

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](#)

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 `rx_rssi`, `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/YYY';
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.