Spectra User Manual

Distributed Spectrum Intelligence Platform

Simon Knight

Adelaide, Australia

March 2026

> **About this manual.** This manual covers setting up SDR hardware, running autonomous monitoring missions, signal classification, protocol decoding, emitter fingerprinting, and natural-language querying with the Spectra distributed spectrum intelligence platform. It is intended for RF engineers and spectrum operators deploying Spectra in both laboratory and field environments.

Contents

1 Quick Start

Get Spectra monitoring a frequency range in under ten minutes.

Step 1. **Connect your SDR hardware.** Plug an RTL-SDR, AirSpy R2/Mini, or Kraken SDR into a USB 3.0 port. Confirm the device is visible:

```
rtl_test -t
# or for AirSpy:
airspy_info
```

Step 2. **Start the SDR hardware server.** Launch `rtltcp-rust` on the machine attached to the SDR. The server abstracts the hardware behind a unified TCP and REST interface:

```
rtltcp-rust --device 0 --sample-rate 2400000 --bind 0.0.0.0:1234
```

The server prints its REST base URL (`http://0.0.0.0:8080` by default) once initialisation is complete.

Step 3. **Start the Spectra backend.** In a separate terminal, launch the FastAPI orchestrator and point it at the running SDR server:

```
cd spectra/
uvicorn spectra.main:app --host 0.0.0.0 --port 8000 \
    --env-file config/spectra.env
```

Step 4. **Launch the terminal UI.** Open a third terminal and start `spectra-tui` to see the live spectrum waterfall:

```
spectra-tui --api http://localhost:8000
```

Press **W** to open the waterfall view. Signal detections appear as coloured blobs against the scrolling spectrum background.

Step 5. **Start a basic scan mission.** From the TUI main menu press **M**, then **N** (new mission). Choose preset **QUICK_SCAN** to sweep 88–108 MHz FM with default detection thresholds. Press **Enter** to arm and **S** to start.

Within a few seconds detected signals appear in the *Census* pane with modulation labels and confidence scores.

2 Installation and Prerequisites

2.1 SDR Hardware

Spectra supports three families of SDR hardware.

▷ **Supported SDR Hardware**

- **RTL-SDR** (RTL2832U chipset) — 500 kHz–1.75 GHz, 2.56 MS/s max. Lowest cost option; adequate for HF–UHF monitoring.
- **AirSpy R2 / Mini** — 24–1750 MHz, 10 MS/s (R2) or 6 MS/s (Mini). Higher dynamic range and sample rate than RTL-SDR.

- **Kraken SDR** — Five-channel coherent RTL-SDR array for direction-finding and passive radar. Requires the Passive Radar integration (see Section 8).

2.2 Software Dependencies

▷ **Required Software**

- **Python 3.10+** with `pip` or `uv`
- **Rust toolchain** (stable, 1.75+) — for `rtltcp-rust` and `spectra-tui`
- **FastAPI / Uvicorn** — `pip install fastapi uvicorn`
- **PyTorch 2.x** (CPU or CUDA) *or* **CoreML Tools** (macOS) — for ML inference
- **DuckDB** — persistent signal census database
- **ChromaDB** — vector database for emitter fingerprinting
- **Ollama** — local LLM serving for natural-language queries
- **multimon-ng** — POCSAG, DTMF, FLEX, and EAS protocol decoding
- **DSDcc** — P25, DMR, D-STAR, NXDN digital voice decoding
- **librtlsdr** and/or **libairspy** — native SDR drivers

2.3 Installing Spectra

```
git clone https://github.com/simonknight/spectra.git
cd spectra
pip install -e ".[ml,audio]"
cargo build --release --manifest-path rtltcp-rust/Cargo.toml
cargo build --release --manifest-path spectra-tui/Cargo.toml
```

```
sudo apt-get install multimon-ng
# Build DSDcc from source:
git clone https://github.com/f4exb/dsdcc.git && cd dsdcc
cmake -DBUILD_TYPE=Release . && make -j$(nproc) && sudo make install
```

Apple Silicon: mlx-whisper

On Apple Silicon (M1/M2/M3/M4) Macs, install `mlx-whisper` instead of the standard `openai-whisper` package. The MLX backend achieves 3–8× faster transcription by exploiting the unified memory architecture:

```
pip install mlx-whisper
```

Set `SPECTRA_WHISPER_BACKEND=mlx` in your environment file. The CoreML accelerated model is automatically selected.

Single-SDR Steering Limitation

Each physical SDR device can only tune to one centre frequency at a time. When Spectra steers the receiver to decode a detected signal, ongoing mission scanning is paused for the duration of the decode window (typically 200–2000 ms depending on capture level). Operators monitoring dense frequency ranges with a single RTL-SDR should use `mission.pipeline_config.capture_level = BASIC` to minimise steering interruptions. A multi-SDR deployment (see Section 6) eliminates this constraint.

2.4 Configuration File

Copy the template and adjust paths and device parameters before first launch:

```
cp config/spectra.env.template config/spectra.env
```

```
$EDITOR config/spectra.env
```

Key settings in `spectra.env`:

```
RTLTCP_HOST=127.0.0.1
RTLTCP_PORT=1234
SPECTRA_DB_PATH=/var/lib/spectra/census.duckdb
CHROMA_PERSIST_DIR=/var/lib/spectra/chroma
OLLAMA_BASE_URL=http://localhost:11434
OLLAMA_MODEL=llama3.1:8b
SPECTRA_WHISPER_BACKEND=pytorch    # or mlx on Apple Silicon
SPECTRA_LOG_LEVEL=INFO
```

2.5 Hardware Optimization (SIMD/vDSP)

Spectra is designed for high-throughput real-time DSP. On Apple Silicon (M1/M2/M3), it leverages the **Accelerate framework (vDSP)** for hardware-accelerated FFTs and FIR filtering.

- **Automatic Detection**: Spectra automatically detects vDSP availability at startup. On macOS, ensure `ctypes` can load `Accelerate.framework`.
- **Manual Override**: To disable hardware acceleration and force NumPy fallback (for debugging), set:

  ```
  export SPECTRA_DISABLE_VDSP=1
  ```

- **Performance**: vDSP acceleration reduces CPU usage by 40–60% during wideband monitoring compared to standard NumPy FFTs.

3 Core Workflows

3.1 Real-Time Monitoring

Real-time monitoring provides a continuous spectrum waterfall with live signal annotation. No mission is required; the backend scans continuously using the default detection parameters.

Step 1. Ensure `rtltcp-rust` and the Spectra backend are running (see Section 1).

Step 2. Launch `spectra-tui` and open the *Waterfall* view with **W**. The waterfall scrolls at approximately 10 frames per second; brighter colours indicate higher power.

```
W          Open waterfall
+/-        Adjust colour scale (dB range)
[ / ]      Shift centre frequency down / up by 100 kHz
F          Enter frequency directly (MHz)
G          Jump to signal under cursor in Census
Q / Esc    Return to main menu
```

Step 3. Detected signals appear as coloured overlays. The label bar at the bottom shows the ML modulation class and confidence for the most recently decoded signal.

Step 4. To increase scan bandwidth, adjust the SDR sample rate in `rtltcp-rust`. A 2.4 MS/s sample rate yields approximately 2 MHz of visible spectrum; an AirSpy at 10 MS/s covers 10 MHz.

> **Gain Setting**
>
> Over-driven input degrades classification accuracy. Start with `rtltcp-rust --gain auto` and inspect the waterfall for noise floor quality. If signals are saturating (flat-topped), reduce gain incrementally using `--gain-db 28`.

3.2 Autonomous Missions

Missions let Spectra operate unattended, sweeping a defined frequency list, applying configurable capture levels, and stopping when a trigger condition is met.

Step 1. Create a `MissionDef` configuration file in YAML:

Listing 1: Example mission: VHF Public Safety scan

```yaml
name: vhf_public_safety
description: "Monitor VHF public safety bands 138-174 MHz"
frequencies_hz:
  - 155_340_000    # Fire dispatch
  - 156_800_000    # Marine channel 16
  - 162_400_000    # NOAA WX 1
  - 163_275_000    # Local EMS
dwell_time_s: 0.5
pipeline_config:
  capture_level: DEEP
  classifier_confidence_gate: 0.65
  enable_protocol_decode: true
  enable_sei: true
stop_conditions:
  max_duration_s: 3600
  max_signal_hits: 10000
detection:
  threshold_db: -65.0
  merge_gap_hz: 5000
  min_bandwidth_hz: 1000
```

Step 2. Load and arm the mission via the REST API or TUI:

```
curl -s -X POST http://localhost:8000/api/missions/load \
    -H "Content-Type: application/json" \
    -d @vhf_public_safety.yaml | jq .
```

Step 3. Start the mission:

*TUI shortcut: **M** then **S** to start the armed mission.*

```
curl -s -X POST http://localhost:8000/api/missions/start | jq .
```

Step 4. Monitor mission progress in the TUI *Mission* pane (**M** key) or via the status endpoint:

```
curl -s http://localhost:8000/api/missions/current | jq .status
```

Step 5. When the stop condition triggers, the mission transitions to `COMPLETE`. Request a briefing (see Section 3.9) to summarise detections.

3.3 Satellite Intelligence

Spectra includes a TLE-based satellite scheduler for autonomous acquisition of Low Earth Orbit (LEO) satellites (NOAA Weather, Orbcomm, CubeSats).

Step 1. **Configure your ground station.** Edit `configs/satellites.json` to set your geographical coordinates:

```json
{
  "station": {
    "lat_deg": -34.9,
    "lon_deg": 138.6,
    "elevation_m": 50.0
```

```
  },
  "satellites": [
    {
      "name": "NOAA 19",
      "center_freq_hz": 137100000,
      "sample_rate_hz": 1000000,
      "tle_name": "NOAA 19"
    }
  ]
}
```

Step 2. **Fetch TLE data.** Spectra automatically fetches current Two-Line Elements from CelesTrak. If internet access is restricted, provide a local `tle.txt` in the config directory.

Step 3. **Monitor upcoming passes.** Use the TUI *Satellites* pane (**S** key) or the REST API:

```
curl -s http://localhost:8000/api/missions/satellites/passes | jq .
```

Step 4. **Arm the scheduler.** When a pass begins (AOS), the `MissionEngine` automatically tunes the radio, applies **Doppler correction**, and starts a `DEEP` capture.

3.4 Autopilot Sweeps

The *Autopilot* mode explores the RF spectrum autonomously by prioritising frequency bands based on historical activity and novelty.

- **Interestingness Scoring**: Autopilot ranks bands using three factors: **Novelty** (fewer recent hits), **Activity** (high short-term burst rate), and **Anomaly** (deviation from baseline bandwidth/labels).
- **Bandplan**: Edit `configs/bandplan.default.json` to define segments for the Autopilot to explore.
- **Activation**: Toggle Autopilot via the TUI (**A** key) or the `/api/missions/autopilot/start` endpoint.

3.5 Signal Classification

Spectra uses a two-stage classification pipeline that balances throughput against accuracy.

3.5.1 Stage 1: Fast Classifier

The fast classifier runs on every detected signal using a lightweight convolutional model (RadioML 2018.01a architecture, 11 classes, 300 ms inference budget on CPU). It operates on a 128-sample IQ snapshot captured at the moment of detection.

Step 1. The energy detector raises a `DetectedSignal` event whenever a frequency bin exceeds the configured threshold.

Step 2. The orchestrator schedules an IQ capture: the SDR is steered to the signal's centre frequency and a short burst is recorded.

Step 3. The fast classifier scores the snapshot and attaches a `SignalClassification` with `is_retrospective=False`.

Step 4. If the confidence score exceeds the configured gate (`classifier_confidence_gate`, default 0.65), the classification is persisted directly to DuckDB.

Step 5. If confidence is below the gate, the signal is queued for Stage 2.

3.5.2 Stage 2: Retrospective Classifier

Step 1. The retrospective classifier processes queued signals using a larger, multi-scale model with a 2048-sample window. It has a higher accuracy but runs asynchronously with up to a 5 second lag.

Step 2. Classifications from Stage 2 are marked `is_retrospective=True` and may override Stage 1 results in the DuckDB census.

Step 3. SigIDWiki enrichment is applied to classifications above the gate: the label is matched against the SigIDWiki database to attach human-readable descriptions, typical bandwidth ranges, and known emitter categories.

> **Expanding Classifier Coverage**
>
> The default classifier covers 11 modulation classes (AM, FM, USB, LSB, CW, BPSK, QPSK, 8PSK, QAM16, QAM64, WBFM). To add classes, retrain using the Spectra synthetic training lab and drop the new `.torchscript` model into `models/`. See *SPECTRA-TR-001* Section 7 for the training pipeline.

3.6 Protocol Decoding

Protocol decoding is handled by external subprocess decoders: `multimon-ng` for common digital modes and `DSDcc` for digital voice.

Step 1. When a signal classification reaches a protocol-decodable class (e.g. POCSAG, DTMF, P25, DMR), the orchestrator launches a subprocess decoder with a short recorded IQ burst as input.

Step 2. The decoder's stdout is captured and parsed. Decoded text, talkgroup IDs, and unit IDs are attached to the signal hit record in DuckDB.

Step 3. For long-lived emissions (push-to-talk audio), the decode window extends until squelch close or a configurable maximum duration.

Step 4. Decoded text appears in the TUI *Decode* pane (**D** key) and is included in the `/api/census/deep` response.

```
multimon-ng -h | head -5
```

> **Subprocess Decoder Crashes**
>
> If a decoder subprocess exits with a non-zero code, Spectra logs a warning and continues scanning; the affected signal hit is marked `decode_status=FAILED`. The most common cause is a missing shared library (`libsndfile`). Inspect logs with:
>
> ```
> journalctl -u spectra --since "5 minutes ago" | grep DECODER
> ```
>
> Reinstall `multimon-ng` from source if library errors persist.

3.7 Emitter Fingerprinting (SEI)

Specific Emitter Identification (SEI) extracts transmitter-unique features from IQ waveforms and stores them as vector embeddings in ChromaDB.

Step 1. Enable SEI in the mission pipeline config: `enable_sei: true` (requires `capture_level: DEEP` or `FULL`).

Step 2. On each qualifying signal, the SEI pipeline extracts a 64-dimensional feature vector from phase noise, spectral asymmetry, carrier offset drift, and transient rise-time characteristics.

Step 3. The vector is compared against the ChromaDB collection using cosine similarity. If the nearest-neighbour distance is below the configured threshold (`sei_match_threshold`, default 0.15), the signal is attributed to an existing emitter record.

Step 4. If no match is found, a new emitter record is created with a unique `emitter_id` UUID. The emitter entry stores the feature centroid and first/last seen timestamps.

Step 5. Emitter records are available at `/api/emitters` and are cross-referenced in every `signal_hits` row via the `emitter_id` foreign key.

```
curl -s http://localhost:8000/api/emitters \
    | jq '.[] | {id, label, sighting_count, last_seen}'
```

3.8 Audio Intelligence

For analogue and digital voice signals, Spectra can transcribe audio using `mlx-whisper` (Apple Silicon) or `openai-whisper` (all platforms) and detect voice activity.

Step 1. Ensure the Whisper backend is installed and configured (see Section 2.2). Set `SPECTRA_WHISPER_BACKEND` appropriately.

Step 2. Audio transcription activates automatically when the classifier labels a signal as AM, USB, LSB, WBFM, P25, or DMR voice and the capture level is `DEEP` or `FULL`.

Step 3. The SDR steers to the signal and records PCM audio at 48 kHz for the duration of the voice activity detection (VAD) window.

Step 4. The Whisper model transcribes the audio. Transcripts are stored in the `decode_text` column of `signal_hits` and are included in NL query context (see Section 3.9).

Step 5. VAD events (speech start/stop boundaries) are stored as metadata and can be exported via `/api/census/deep`.

> **Noisy RF Audio**
>
> Spectrally noisy recordings reduce Whisper accuracy significantly. Apply squelch filtering before transcription by setting `pipeline_config.audio_squelch_db` to a value 10–15 dB above the measured noise floor. The default is `-50.0` dB.

3.9 Natural-Language Querying

The NLQ subsystem provides a conversational interface to the signal census, using a RAG (Retrieval-Augmented Generation) pipeline backed by DuckDB and Ollama.

Step 1. Ensure Ollama is running and the configured model is pulled:

```
ollama pull llama3.1:8b
ollama serve   # if not already running as a service
```

Step 2. Submit a natural-language question via the API:

```
curl -s -X POST http://localhost:8000/api/nlq \
    -H "Content-Type: application/json" \
    -d '{"query": "How many POCSAG signals were decoded in the last hour?"}' \
    | jq .answer
```

Step 3. The RAG pipeline retrieves relevant census rows from DuckDB and constructs a context-augmented prompt. The LLM generates a natural-language response grounded in the retrieved data.

Step 4. For intelligence briefings that summarise an entire mission, use the `/api/briefing` endpoint, which generates a narrated summary of all detected emitters, protocol decodes, and anomalies.

NLQ Accuracy

Natural-language query accuracy is approximately 80% on standard signal-census questions. Complex multi-step reasoning (e.g. "compare emitter activity across two frequency bands") may produce incorrect SQL or hallucinated figures. Always verify quantitative claims against the raw census data using DuckDB directly:

```
duckdb /var/lib/spectra/census.duckdb \
    "SELECT label, COUNT(*) FROM signal_hits
     WHERE timestamp_ms > epoch_ms(now()) - 3600000
     GROUP BY label ORDER BY count DESC"
```

3.10 Data Forensics & SigMF Recordings

While metadata analysis is Spectra's primary goal, the system can record raw IQ data for external analysis or forensic verification.

- **Recording Format**: Spectra uses the **SigMF v1.2.6** standard (interleaved complex float32, little-endian). Every recording includes a `.sigmf-data` file and a `.sigmf-meta` JSON metadata file.
- **Retrospective Capture**: Spectra maintains a 5-second circular buffer. When a `DEEP` capture is triggered, the buffer is included, providing the pre-trigger context of the signal.
- **Manual Recording**: Trigger a manual capture via the TUI (**R** key) or the REST API:

```
curl -X POST http://localhost:8000/api/capture/manual \
    -d '{"freq_hz": 433920000, "duration_s": 5.0}'
```

4 Mission Configuration Reference

4.1 MissionDef Fields

Table 1: MissionDef top-level fields

Field	Type	Description
`name`	string	Unique mission identifier (slug format)
`description`	string	Human-readable mission description
`frequencies_hz`	list[int]	List of centre frequencies to scan in Hz
`dwell_time_s`	float	Time to dwell on each frequency before stepping (seconds)
`pipeline_config`	PipelineConfig	Capture and processing pipeline settings
`stop_conditions`	StopConditions	Conditions that end the mission
`detection`	DetectionParams	Energy detector tuning parameters

4.2 PipelineConfig

Table 2: PipelineConfig fields

Field	Type	Description
`capture_level`	enum	`BASIC`, `DEEP`, or `FULL`
`classifier_confidence_gate`	float	Minimum confidence to persist classification (0.0–1.0)
`enable_protocol_decode`	bool	Run multimon-ng/DSDcc subprocess decoders
`enable_sei`	bool	Enable emitter fingerprinting pipeline
`enable_audio_transcription`	bool	Enable Whisper audio transcription
`audio_squelch_db`	float	Squelch threshold for audio capture (dBFS)
`sei_match_threshold`	float	ChromaDB cosine distance threshold for emitter matching
`max_decode_duration_s`	float	Maximum time to spend decoding a single signal

4.3 Capture Levels

The capture level controls how deeply Spectra processes each detected signal and directly governs the steering interruption budget.

Level	IQ Window	Pipeline Stages	Steer Budget
`BASIC`	128 samples	Fast classify only	~50 ms
`DEEP`	2048 samples	Fast + retrospective classify, decode, SEI	~500 ms
`FULL`	8192 samples	All stages + audio transcription + VAD	~2000 ms

Choosing a Capture Level

Use `BASIC` for dense frequency ranges or when minimising steering gaps is critical. Use `DEEP` for targeted monitoring where protocol decodes and emitter attribution are required. Reserve `FULL` for in-depth analysis of specific signals (e.g. recording and transcribing voice traffic on a known channel).

4.4 Detection Parameters

Table 3: DetectionParams fields

Field	Type	Default	Description
`threshold_db`	float	`-65.0`	Power threshold above noise floor (dB)
`merge_gap_hz`	int	`5000`	Merge adjacent detections within this gap (Hz)

Field	Type	Default	Description
`min_bandwidth_hz`	int	`1000`	Discard detections narrower than this (Hz)
`max_bandwidth_hz`	int	`500000`	Discard detections wider than this (Hz)
`hold_time_s`	float	`0.1`	Extend detection while signal persists (seconds)

5 API Reference

5.1 REST Endpoints

All endpoints are served at `http://<host>:8000` by default. Responses are JSON unless otherwise noted.

Health and Status

Command	Description
`GET /api/health`	Lightweight liveness check; returns `{"status":"ok"}`
`GET /api/health/deep`	Deep health: checks SDR, DuckDB, ChromaDB, Ollama connectivity
`GET /api/version`	Returns Spectra version string and build metadata

Signal Census

Command	Description
`GET /api/census`	Returns the last 100 signal hits (summary fields only)
`GET /api/census/deep`	Full census with decode text, SEI attribution, and metadata
`GET /api/census?freq=<hz>&window=<s>`	Filtered census for a specific frequency and time window

Emitter Registry

Command	Description
`GET /api/emitters`	List all known emitters with sighting counts
`GET /api/emitters/<id>`	Detailed emitter record including feature centroid
`PATCH /api/emitters/<id>`	Update emitter label or notes
`DELETE /api/emitters/<id>`	Remove emitter from registry

Missions

Command	Description
`GET /api/missions`	List all mission definitions
`POST /api/missions/load`	Load a MissionDef (JSON or YAML body)
`GET /api/missions/current`	Status of the currently running mission
`POST /api/missions/start`	Arm and start the loaded mission
`POST /api/missions/stop`	Gracefully stop the running mission

Intelligence

Command	Description
`GET /api/briefing`	Narrated mission briefing (Ollama LLM)
`POST /api/nlq`	Natural-language query (body: `{"query":"..."}`)

5.2 Deep Health Response

The `/api/health/deep` endpoint reports per-subsystem status, which is useful for diagnosing connectivity issues in multi-host deployments:

```
{
  "status": "degraded",
  "sdr":     {"status": "ok",      "device": "RTL-SDR Blog V3"},
  "duckdb":  {"status": "ok",      "path": "/var/lib/spectra/census.duckdb"},
  "chroma":  {"status": "ok",      "collection": "spectra_emitters"},
  "ollama":  {"status": "timeout", "model": "llama3.1:8b"},
  "whisper": {"status": "ok",      "backend": "pytorch"}
}
```

A `degraded` top-level status means one or more subsystems are unavailable. Core scanning and classification continue; affected pipeline stages (NLQ, audio transcription) are disabled until connectivity is restored.

5.3 WebSocket TLV Channels

Real-time data is streamed to connected clients over WebSocket at `ws://<host>:8000/ws`. Messages use a compact TLV (Type-Length-Value) binary framing designed for low-latency delivery to `spectra-tui` and edge nodes.

Type Byte	Channel	Payload
`0x01`	FFT frame	2048 float32 power values (dBFS)
`0x02`	Signal detection	DetectedSignal struct (binary packed)
`0x03`	Classification	SignalClassification (label, confidence, freq)
`0x04`	Decode event	Protocol decode text (UTF-8)
`0x05`	SEI attribution	Emitter ID and match distance
`0x06`	Mission status	Mission state enum + hit counter
`0x10`	DEMAND_CONTROL	Flow control from server to edge node

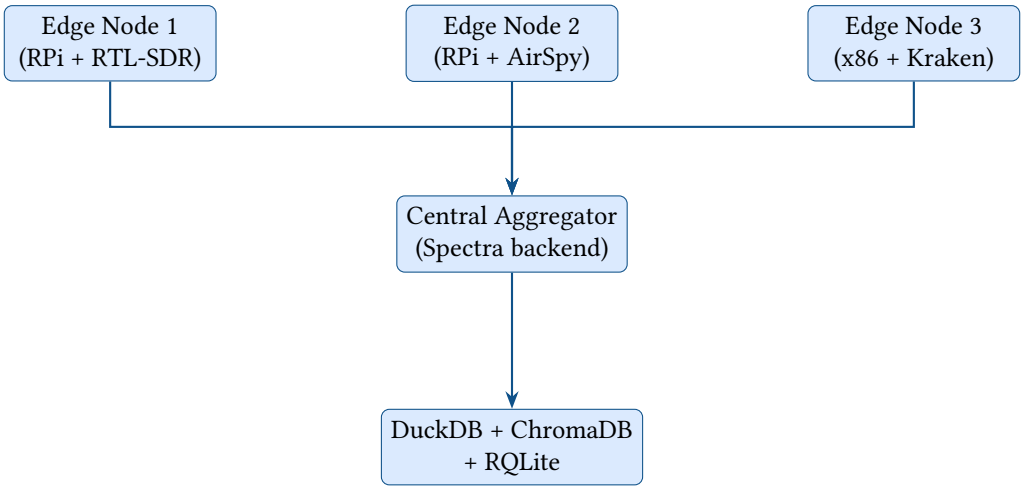
Clients that only need summary data may subscribe to specific channels by sending a `SUBSCRIBE` control message after the WebSocket handshake:

```
{"type": "SUBSCRIBE", "channels": [2, 3, 6]}
```

6 Edge Deployment

6.1 Overview

Spectra supports distributed deployments where lightweight edge nodes each host an SDR and run a minimal scanning agent. Edge nodes report detections to a central aggregator over the TLV WebSocket protocol, enabling multi-site spectrum monitoring from a single dashboard.



6.2 Raspberry Pi Edge Node Setup

Step 1. Install the edge agent on a Raspberry Pi 4 or 5 (64-bit Raspberry Pi OS):

```
pip install spectra-edge
# Build the RTL-SDR hardware server:
cargo build --release --manifest-path rtltcp-rust/Cargo.toml
sudo cp target/release/rtltcp-rust /usr/local/bin/
```

Step 2. Configure the edge agent to point at the central aggregator:

```
cat > /etc/spectra-edge.env << 'EOF'
AGGREGATOR_WS=ws://192.168.1.100:8000/ws
EDGE_NODE_ID=edge-rpi-01
EDGE_LOCATION="Site A, Rooftop"
RTLTCP_HOST=127.0.0.1
RTLTCP_PORT=1234
EOF
```

Step 3. Start both services:

```
systemctl enable --now rtltcp-rust spectra-edge
```

Step 4. Verify the edge node appears in the aggregator's health dashboard:

```
curl -s http://192.168.1.100:8000/api/health/deep | jq .edge_nodes
```

6.3 DEMAND_CONTROL Flow Control

Under high detection rates, edge nodes may produce more TLV messages than the aggregator can process. The `DEMAND_CONTROL` message (type `0x10`) carries a back-pressure signal from server to edge:

```
{
  "type": 16,
```

```
  "demand_factor": 0.5,   // 0.0 = stop sending; 1.0 = full rate
  "channel_mask": 7       // bitmask of channels to throttle
}
```

The edge agent responds by dropping or coalescing FFT frames (channel `0x01`) first, preserving detection events (channels `0x02`, `0x03`) at full rate.

6.4 Multi-Site Federation with RQLite

For geographically separated deployments, multiple Spectra aggregators can federate their DuckDB census data using RQLite as a distributed SQLite layer.

```
rqlited -node-id 1 -http-addr 192.168.1.100:4001 \
        -raft-addr 192.168.1.100:4002 /data/rqlite
rqlited -node-id 2 -http-addr 192.168.1.101:4001 \
        -raft-addr 192.168.1.101:4002 \
        -join 192.168.1.100:4001 /data/rqlite
rqlited -node-id 3 -http-addr 192.168.1.102:4001 \
        -raft-addr 192.168.1.102:4002 \
        -join 192.168.1.100:4001 /data/rqlite
```

Set `SPECTRA_RQLITE_URL=http://192.168.1.100:4001` in each aggregator's environment file to enable federated census queries across all sites.

Edge Node Latency

Raspberry Pi 4 edge nodes running RTL-SDR at 2.4 MS/s typically introduce 200–400 ms of end-to-end latency between signal detection and aggregator receipt, depending on network conditions. This is acceptable for most monitoring missions but may cause missed transient signals shorter than 300 ms. Use a direct Ethernet connection rather than Wi-Fi for critical deployments.

7 Troubleshooting

Problem 1: Single SDR misses signals during steering

Cause: The SDR can only tune to one frequency at a time. While Spectra is capturing a burst for classification or decode, the scanner is paused and signals on other frequencies are missed.

Solution: Reduce `capture_level` to `BASIC` to shorten steering windows. For comprehensive coverage, deploy a second SDR on an overlapping frequency range and register it as a second `rtltcp-rust` instance.

Problem 2: Classifier labels a signal as "UNKNOWN" or low confidence

Cause: The signal modulation is not represented in the training set, or the IQ capture was taken during a guard period or transient.

Solution: Increase `dwell_time_s` to ensure the capture window coincides with the active emission. If the modulation class is genuinely absent, retrain the classifier using the synthetic training lab; see *SPECTRA-TR-001* Section 7.

Problem 3: Energy detector generates excessive false positives

Cause: The configured `threshold_db` is too close to the noise floor, causing noise spikes to trigger detections.

Solution: Increase `threshold_db` by 5–10 dB and set `min_bandwidth_hz` to filter out narrowband spurs. Run the noise floor estimator: `curl -s http://localhost:8000/api/noise_floor | jq .p99_db` and use a threshold at least 8 dB above the p99 value.

Problem 4: Natural-language query returns wrong numbers or hallucinated data

Cause: The LLM generated incorrect SQL or drew on parametric knowledge rather than the retrieved census context.

Solution: Verify the figure directly in DuckDB (see the example in Section 3.9). Improve retrieval quality by ensuring the DuckDB census is up to date (`/api/census/deep`). Switching to a larger Ollama model (e.g. `llama3.1:70b` on a capable server) increases SQL generation accuracy.

Problem 5: Audio transcription produces garbled or empty output

Cause: Noisy RF, low squelch threshold, or recording during the transient leading edge of a transmission causes Whisper to transcribe static or silence.

Solution: Increase `audio_squelch_db` by 10 dB and enable `enable_vad: true` so Whisper only processes speech-active segments. On Apple Silicon, ensure `SPECTRA_WHISPER_BACKEND=mlx` and that the MLX model is cached locally.

Problem 6: subprocess decoder (`multimon-ng`) crashes immediately

Cause: A missing shared library (`libsndfile`, `libpulse`) or incompatible binary prevents the decoder from starting.

Solution: Run `ldd $(which multimon-ng) | grep "not found"` to identify missing libraries. Install them via the system package manager. As a workaround, disable protocol decoding temporarily with `enable_protocol_decode: false`.

Problem 7: Edge node does not appear in aggregator health check

Cause: Firewall rules blocking port 8000, incorrect `AGGREGATOR_WS` URL, or `spectra-edge` service failing to start.

Solution: Check `systemctl status spectra-edge` on the edge node. Verify network connectivity: `curl -s ws://<aggregator>:8000/api/health`. Ensure port 8000 is open on the aggregator host.

8 Integration with Other Tools

8.1 Passive Radar

Spectra and the Passive Radar system share the Kraken SDR hardware platform. When running in passive radar mode, the Kraken's five coherent channels are allocated to direction-finding; Spectra's scanning missions must be suspended on that hardware instance.

See also: *Passive Radar User Manual* (PASSIVE-UM-001) — Kraken SDR channel allocation and switching between Spectra and passive radar modes.

8.2 rtltcp-rust

`rtltcp-rust` is the SDR hardware abstraction layer that Spectra depends on. It provides unified TCP streaming, a REST control interface, and gain/frequency management for RTL-SDR, AirSpy, and Kraken devices.

```
rtltcp-rust --help
rtltcp-rust --device 0 --sample-rate 2400000
rtltcp-rust --list-devices          # enumerate attached SDRs
rtltcp-rust --device airspy:0 --sample-rate 10000000
```

See also: *rtltcp-rust User Manual* (RTLTCP-UM-001) — Device enumeration, gain profiles, and REST API reference.

8.3 DuckDB Signal Census

All signal hits, classifications, and emitter attributions are stored in a DuckDB database. DuckDB can be queried directly for custom analytics beyond the built-in NLQ interface:

Listing 2: Example: top 10 emitters by sighting count

```sql
SELECT
    e.emitter_id,
    e.label,
    COUNT(h.id)       AS sightings,
    MIN(h.timestamp_ms) AS first_seen_ms,
    MAX(h.timestamp_ms) AS last_seen_ms
FROM signal_hits h
JOIN emitters e ON h.emitter_id = e.emitter_id
GROUP BY e.emitter_id, e.label
ORDER BY sightings DESC
LIMIT 10;
```

See also: *DuckDB Documentation* — `https://duckdb.org/docs` — SQL dialect reference, JSON functions, and `ATTACH` for multi-database federation.

9 Further Reading

- *Spectra Technical Reference* (SPECTRA-TR-001) — Architecture, data model, subsystem design rationale, training pipeline, and API contract definitions. The primary reference for developers extending the platform.
- *rtltcp-rust Technical Reference* (RTLTCP-TR-001) — TLV framing specification, device abstraction layer design, and performance characterisation across SDR hardware families.
- *Passive Radar Technical Reference* (PASSIVE-TR-001) — Coherent multi-channel processing, Kraken SDR integration, and direction-finding algorithms.
- *RadioML 2018.01a Dataset* — DeepSig, 2018. Training dataset for the Spectra modulation classifier. `https://www.deepsig.ai/datasets`
- *SigIDWiki* — Community-maintained signal identification database used for post-classification enrichment. `https://www.sigidwiki.com`