

Advanced Features

- [Meshtastic Web Interface and Remote Access](#)
- [Meshtastic Module Configuration Reference](#)
- [Traceroute and Path Diagnostics](#)
- [Neighbor Info and Signal Mapping](#)

Meshtastic Web Interface and Remote Access

Many Meshtastic users don't realize a full web-based interface is available in addition to the mobile app. The web interface provides additional tools for configuration and administration that aren't available in the app.

The Meshtastic web app (client.meshtastic.org)

The official Meshtastic web application is hosted at client.meshtastic.org and runs in any modern browser (Chrome or Edge recommended for WebSerial support), connecting to your device via USB serial or Bluetooth.

Connecting via USB serial

1. Connect your Meshtastic device via USB cable
2. Open client.meshtastic.org in Chrome or Edge
3. Click "Connect" and select "Serial"
4. Choose your device from the port list
5. The device configuration loads automatically

Note: WebSerial requires Chrome or Edge. Firefox and Safari do not support WebSerial. If the device doesn't appear in the port list, install the appropriate driver (CH340 for Heltec, standard CDC for nRF52-based devices).

Connecting via Bluetooth

1. Open client.meshtastic.org in a Chromium-based browser
2. Click "Connect" and select "Bluetooth"
3. Pair with your device - the device should be in BLE advertising mode (typically shown by LED blink pattern)

What the web app offers beyond the mobile app

Feature	Mobile app	Web app
Node configuration	Yes	Yes (more detailed)
Channel management	Yes	Yes
Message view	Yes	Yes
Node map	Yes	Yes
Configuration export/import (YAML)	Channel/config sharing via URL/QR; recent apps also support config backup	Yes
Raw packet log	Limited	Full packet log
Serial debug console	No	Yes
Firmware update (OTA)	Yes (BLE OTA on supported devices)	Browser flashing via USB (Web Flasher); verify current per-platform support
Full module config access	Partial	Yes

Configuration backup and restore

The web app's configuration export feature is an easy way to back up and restore a Meshtastic node's settings (the exported file is YAML; the CLI uses the same format):

1. Connect to device via web app
2. Navigate to **Config → Export Config**
3. Save the YAML file to your computer

To restore: connect the device (after a reflash or factory reset), navigate to **Config → Import Config**, and upload the saved YAML file. This restores all radio settings, channel configurations, and module settings in one step. The equivalent CLI workflow is `meshtastic --export-config > config.yaml` to back up and `meshtastic --configure config.yaml` to restore (there is no `--import-config` flag).

Remote node management via WiFi

ESP32-based Meshtastic nodes (Heltec V3/V4, T-Beam, etc.) can be connected to a local WiFi network so they can be reached over the network:

```
meshtastic --set network.wifi_ssid "YourNetworkSSID"
meshtastic --set network.wifi_psk "YourPassword"
meshtastic --set network.wifi_enabled true
```

Once connected to WiFi, the device's local IP address is shown in the app or via `meshtastic --info`. That IP is used by the Meshtastic web client and API (TCP port 4403) rather than a self-hosted admin webpage; connect to it from client.meshtastic.org (choose the network/TCP connection option) to configure and monitor the node without USB or Bluetooth.

Use case: A deployed rooftop repeater with WiFi access can be configured and monitored remotely from any device on the same network, eliminating the need for physical access for routine configuration changes.

Security warning: The node's network interface has no built-in authentication. Anyone who can reach the node's IP on the network can connect to it, reconfigure it, change channels, or read its configuration (including channel PSKs via export). Only expose a node on a trusted, isolated network - never on open or shared WiFi. For untrusted environments, enable managed mode (`security.is_managed`) and use PKC admin keys for remote administration.

Serial debug console

The web app's serial console shows raw debug output from the device firmware. This is invaluable for diagnosing unusual behavior:

- View all received packets with raw headers and signal data
- See routing decisions in real time
- Monitor power-related messages (battery voltage, charge state)
- Debug GPS acquisition

The serial console is accessed via **Tools → Serial Console** in the web app. Output can be copied and shared when reporting bugs.

Meshtastic Module Configuration Reference

Meshtastic's module configuration system enables advanced features beyond basic mesh communication. This page covers the most commonly needed modules and their key settings.

Range Test Module

Automates systematic coverage testing by sending timed packets that include position data.

```
# Enable and configure via CLI
meshtastic --set range_test.enabled true
meshtastic --set range_test.sender 60 # Send every 60 seconds
meshtastic --set range_test.save true # Save results to rangetest.csv on the device's internal flash (ESP32 only)
```

The receiving node logs packets with GPS coordinates and signal data to its internal flash as `rangetest.csv` (ESP32 boards only; most boards have no SD card, so `range_test.save` has no effect on them). Download the file from the node's web interface at `meshtastic.local/rangetest.csv`, then import it into Google My Maps, uMap (OpenStreetMap), QGIS, or Google Earth to visualize coverage as a heatmap. **Note:** a range-test sender continuously floods airtime, so run it only briefly and on an isolated test channel to avoid disrupting your live mesh.

Telemetry Module

Controls how often nodes broadcast battery, environment, and air quality data.

```
meshtastic --set telemetry.device_update_interval 1800 # Battery/voltage every 30 min
meshtastic --set telemetry.environment_update_interval 900 # Temperature/humidity every 15 min
meshtastic --set telemetry.air_quality_enabled true # Enable if a particulate sensor (e.g. PMSA003I) is attached
```

Telemetry data is visible in the app's node detail view. Excessive telemetry frequency contributes to network congestion in dense deployments - don't set intervals below 5 minutes for infrastructure nodes. **Note:** the Air Quality module requires a supported particulate sensor (e.g. PMSA003I). A BME680/BME68x reports gas resistance under Environment Telemetry, not the Air Quality module.

Position Module

Controls how and when GPS position is broadcast.

```
meshtastic --set position.position_broadcast_secs 1800 # Broadcast every 30 min
meshtastic --set position.broadcast_smart_minimum_distance 100 # Smart broadcast if moved 100m
meshtastic --set position.gps_update_interval 120 # Update GPS position every 2 min
meshtastic --set position.gps_mode ENABLED # DISABLED to save power on static nodes
```

For static repeaters: Disable GPS or set a fixed position instead of active GPS polling to save power:

```
meshtastic --set position.gps_mode DISABLED
meshtastic --setlat 47.6062 --setlon -122.3321 --setalt 150
```

External Notification Module

Triggers a buzzer, LED, or vibration motor on message receipt. Useful for personal portable nodes where you can't always watch the screen.

```
meshtastic --set external_notification.enabled true
meshtastic --set external_notification.alert_message true
meshtastic --set external_notification.alert_bell true # Triggers buzzer
meshtastic --set external_notification.output_ms 1000 # Buzzer on for 1 second
```

Store and Forward Module

Enables a node to store messages and deliver them to nodes that were offline when the message was sent. Requires an ESP32 device with onboard PSRAM (e.g. T-Beam v1.0+, T3S3). Records are held in volatile PSRAM (lost on reboot or power loss), not flash. nRF52 boards (RAK4631, T-Echo) cannot act as a Store and Forward server at all.

```
meshtastic --set store_forward.enabled true
meshtastic --set store_forward.records 0 # Max records to store; default 0 auto-sizes to ~2/3
of PSRAM (~11,000 records). 100 here is just an example
meshtastic --set store_forward.history_return_max 25 # Return up to 25 historical messages on
request
```

Note: [Store and Forward](#) is resource-intensive and delivery is best-effort and request-driven, not guaranteed. Enable it only on an ESP32 device with onboard PSRAM; nRF52 devices cannot run a Store and Forward server. This feature provides some of the functionality that MeshCore's room servers offer natively.

Serial Module

Bridges mesh messages to/from a connected serial device (microcontroller, computer, sensors). Enables custom IoT integrations without modifying firmware.

```
meshtastic --set serial.enabled true
meshtastic --set serial.baud BAUD_115200
meshtastic --set serial.timeout 0 # No timeout
meshtastic --set serial.mode TEXTMSG # Treat serial data as text messages
```

Use case: Connect an Arduino to a Meshtastic node via serial. The Arduino reads sensor data and sends it as a serial string, which the Meshtastic node broadcasts as a mesh message.

Canned Messages Module

Pre-loads a set of frequently used messages that can be selected and sent without typing. Especially useful on devices without keyboards (T-Echo, T1000-E).

```
meshtastic --set canned_message.enabled true
meshtastic --set-canned-message "OK, copy|Need assistance|ETA 15 minutes|At base|En route"
```

Messages are separated by pipe characters. On devices with buttons, scroll through the list and press to send. On the T-Echo, use the rotary encoder to navigate and click to send.

Module configuration best practices

- **Don't enable everything:** Each active module adds to channel traffic. Enable only what you actually use.
- **Set conservative intervals:** Telemetry and position intervals of 15 - 30 minutes are appropriate for most deployments. Shorter intervals increase traffic without proportional benefit.
- **Test after configuration:** After enabling any module, verify it behaves as expected before relying on it operationally.
- **Back up your config:** After a working configuration, export it via the web app so you can restore it after a firmware update.

Traceroute and Path Diagnostics

What Is Meshtastic Traceroute?

Traceroute is a diagnostic feature built into Meshtastic firmware that lets you discover the path a packet takes through the mesh to reach a destination node. Unlike a simple ping, traceroute collects the ID of each intermediate node that relayed the packet, along with signal quality data for each hop. Treat the result as one probe's path, not a definitive map: like normal traffic, a traceroute probe's route can differ from where your real messages flow, so it shows how this particular packet was routed rather than a guaranteed picture of all traffic.

How It Works

When you initiate a traceroute, your node sends a special packet addressed to the destination node ID. Each relaying node that knows the channel's encryption key appends its own node ID and the SNR at which it received the packet to the route record (this is not limited to nodes configured with a ROUTER role - any rebroadcasting node that shares the channel key is recorded). From firmware version ≥ 2.5 , the route back to the origin is also recorded, with the SNR for each link. When the destination receives the packet, it sends the complete hop list back to you. The result is an ordered list: source $\rightarrow$ relay 1 $\rightarrow$ relay 2 $\rightarrow$... $\rightarrow$ destination.

Traceroute uses Meshtastic's managed flood-routing mechanism (see the mesh-algorithm docs), the same mechanism as normal traffic, but the route-record payload (the RouteDiscovery protobuf) grows with each hop. This means traceroute is more bandwidth-intensive than a regular message and should be used sparingly on busy networks.

Running Traceroute from the App

In the Meshtastic Android or iOS app:

1. Open the **Node List** and tap the target node.
2. Tap the three-dot menu or the node detail view.

3. Select **Traceroute**.
4. Wait for the result - it may take 10 - 30 seconds depending on hop count and channel utilization.

The result displays each hop's node ID and the SNR value observed at that hop.

Running Traceroute from the CLI

```
meshtastic --traceroute '!a1b2c3d4'
```

Replace `!a1b2c3d4` with the destination's node ID (the exclamation-mark prefix is required).

Single-quote the node ID as shown - in bash the bare `!` triggers history expansion and you will get an "event not found" error otherwise. The CLI prints each hop as it is received:

```
Traceroute to !a1b2c3d4:
You (!ff00ee11) --> !11223344 (SNR: 8.25) --> !55667788 (SNR: 4.50) --> !a1b2c3d4
(destination)
```

Interpreting the Output

- **Hop count:** fewer hops generally means lower latency and higher reliability.
- **SNR values:** higher is better, but there is no single fixed "weak link" cutoff - the usable SNR floor is **spreading-factor-dependent**. LoRa can demodulate well below 0 dB: roughly -7.5 dB at SF7 down to about -20 dB at SF12. On the default LongFast preset (SF11) the decode floor is around -17 dB, so a hop at -10 dB on LongFast is still usable with margin. A link is weak only when its SNR is within a few dB of the preset's floor (on LongFast, worse than roughly -14 dB is marginal). See a Semtech LoRa SNR demodulation table for the per-SF limits.
- **Missing hops:** if the traceroute times out without a complete result, the destination may be unreachable or the return path failed.

Using Traceroute to Diagnose Coverage Gaps

Run traceroute from multiple points in your network to map which nodes are serving as critical relays. If a single node appears in most traceroute paths, it is a single point of failure - consider adding a redundant node nearby. An unexpectedly long hop count (e.g. 5 hops when you expect 2) suggests a shorter path exists but is not being used, possibly due to SNR thresholds or a misconfigured router role.

Comparing Two Runs to Detect Path Changes

Run traceroute to the same destination before and after a node failure or configuration change. A change in the hop list confirms that traffic is now routing differently. This is especially useful after repositioning a relay node or changing antenna - it lets you verify the improvement objectively rather than relying on signal reports alone. Bear in mind each run reflects one probe's path, so compare several runs rather than reading a single result as definitive.

Neighbor Info and Signal Mapping

What Is the Neighbor Info Module?

The Neighbor Info module is a built-in Meshtastic feature that periodically broadcasts a summary of every node your device can hear directly, along with the signal-to-noise ratio (SNR) of each link. This data lets you build an objective picture of your mesh's link quality without relying on anecdotal reports.

What Neighbor Info Reports

Each Neighbor Info packet contains:

- **Node ID** of each directly-heard neighbor
- **SNR** (signal-to-noise ratio in dB) - how cleanly the signal was received
- The **timestamp** of the last reception from that neighbor

Note: Neighbor Info reports **SNR only** per neighbor - the NeighborInfo payload does not carry a per-neighbor RSSI value. RSSI is available locally for each packet your own node receives, but it is not part of the broadcast neighbor report.

Reports are sent at a configurable interval (default 21600 seconds / 6 hours). The firmware enforces a minimum of 14400 seconds / 4 hours - values below this are rejected. The data is broadcast on the mesh and visible to any node that receives it.

Privacy note: enabling Neighbor Info publicly discloses which nodes your node can hear - an RF "social graph" of who-can-hear-whom. On any channel that is uplinked to MQTT, that graph is published to the internet (and can appear on third-party maps). For sensitive or EmComm deployments, consider leaving Neighbor Info disabled on nodes whose neighbor relationships are operationally sensitive.

Enabling Neighbor Info

1. Open the [Meshtastic app](#) and connect to your node.
2. Go to **Radio Configuration → Modules → Neighbor Info**.
3. Toggle **Enabled** to on.
4. Optionally adjust the **Update Interval** (in seconds; minimum 14400 / 4 hours). Shorter intervals give fresher data but increase channel utilization.
5. Save and reboot the node.

Via CLI: `meshtastic --set neighbor_info.enabled true --set neighbor_info.update_interval 21600` (any value ≥ 14400 ; lower values are rejected by firmware).

Reading the Data

In the Meshtastic app, go to **Node List → [select a node] → Neighbor Info** to see that node's reported neighbors. In the Python CLI, listen for neighbor info packets:

```
meshtastic --listen
# Look for portnum: NEIGHBORINFO_APP packets in the JSON output
```

The JSON payload includes a `neighbors` array, with each neighbor carrying its node ID and `snr`. (Confirm exact serialized field names against a live NEIGHBORINFO_APP JSON sample or the `neighbor_info.proto` before relying on them in a parser.)

Building a Link Quality Map

Collect neighbor info data from all nodes over a period of time to build a directional link quality graph. Each directed edge in the graph represents one node hearing another, with the SNR as the edge weight. Nodes with low average SNR to all their neighbors are candidates for antenna upgrades or repositioning.

Tools like **Meshview** or a custom Python script consuming the MQTT JSON feed can automatically visualize this as a network graph, coloring links by quality. Useful SNR thresholds are **preset-dependent**, because LoRa's decode floor varies with spreading factor (roughly -7.5 dB at SF7 up to about -20 dB at SF12; $\sim$ -17 dB on the default LongFast/SF11 preset). As a suggested visualization convention for the default LongFast preset (not an official spec): green > 0 dB, yellow -10 to 0 dB, and red only when within a few dB of the $\sim$ -17 dB decode floor.

Exporting to CSV for Analysis

Using a simple Python MQTT subscriber. Note that JSON packets are published to `msh/REGION/2/json/CHANNELNAME/USERID` - segmented by channel name and gateway user ID, not by packet type. Subscribe to the JSON tree (or a specific channel) and filter on the JSON `type` field instead of using a type-named topic:

```
import paho.mqtt.client as mqtt, json, csv, time

rows = []

def on_message(client, userdata, msg):
    d = json.loads(msg.payload)
    if d.get("type") == "neighborinfo":
        for n in d["payload"].get("neighbors", []):
            rows.append({
                "reporter": hex(d["from"]),
                "neighbor": hex(n["node_id"]),
                "snr": n["snr"],
                "time": time.strftime("%Y-%m-%d %H:%M:%S")
            })

client = mqtt.Client()
client.on_message = on_message
client.connect("localhost", 1883)
client.subscribe("msh/US/2/json/#") # all channels; or msh/US/2/json/LongFast/# for one
client.loop_forever()
```

Write `rows` to a CSV and import into a spreadsheet to sort and filter weak links.

Identifying Weak Backbone Links

Sort the exported CSV by SNR ascending. Treat a link as fragile when its SNR is within a few dB of the decode floor for your active modem preset - not at a fixed cutoff. On the default LongFast preset (SF11, floor ~ -17 dB) a -5 dB link still has roughly 12 dB of margin and is healthy; the fragile links there are the ones worse than about -14 dB. A -5 dB cutoff is only appropriate near SF7 (floor ~ -7.5 dB). For genuinely weak links, consider: raising antenna height, switching to a high-gain directional antenna, or adding an intermediate relay node to split the hop into two shorter, stronger links.