo systemctl start mosquitto
```

Step 4 - Configure the Meshtastic node

Connect to the node via the Meshtastic app or CLI and set:

- **MQTT server / address:** `localhost` (if running on the same Pi) or the Pi's LAN IP from another device.
- **MQTT port:** 1883
- **Uplink enabled:** Yes (per-channel; uplink defaults to off, so enable it on each channel you want to bridge)
- **Downlink enabled:** Yes (to receive messages from MQTT back to the mesh)
- **Encryption:** `mqtt.encryption_enabled` is a separate toggle for whether protobuf payloads are sent encrypted. Set `mqtt.encryption_enabled = false` to send unencrypted protobuf. For plaintext JSON, set `mqtt.json_enabled = true` - JSON packets are always unencrypted, regardless of `encryption_enabled`. (JSON is not supported on nRF52 boards such as RAK WisBlock.)

Via CLI:

```
meshtastic --set mqtt.address localhost --set mqtt.enabled true --set mqtt.json_enabled true
# uplink is per-channel and defaults to off - enable it on the channel you want bridged:
meshtastic --ch-index 0 --ch-set uplink_enabled true
```

Step 5 - Verify packets are flowing

```
mosquitto_sub -h localhost -t 'msh/#' -v
```

With JSON enabled (`mqtt.json_enabled true`), text, position, and telemetry packets appear as readable JSON under `msh/REGION/2/json/...`. Without JSON, the payload under `msh/REGION/2/e/...` is raw protobuf (encrypted or unencrypted) and `mosquitto_sub` displays it as binary, not readable separate messages. If `mosquitto_sub` returns immediately or hangs with nothing at all, confirm `allow_anonymous true` is actually applied - Mosquitto 2.0 refuses anonymous connections by default; add `-u/-P` if you set a password.

Remote Access Options

- **LAN only:** Listen on `127.0.0.1` or LAN IP. Accessible only within your local network - simplest and most secure.
- **Internet-exposed with auth:** Set `listener 1883 0.0.0.0` with a password file. Open port 1883 in your router/firewall only if you need external access. Consider using TLS (Mosquitto supports it natively).
- **Cloud MQTT broker:** Point your nodes at EMQX Cloud, HiveMQ Cloud, or a self-hosted Mosquitto VPS. Multiple Pi gateways in different locations all publish to the same broker - gives you a unified view of all gateways from anywhere.

systemd Service for the Meshtastic Connection

If you run a Python script to bridge or monitor the mesh, create `/etc/systemd/system/mesh-bridge.service`. **Replace** `youruser` **below with the actual username you created when imaging** - since Pi OS Bookworm there is no default `pi` account, and the unit will fail to start if the user does not exist:

```
[Unit]
Description=Meshtastic mesh bridge
After=network.target mosquitto.service

[Service]
User=youruser
ExecStart=/usr/bin/python3 /home/youruser/mesh_bridge.py
Restart=on-failure
RestartSec=10

[Install]
WantedBy=multi-user.target
```

```
sudo systemctl enable mesh-bridge
sudo systemctl start mesh-bridge
```

Node-RED on the Same Pi

Install Node-RED for visual flow-based packet processing with zero additional cloud dependency:

```
bash <(curl -sL https://github.com/node-red/linux-installers/releases/latest/download/update-noc
```

Use the MQTT-in node subscribed to `msh/#` to receive all mesh packets, then add function nodes to parse JSON, filter by type, and push to dashboards, databases, or notification services. The Node-RED UI dashboard module provides a web-accessible map and message log without any external services.

Power Budget

Board	Idle / light-load power	Notes
-------	-------------------------	-------

Raspberry Pi 4 (2 GB)	~3.4 W	Idle/light-load figure, not peak. PoE HAT adds ~1 W; suitable for rooftop enclosure with PoE switch
Raspberry Pi 3B+	~2.9 W	Idle/light-load figure. Good balance of capability and power
Raspberry Pi Zero 2W	~0.9 W	Idle/light-load figure. Best for solar/battery; limited to single USB device, requires USB OTG adapter

Add ~0.5 - 1 W (average) for the connected Meshtastic node; TX bursts at +22 dBm draw more momentarily. The figures above are idle/light-load estimates, not peak. A 12 V/7 Ah SLA battery is ~84 Wh nominal, but SLA chemistry should only be discharged to ~50% depth to avoid damage (~42 Wh usable), and a 12 V->5 V buck converter loses ~10-15%. Against a Zero 2W + node load of roughly 1.5-1.9 W, expect about **20-25 hours** of runtime, not 40+. Reaching 40+ hours would require deep discharge that shortens SLA lifespan.

Setting Up a Meshtastic MQTT-to-Internet Gateway

An MQTT gateway connects your Meshtastic mesh to the internet, enabling message delivery to non-LoRa clients, integration with home automation, and connection to the global Meshtastic MQTT network.

What the MQTT Gateway Does

A Meshtastic node in "MQTT gateway" mode:

- Receives LoRa packets on channels that have MQTT uplink enabled
- Forwards them to an MQTT broker (local or cloud)
- Receives messages from the MQTT broker and injects them into the LoRa network (on channels with downlink enabled)
- Bridges your mesh to the global Meshtastic network (if using the public MQTT server)

Note: uplink and downlink are configured **per channel** and both default to **OFF**. MQTT only carries traffic for a channel once the MQTT module is enabled *and* that channel has uplink (and/or downlink) turned on.

Requirements

- A WiFi-capable Meshtastic node (T-Beam, Heltec V3, T-Beam Supreme - all have WiFi)
- WiFi network with internet access at the gateway location
- An MQTT broker: either the public Meshtastic broker (mqtt.meshtastic.org) or a self-hosted one

Configuration Steps

Option A: Using the Meshtastic Public MQTT Server

The public broker `mqtt.meshtastic.org` is **not anonymous** - it requires username `meshdev` / password `large4cats` (these are also the firmware's built-in defaults).

```
# Configure via CLI:
meshtastic --set mqtt.enabled true
meshtastic --set mqtt.address mqtt.meshtastic.org
meshtastic --set mqtt.username meshdev
meshtastic --set mqtt.password large4cats

# Configure which channels to bridge (uplink/downlink are per-channel and default OFF)
meshtastic --ch-index 0 --ch-set uplink_enabled true
meshtastic --ch-index 0 --ch-set downlink_enabled true

# Configure WiFi (if not already done)
meshtastic --set network.wifi_ssid "YourSSID"
meshtastic --set network.wifi_psk "YourPassword"
```

Privacy note: The public MQTT server relays messages globally. Only use it for the default (unencrypted) channel unless you want your encrypted channel traffic relayed globally.

Option B: Self-Hosted Mosquitto MQTT Broker

```
# Install Mosquitto on Raspberry Pi or VPS:
sudo apt install mosquitto mosquitto-clients
sudo systemctl enable --now mosquitto

# Mosquitto 2.0+ defaults allow_anonymous to FALSE and binds localhost-only.
# A bare install rejects LAN clients until you define a listener. Create
# /etc/mosquitto/conf.d/local.conf with:
#   listener 1883 0.0.0.0
#   allow_anonymous true   # trusted LAN only; otherwise use a password_file
# For authentication instead of anonymous access:
#   mosquitto_passwd -c /etc/mosquitto/passwd youruser
```

```
# password_file /etc/mosquitto/passwd
# allow_anonymous false

sudo systemctl restart mosquitto

# Configure node to use your local broker:
meshtastic --set mqtt.address 192.168.1.100
meshtastic --set mqtt.enabled true
```

Verifying the Gateway is Working

```
# Subscribe to all Meshtastic MQTT topics and watch for packets
# (the public broker requires credentials):
mosquitto_sub -h mqtt.meshtastic.org -u meshdev -P large4cats -t "msh/US/2/e/#" -v

# Or on your local broker:
mosquitto_sub -h 192.168.1.100 -t "msh/#" -v
```

You should see packets appearing as mesh traffic is received. On the `/e/` (encrypted) topic these are raw binary protobuf-encoded ServiceEnvelope messages, not base64 text.

MQTT Packet Decoding

Meshtastic MQTT packets on the `/e/` topic are protobuf-encoded. The Python `meshtastic` package has **no turnkey MQTT class** - the documented approach is to subscribe with **paho-mqtt** and decode the payload with the bundled protobuf bindings (`meshtastic.protobuf`):

```
pip install meshtastic paho-mqtt

import paho.mqtt.client as mqtt
from meshtastic.protobuf import mqtt_pb2 # ServiceEnvelope lives in mqtt.proto

def on_message(client, userdata, msg):
    # The /e/ payload is RAW BINARY protobuf - feed it straight in, do not base64-decode it
    envelope = mqtt_pb2.ServiceEnvelope()
    envelope.ParseFromString(msg.payload)
    print(envelope)

client = mqtt.Client()
```

```
client.username_pw_set("meshdev", "large4cats")
client.on_message = on_message
client.connect("mqtt.meshtastic.org", 1883)
client.subscribe("msh/US/2/e/#")
client.loop_forever()
```

Note: `from meshtastic.mqtt import MQTT` does not exist - there is no built-in MQTT client class in the library.

Room Server Administration