Spectra: Distributed Spectrum Intelligence Platform

Technical Reference

Simon Knight

Independent Researcher, Adelaide, Australia

March 2026 • Version 1.0

Abstract. Spectra is a distributed spectrum intelligence platform that autonomously monitors, classifies, and analyses radio-frequency signals across the HF–UHF range using software-defined radio hardware. The system comprises four cooperating tiers: a Rust-based SDR hardware server (`rtltcp-rust`) that abstracts RTL-SDR, AirSpy, and Kraken receivers behind a unified TCP/REST interface; a Python orchestrator (FastAPI) that performs real-time DSP, energy-based signal detection, two-stage ML classification, protocol decoding, emitter fingerprinting, and autonomous mission management; a Rust terminal UI (`spectra-tui`) with GPU-accelerated waterfall rendering; and a React web frontend for visualisation and natural-language querying. This report documents the architecture, design rationale, and API surface of each subsystem, covering the full pipeline from IQ sample acquisition through signal classification to narrated intelligence briefings.

Contents

1 Introduction

1.1 What Spectra Solves

Spectrum monitoring has traditionally required either expensive, proprietary receivers that lock the operator into a single vendor's analysis software, or fragmented open-source tools that each handle one piece of the pipeline: one tool to tune the radio, another to display a waterfall, another to decode a protocol. Stitching these together into an autonomous system that detects, classifies, identifies, and reports on signals around the clock is left as an exercise for the operator.

Spectra replaces this patchwork with a single, distributed platform. A user connects one or more software-defined radios (SDRs), points Spectra at a frequency range, and receives classified, enriched intelligence — modulation labels, protocol decodes, emitter fingerprints, and natural-language briefings — without manual intervention.

1.2 Design Philosophy

Two principles guide every design decision in Spectra:

1. **Edge-to-intelligence pipeline.** Every stage from IQ acquisition to briefing generation is connected by typed, binary protocols. There is no file-based hand-off between stages; data flows as streams. A signal detected at the edge can reach a narrated briefing in under one second.
2. **Thin clients, thick server.** The Rust TUI and React web UI are intentionally thin. They render data and accept user commands. All DSP, ML inference, protocol decoding, and mission logic lives in the Python backend. This means adding a new client (mobile, embedded) requires only implementing the TLV WebSocket protocol, not reimplementing the intelligence pipeline.

1.3 Who Should Read This Document

- **SDR operators** deploying Spectra for spectrum monitoring who want to understand how classification, detection, and missions work under the hood.
- **Developers** extending the platform with new decoders, classifiers, or client applications, needing to understand the module structure and API contracts.
- **Researchers** building on Spectra's synthetic training lab or emitter fingerprinting capabilities.

1.4 How to Read This Document

Section 2 covers the data model — the structures that flow between components. Section 3 presents the four-tier architecture. Sections 4–10 cover the six core subsystems in depth, ordered from deepest (signal detection) to broadest (audio intelligence). Section 15 documents the REST and WebSocket API surface. Section 16 lists known limitations with workarounds.

Readers wanting a quick start should read §3 (Architecture) and §15 (API Reference), then refer to individual subsystem sections as needed.

2 Data Model

Spectra's data model centres on four structures that flow through the pipeline.

2.1 DetectedSignal

The atomic unit of the detection pipeline. Produced by the energy detector and consumed by every downstream stage.

Listing 1. DetectedSignal dataclass

```
@dataclass
```

```
class DetectedSignal:
    freq_start_hz: float   # Lower edge of detected emission
    freq_end_hz: float     # Upper edge
    power_db: float        # Peak power in dB
    mean_power_db: float   # Mean power across bandwidth
    bandwidth_hz: float    # freq_end - freq_start
```

2.2 SignalClassification

The output of the two-stage classification pipeline. Carries the ML label, confidence, alternatives, and optional SigIDWiki enrichment.

Listing 2. SignalClassification dataclass

```
@dataclass
class SignalClassification:
    label: str                              # e.g. "FM", "QPSK", "POCSAG"
    confidence: float                       # 0.0--1.0 calibrated probability
    alternatives: list[tuple[str, float]]   # Runner-up labels
    freq_start_hz: float
    freq_end_hz: float
    timestamp_s: float
    is_retrospective: bool = False          # True if from deep analysis
    sigidwiki_matches: list[SignalMetadata] = []
    metadata_source: str = "ml"             # "ml", "sigidwiki", "combined"
```

2.3 Signal Hit (Persistent Record)

When a detection is significant enough to persist, it becomes a row in the DuckDB `signal_hits` table:

Listing 3. signal_hits schema

```
CREATE TABLE signal_hits (
    id              VARCHAR PRIMARY KEY,
    timestamp_ms    BIGINT NOT NULL,
    center_freq_hz  DOUBLE NOT NULL,
    sample_rate_hz  DOUBLE NOT NULL,
    freq_start_hz   DOUBLE NOT NULL,
    freq_end_hz     DOUBLE NOT NULL,
    bandwidth_hz    DOUBLE NOT NULL,
    peak_db         DOUBLE NOT NULL,
    mean_power_db   DOUBLE NOT NULL,
    device_name     VARCHAR NOT NULL,
    emitter_id      VARCHAR,      -- FK to emitters table (SEI)
    label           VARCHAR,      -- classification label
    confidence      DOUBLE,
    artifact_path   VARCHAR       -- path to SigMF recording
);
```

2.4 Mission Lifecycle

Missions are the unit of autonomous operation. A `MissionDef` describes *what* to scan; a `MissionRun` tracks the execution state.

Listing 4. Mission data model (simplified)

```
@dataclass
class MissionDef:
    name: str
    freq_start_hz: float
    freq_end_hz: float
```

```
    dwell_ms: int
    modulations: list[str]        # filter to these modulation types
    capture_level: CaptureLevel   # BASIC | DEEP | FULL

@dataclass
class MissionRun:
    run_id: str
    def_id: str
    state: MissionStatus          # queued | running | paused | completed
    created_at: datetime
    events: list[MissionEvent]
```

Capture levels

`BASIC`: classification only. `DEEP`: classification + protocol decode + SigMF recording. `FULL`: all of DEEP plus SEI fingerprinting and emitter vector storage.

3 Architecture

3.1 Four-Tier Design

Spectra is composed of four cooperating tiers, each in its own process:

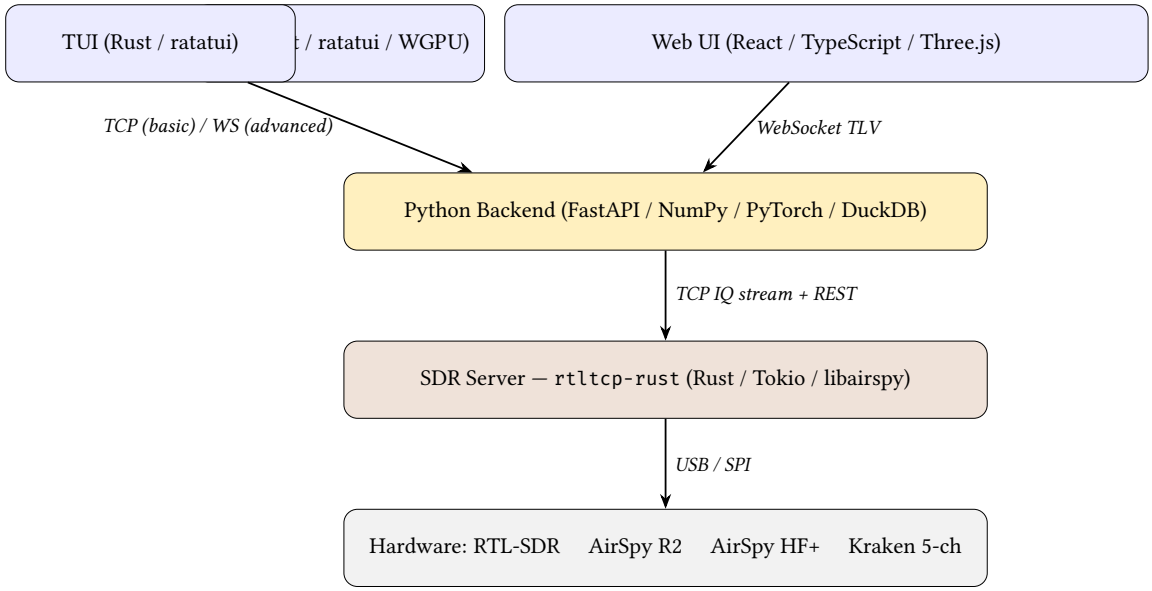

Figure 1. Spectra four-tier architecture. Thin arrows indicate the primary data-flow direction (IQ flows up; commands flow down).

3.2 Technology Stack

Layer	Technology	Purpose
SDR Server	Rust, Tokio, libairspy, rtlsdr-sys	Hardware abstraction, IQ streaming
Backend	Python 3.10+, FastAPI, Uvicorn	Orchestration, DSP, ML, missions
ML Inference	PyTorch, CoreML, ONNX Runtime	Modulation classification
Transcription	mlx-whisper (Apple MLX)	Speech-to-text on Apple Silicon
Persistence	DuckDB	Signal census, emitter database
Vector Search	ChromaDB	Emitter embedding similarity
NL Querying	Ollama (llama3.2:1b)	Natural-language briefings
TUI	Rust, ratatui, crossterm, WGPU	Terminal interface, GPU waterfall
Web UI	React, TypeScript, Three.js	Browser-based visualisation

3.3 Project Layout

Path	Contents
`src/spectra/core/`	Orchestrator, DSP, streaming, protocol handlers
`src/spectra/intelligence/`	Classifier, detector, decoders, fingerprinting, missions
`src/spectra/autonomy/`	Autopilot, satellite scheduler, SigMF recorder
`src/spectra/api/`	FastAPI routes (REST + WebSocket)
`crates/spectra-tui/`	Rust TUI application
`signals/spectra-web/`	React web frontend

3.4 Wire Protocol: TLV over WebSocket

All client–server communication uses a custom Type-Length-Value binary protocol over WebSocket. This was chosen over JSON-per-message because spectrum data is high-bandwidth (1000+ FFT frames/sec) and benefits from compact binary encoding.

Listing 5. TLV envelope format

```
# 8-byte envelope: [channel: u8] [flags: u8] [seq: u16 LE] [length: u32 LE]
# Followed by: [payload: N bytes]
HEADER_SIZE = 8

class Channel(IntEnum):
    SPECTRUM       = 0x01  # Binary: u8 bins with 16-byte header
    DETECTIONS     = 0x02  # JSON array of detected signals
    CLASSIFICATION = 0x03  # JSON classification result
    AUDIO          = 0x04  # Binary: i16 LE PCM with header
    CONTROL        = 0x05  # JSON command/response
    META           = 0x06  # JSON session metadata
    SPARSE_IQ      = 0x07  # Binary: complex64 IQ (edge ingest)
    DEMAND_CONTROL = 0x08  # JSON: server->edge demand toggles
```

Compression

Flag bit 0 (`COMPRESSED`) indicates zlib-deflated payload. This is used for large JSON payloads (detections, classifications) but never for spectrum or audio data, where the overhead of compression exceeds the size saving.

4 Signal Detection

4.1 Problem Statement

Given a stream of FFT power spectra (one per frame, typically 10–30 fps), identify frequency regions that contain active transmissions and pass them downstream for classification.

4.2 Design Options Considered

A. Adaptive noise floor estimation Continuously estimate the noise floor using a sliding median and trigger on exceedance. Used by GNU Radio and many commercial receivers.

B. Fixed energy threshold Compare each FFT bin against a configurable power threshold in dB. Simple and deterministic.

C. Matched-filter detection Correlate against known signal templates. Optimal for known signals but requires a template library.

4.3 Decision and Rationale

Spectra uses **Option B (fixed energy threshold)** with configurable parameters. The rationale: simplicity and determinism. In a system that already has a downstream ML classifier, the detector's job is not to identify signals — only to find frequency regions worth classifying. A simple threshold with a merge step and minimum-bandwidth filter achieves this reliably. Adaptive noise-floor estimation adds complexity and state that is difficult to reason about during debugging.

4.4 Implementation

The `EnergyDetector` class implements a three-phase pipeline:

Algorithm 1 Energy-based signal detection

Require: Power spectrum $P[0..N-1]$ in dB, frequency axis $F[0..N-1]$
Require: Threshold τ (dB), merge gap g (Hz), min bandwidth $b_{\min}$ (Hz)
1: $above \leftarrow \{i : P[i] > \tau\}$
2: Group consecutive indices in $above$ into contiguous regions
3: **for** each region $[i_{\text{start}}, i_{\text{end}}]$ **do**
4: Create `DetectedSignal` with $F[i_{\text{start}}]$, $F[i_{\text{end}}]$, $\max(P[i_{\text{start}}..i_{\text{end}}])$, $\text{mean}(P[i_{\text{start}}..i_{\text{end}}])$
5: **end for**
6: Merge signals whose gap $< g$
7: Discard signals whose bandwidth $< b_{\min}$
8: **return** remaining signals

Default parameters: $\tau = -80$ dB, $g = 2000$ Hz, $b_{\min} = 5000$ Hz.

4.5 Consequences and Trade-offs

The fixed threshold means that in noisy environments, the detector may produce false positives (the classifier handles them). In quiet environments, weak signals below −80 dB are missed. Operators can adjust `threshold_db` per-device via the mission configuration.

5 Two-Stage Classification

5.1 Problem Statement

Given a detected signal (frequency bounds + IQ samples), assign a modulation label from a set of 14 standard classes: AM-DSB, FM, WBFM, BPSK, QPSK, 8PSK, QAM16, QAM64, GFSK, CPFSK, OOK, and others.

5.2 Design Options Considered

A. Single-pass classifier Run one model on every detection. Simple but slow models limit throughput; fast models sacrifice accuracy on edge cases.

B. Two-stage pipeline A fast model for real-time classification with a confidence gate. Low-confidence results are queued for retrospective deep analysis using a larger model and more IQ samples.

C. Ensemble voting Run multiple models and vote. Highest accuracy but highest cost.

5.3 Decision and Rationale

Spectra uses **Option B (two-stage pipeline)** because it gives the best latency/accuracy trade-off. Most signals (strong FM broadcasts, WBFM, etc.) are classified with high confidence in the first stage. Only ambiguous signals (weak digital modes, overlapping emissions) trigger the expensive retrospective path.

5.4 Implementation

Listing 6. Classification pipeline configuration

```
@dataclass
class PipelineConfig:
    confidence_threshold: float = 0.6   # Below this triggers retrospective
    retrospective_enabled: bool = True
    max_retrospective_queue: int = 32
    retrospective_sample_count: int = 8192
    detection_threshold_db: float = -80.0
    sigidwiki_enabled: bool = True
    sigidwiki_on_low_confidence: bool = True
```

The real-time stage runs on every FFT frame. When confidence falls below 0.6, the signal's frequency bounds and timestamp are placed on a bounded async queue. A background coroutine retrieves historical IQ samples from the `CircularIQBuffer` (a rolling window of recent frames) and runs the retrospective model.

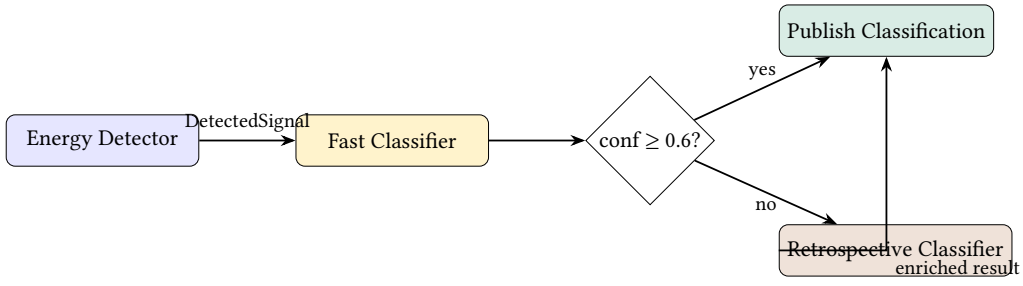

Figure 2. Two-stage classification pipeline. The confidence gate at 0.6 routes low-confidence detections to the retrospective classifier.

5.5 Model Backends

Three inference backends are supported, selectable at startup:

Backend	Runtime	Notes
CoreML	Apple Neural Engine	Fastest on Apple Silicon (M-series)
ONNX Runtime	CPU / GPU	Cross-platform, used for deployment
PyTorch	CPU / MPS	Used during training and development

5.6 SigIDWiki Enrichment

When the ML classifier produces a label, the `metadata/sigidwiki.py` module optionally queries the Signal Identification Wiki database to enrich the classification with known service names, frequency allocations, and protocol descriptions. This is especially useful for narrowband digital modes (POCSAG, DMR, P25) where the modulation label alone ("FSK") is not informative.

> **Warning**
>
> SigIDWiki enrichment requires network access and adds 50–200 ms latency. It is disabled by default in low-latency mission profiles and enabled only when `sigidwiki_on_low_confidence` is set.

6 Autonomous Missions

6.1 Problem Statement

An unattended monitoring station must scan designated frequency bands, log detections, and stop when a condition is met (time limit, hit count, rarity threshold) — all without operator intervention.

6.2 Design Options Considered

A. Cron-based sweeps Schedule frequency sweeps as periodic jobs. Simple but inflexible — cannot react to what is being received.

B. Mission engine with priority arbitration Define missions as first-class objects with lifecycle state, stop conditions, and priority-based hardware steering.

C. Rule-based reactive system Define triggers ("if signal X detected, then do Y"). Flexible but hard to reason about in combination.

6.3 Decision and Rationale

Spectra uses **Option B (mission engine)** because missions are the natural unit of operator intent. An operator says "watch 430–440 MHz for 30 minutes, capture anything unusual at FULL depth." This maps directly to a `MissionDef`.

The priority arbitration model allows concurrent missions without resource conflicts: only the highest-priority active mission steers the SDR hardware. Lower-priority missions continue to observe and log but cannot retune.

6.4 Implementation

The `MissionEngine` manages the full lifecycle:

1. **Definition**: operator creates a `MissionDef` (frequency range, dwell time, modulation filters, capture level, stop conditions).
2. **Scheduling**: the `MissionScheduler` queues the mission and transitions it to `running` when resources are available.
3. **Execution**: the engine retunes the SDR (if this mission holds steering priority), receives detections from the pipeline, scores each hit via the `NotabilityScorer`, and evaluates stop conditions.

4. **Reporting**: on completion, the `BriefingGenerator` produces a structured markdown briefing and daily logbook entry.
5. **Focus reload**: the `FocusStore` writes current scanning constraints to a file, polled every 500 ms. External tools can modify the focus file to steer the mission in real time.

6.5 Stop Conditions

Three stop-condition types are supported, composable via AND/OR:

Condition	Parameter	Semantics
Time limit	`max_duration_s`	Mission ends after N seconds
Hit count	`max_hits`	Mission ends after N detections
Rarity	`min_notability`	Mission ends when a signal with notability $\geq$ threshold is found

6.6 Consequences and Trade-offs

The single-SDR steering model means that only one mission at a time controls the radio. In a multi-SDR deployment, each SDR can be assigned to a different mission via the `device_name` parameter. The focus-file polling mechanism (500 ms interval) introduces up to 500 ms of steering latency, acceptable for HF/VHF monitoring but potentially too slow for burst-mode signals.

7 Emitter Fingerprinting (SEI)

7.1 Problem Statement

Two transmitters on the same frequency using the same modulation appear identical to the classifier. Specific Emitter Identification (SEI) aims to distinguish individual radios by their hardware-specific RF signatures — imperfections in the oscillator, mixer, and ADC that produce unique "fingerprints" in the IQ data.

7.2 Design Options Considered

A. Handcrafted features + nearest-neighbour Extract known features (I/Q imbalance, CFO, transient shape) and classify with k-NN.

B. Deep metric learning (TripletNet) Train a neural network to map IQ bursts into an embedding space where same-emitter bursts cluster. Use vector search for identification.

C. Raw IQ classification Train a supervised classifier directly on emitter-labelled IQ data. Requires labelled data for every emitter.

7.3 Decision and Rationale

Spectra uses **Option A for feature extraction** combined with **Option B for identification**. Handcrafted features (I/Q imbalance, phase noise, transient edge shape via GLRT) are computed first, then concatenated with learned embeddings from the TripletNet. This hybrid approach gives interpretable features (useful for debugging) while benefiting from the representation power of deep learning.

7.4 Feature Extraction

The `fingerprinting/feature_extract.py` module computes:

- **I/Q imbalance**: amplitude ratio g and phase error ϕ, computed as:

$$g = \sqrt{\frac{E[Q^2]}{E[I^2]}}, \qquad \phi = \arcsin\left(\frac{E[IQ]}{\sqrt{E[I^2]}\,\sqrt{E[Q^2]}}\right)$$

  Perfect balance gives g=1, ϕ=0.
- **Carrier frequency offset (CFO)**: estimated from the phase rotation rate of the IQ stream.
- **Transient edge shape**: the GLRT (Generalised Likelihood Ratio Test) detector in `fingerprinting/transient_det` identifies burst edges. The transient envelope at turn-on is a hardware signature.

7.5 Vector Search

Fingerprint vectors are stored in ChromaDB. When a new burst arrives, its embedding is compared against the database using cosine similarity. If the nearest neighbour exceeds a configurable threshold, the burst is linked to that emitter's ID in the DuckDB `emitters` table.

Emitter linking flow

```
# 1. Extract features from IQ burst
g, phi = estimate_iq_imbalance(iq_data)
cfo = estimate_cfo(iq_data, sample_rate)

# 2. Compute TripletNet embedding
embedding = triplet_model.encode(iq_data)

# 3. Concatenate handcrafted + learned features
feature_vec = np.concatenate([[g, phi, cfo], embedding])

# 4. Search ChromaDB for nearest emitter
match = vector_search.query(feature_vec, threshold=0.85)

# 5. Link detection to emitter in DuckDB
if match:
    persistence.link_hit_to_emitter(hit_id, match.emitter_id)
```

8 Protocol Decoding

8.1 Problem Statement

Once a signal is classified (e.g., as "POCSAG" or "DMR"), the raw IQ data should be decoded into human-readable protocol content — pager messages, radio IDs, RDS station names.

8.2 Design Options Considered

A. Native decoders in Python Write decoders from scratch. Full control but enormous effort for mature protocols.

B. Subprocess wrappers around established tools Shell out to Multimon-ng, DSDcc, and similar tools. Leverages decades of community work.

C. Rust decoder library Compile decoders to a shared library callable from Python. Best performance but highest integration cost.

8.3 Decision and Rationale

Spectra uses **Option B (subprocess wrappers)** because Multimon-ng and DSDcc are battle-tested decoders with wide protocol coverage. The wrapper approach means Spectra benefits from upstream bugfixes without maintaining its own codec implementations.

8.4 Implementation

The `ProtocolManager` in `decoders/protocol_manager.py` manages decoder subprocess lifecycles:

- **Multimon-ng**: decodes POCSAG, RDS, DTMF. Receives audio via stdin pipe; emits JSON on stdout.
- **DSDcc**: decodes DMR, P25. Receives audio via stdin; emits voice metadata extracted by regex from stderr.

Decoded content is stored in the DuckDB `decoded_content` table, linked to the originating signal hit by `hit_id`.

> **Warning**
>
> Subprocess-based decoding adds 100–300 ms startup latency per decoder instance. The `ProtocolManager` keeps decoder processes alive across detections to amortise this cost. If a decoder crashes, it is automatically restarted on the next matching detection.

9 Synthetic RF Training Lab

9.1 Problem Statement

Training a modulation classifier requires large, labelled IQ datasets. Over-the-air recordings are expensive to collect and difficult to label accurately. Synthetic data generation provides unlimited, perfectly labelled training examples, but the quality of the classifier depends on the realism of the synthetic data.

9.2 Design Options Considered

A. Use existing datasets (RadioML, DeepSig) Pre-made datasets. Convenient but limited to fixed SNR ranges and modulation sets.

B. Generate synthetic baseband + channel impairments Write deterministic baseband generators for each modulation, then apply configurable channel models (AWGN, Doppler, Rayleigh fading).

C. GAN-based augmentation Train a GAN to generate realistic IQ samples. High realism but difficult to control and label.

9.3 Decision and Rationale

Spectra uses **Option B (deterministic generators + channel models)** because it gives complete control over the signal-to-noise ratio, fading profile, and frequency offset of each training example. Labels are ground-truth by construction. The synthetic pipeline can generate arbitrarily large datasets in minutes on a single CPU.

9.4 Implementation

The synthetic lab comprises five modules:

Module	Responsibility
`synthetic/generator.py`	Baseband IQ: AM, FM, FSK, BPSK, QPSK, QAM
`synthetic/channel.py`	Channel impairments: AWGN, Doppler, Rayleigh
`synthetic/dataset.py`	PyTorch Dataset for SigMF recordings
`synthetic/model.py`	1D-ResNet classifier architecture
`synthetic/train.py`	Training loop; exports ONNX

Generating a synthetic QPSK signal with channel impairments

```
from spectra.intelligence.synthetic.generator import generate_qpsk
from spectra.intelligence.synthetic.channel import apply_awgn, apply_doppler

# Deterministic baseband generation
rng = np.random.default_rng(seed=42)
iq = generate_qpsk(fs_hz=2e6, duration_s=0.01,
                   carrier_hz=0, sym_rate=50e3, rng=rng)

# Channel impairments
iq = apply_awgn(iq, snr_db=10.0, rng=rng)
iq = apply_doppler(iq, fs_hz=2e6, shift_hz=150.0)
# iq is now a labelled "QPSK at 10 dB SNR" training example
```

The training loop in `synthetic/train.py` fine-tunes a 1D-ResNet on the generated data and exports the result as an ONNX model, ready for deployment via the ONNX Runtime backend.

10 Audio Content Intelligence

10.1 Problem Statement

Voice-bearing signals (FM broadcast, AM aviation, SSB amateur) carry semantic content that cannot be captured by modulation classification alone. Converting demodulated audio to text enables full-text search over the signal census and richer briefings.

10.2 Design Options Considered

A. Cloud STT (Whisper API, Google STT) Highest accuracy but requires internet access and introduces latency and privacy concerns.

B. Local Whisper via mlx-whisper Runs entirely on Apple Silicon using the MLX framework. Moderate accuracy, zero network dependency.

C. Vosk / PocketSphinx Lightweight local inference. Lower accuracy than Whisper.

10.3 Decision and Rationale

Spectra uses **Option B (mlx-whisper)** because the target deployment platform is Mac mini M-series, which has dedicated Neural Engine hardware that MLX exploits. Running locally means no network dependency, no per-minute cost, and no privacy risk from sending intercepted audio to a cloud service.

10.4 Implementation

The audio intelligence pipeline has three stages:

1. **Demodulation**: the `AudioStreamer` translates the SDR's IQ stream into the signal of interest. It frequency-shifts the target signal to baseband, applies a channel filter, decimates, and demodulates using the appropriate demodulator (FM/AM/SSB). Output is 48 kHz float audio.
2. **Voice Activity Detection (VAD)**: the audio streamer accumulates 5-second chunks. When energy exceeds a threshold (indicating voice rather than silence or noise), the chunk is dispatched for transcription.
3. **Transcription**: the `AudioTranscriber` resamples to 16 kHz (Whisper's required rate using `scipy.signal.resample_poly`), runs mlx-whisper inference, and returns the text. Transcripts are stored in DuckDB alongside the originating signal hit.

Listing 7. AudioTranscriber core

```python
class AudioTranscriber:
    def __init__(self, model_path="mlx-community/whisper-tiny-mlx"):
        self.model_path = model_path
        self.target_fs = 16000  # Whisper requires 16 kHz

    def transcribe(self, audio_data: np.ndarray, sample_rate: int) -> str:
        audio_16k = self._resample_if_needed(audio_data, sample_rate)
        result = mlx_whisper.transcribe(
            audio_16k, path_or_hf_repo=self.model_path, fp16=True
        )
        return result.get("text", "").strip()
```

Note

The Whisper "tiny" model is used by default for speed (<1 s per 5-second chunk on M2). For higher accuracy on noisy RF audio, operators can switch to "small" or "medium" by changing `model_path`.

11 Natural Language Querying (RAG)

11.1 Problem Statement

Operators want to ask questions like "What unusual signals were detected on UHF last night?" and receive a prose answer, not a SQL result set.

11.2 Implementation

Spectra uses a Retrieval-Augmented Generation (RAG) architecture:

1. The operator's natural-language question is sent to the `/api/nlq` endpoint.
2. The `CustomNLSQLQueryEngine` prompts a local Ollama LLM (llama3.2:1b) to translate the question into a SQL query against the DuckDB schema.
3. The `DuckDBSQLRetriever` executes the SQL and returns tabular results.
4. The LLM synthesises a prose answer from the SQL results, streamed back to the client via Server-Sent Events.

Warning

The text-to-SQL step is imperfect. The 1B-parameter model occasionally generates invalid SQL or misinterprets column names. The retriever wraps execution in a try/except and returns the error message to the LLM for self-correction. In practice, roughly 80% of simple queries succeed on the first attempt.

12 SDR Hardware Server (`rtltcp-rust`)

12.1 Problem Statement

Different SDR hardware (RTL-SDR, AirSpy R2, AirSpy HF+, Kraken 5-channel coherent) each have their own driver APIs. Clients should not need to know which hardware is connected.

12.2 Design and Implementation

`rtltcp-rust` is a standalone Rust server that presents a unified interface:

- **TCP IQ streaming**: standard RTL-TCP protocol for backward compatibility with existing tools (e.g., SDR#, GQRX), plus an extended RTE1 protocol with a 5-byte frame header (1 byte type + 4 bytes LE length) for future extensibility.
- **REST API**: device enumeration, health checks, and system metrics.

Endpoint	Method	Description
`/api/v1/health`	GET	Server status
`/api/v1/server`	GET	Name, version, uptime, device count
`/api/v1/devices`	GET	List SDRs with stats
`/api/v1/devices/{name}`	GET	Single device detail
`/api/v1/system`	GET	CPU, memory, temperature

The server uses Tokio for async I/O. Each connected client gets a dedicated task that reads from the device's sample ring buffer, applies any per-client frequency/gain settings, and writes IQ frames to the TCP socket.

> **RTE1 protocol**
>
> The 5-byte RTE1 header was chosen over the standard 4-byte RTL-TCP header to allow a type byte for future message types (metadata, calibration data, device events) without breaking backward compatibility. Clients that send the standard RTL-TCP handshake receive standard framing; clients that send the RTE1 handshake receive extended framing.

13 Rust TUI (`spectra-tui`)

The terminal UI operates in two modes:

Basic mode Direct TCP connection to `rtltcp-rust`. Displays waterfall and spectrum. No classification or audio. Suitable for lightweight monitoring when the Python backend is not running.

Advanced mode WebSocket connection to the Python backend. Receives classified detections, audio, and briefings via the TLV protocol. Full feature set.

13.1 GPU-Accelerated Waterfall

The optional `native-waterfall` feature compiles a WGPU compute shader that renders the waterfall image on the GPU. The CPU-side ring buffer pushes new spectral rows; the shader maps power values to a colour palette and composites the scrolling image. This achieves smooth rendering at 60+ fps even with 4096-bin FFTs.

When WGPU is not available (e.g., over SSH), the TUI falls back to a CPU-based renderer using Kitty graphics protocol for inline image display.

13.2 Dual-Mode Architecture Rationale

Keeping basic mode ensures the TUI is useful even without the full Python stack. A field operator can connect a single RTL-SDR, launch the TUI, and get a waterfall display with no dependencies beyond the Rust binary and `rtltcp-rust`. Advanced mode adds intelligence when the infrastructure is available.

14 Edge Architecture

For distributed deployments, Spectra supports a hub-and-spoke model:

- **Edge nodes** (Raspberry Pi + RTL-SDR) run a lightweight Python agent that streams FFT spectra and sparse IQ to the central server via the `SPARSE_IQ` and `SPECTRUM` TLV channels.
- **Central server** (Mac mini M-series) runs the full Python backend, performing classification, fingerprinting, and persistence.

The edge ingest server (`core/edge_ingest_server.py`) accepts registrations from edge nodes, manages their lifecycles, and can issue demand-control messages (e.g., "send full IQ for the next 5 seconds") via the `DEMAND_CONTROL` channel.

> **Federation**
>
> For multi-site deployments, Spectra supports federated census via RQLite (`intelligence/federation/rqlite_bridge.py`). Each site maintains a local DuckDB census; the RQLite bridge replicates summaries to a shared distributed SQLite database for cross-site querying.

15 API Reference

15.1 REST Endpoints

Endpoint	Method	Description
`/api/health/deep`	GET	System health: device stats, WS queue pressure, audio counters
`/api/census/deep`	GET	Query signal census with time/frequency filters
`/api/archive/stats`	GET	Rarity index of captured signal types
`/api/emitters`	GET	List known emitters with fingerprint metadata
`/api/briefing`	GET	Narrated intelligence summary for time window
`/api/nlq`	POST	Natural-language query (SSE streaming response)
`/api/tdoa`	POST	Time Difference of Arrival geolocation
`/api/edge/register`	POST	Edge node registration
`/api/edge/sparse-ingest`	POST	Sparse IQ upload from edge nodes

15.2 WebSocket Endpoint

Endpoint	Description
`/ws/stream`	Multiplexed TLV stream. Clients subscribe to channels (SPECTRUM, DETECTIONS, CLASSIFICATION, AUDIO, etc.) via `SUBSCRIBE` messages. Server pushes data on subscribed channels.

15.3 WebSocket Channel Reference

ID	Channel	Payload format
0x01	SPECTRUM	Binary: 16-byte header (center_freq i64 + sample_rate i64) + u8 power bins
0x02	DETECTIONS	JSON: array of `DetectedSignal` objects
0x03	CLASSIFICATION	JSON: `SignalClassification` object
0x04	AUDIO	Binary: 8/24-byte header + i16 LE PCM samples
0x05	CONTROL	JSON: command/response
0x06	META	JSON: session metadata
0x07	SPARSE_IQ	Binary: complex64 IQ samples (edge ingest)
0x08	DEMAND_CONTROL	JSON: server-to-edge demand toggles

16 Limitations

16.1 Single-SDR Steering

Description: Only one mission at a time can steer each SDR. Concurrent missions that require different frequencies on the same device cannot both observe their target bands simultaneously.

Workaround: Deploy multiple SDRs and assign each mission a dedicated `device_name`.

Planned resolution: Wideband SDRs (AirSpy HF+ at 768 kHz bandwidth) can cover multiple signals within a single tuning. Future work will add channelised observation where the SDR stays on a wide band and individual missions filter sub-bands.

16.2 Classifier Coverage

Description: The modulation classifier supports 14 standard classes. Exotic or proprietary modulations (e.g., DMR Tier III trunking, custom OFDM) are classified as the nearest standard modulation, often with low confidence.

Workaround: Use the synthetic training lab to generate training data for new modulation types and retrain the model.

Planned resolution: An "unknown" class with anomaly detection is under investigation.

16.3 NL Query Accuracy

Description: The text-to-SQL RAG pipeline uses a 1B-parameter local LLM. Complex queries (joins across multiple tables, time-range arithmetic) fail approximately 20% of the time.

Workaround: Rephrase the question in simpler terms, or use the `/api/census/deep` REST endpoint with explicit filters.

Planned resolution: Upgrade to a larger local model (7B+) when Apple Silicon memory permits, or add a query-validation step.

16.4 Whisper Accuracy on Noisy RF Audio

Description: mlx-whisper "tiny" struggles with weak, faded, or heavily-interfered voice signals. Transcription error rates on aviation AM (high noise, clipped audio) can exceed 30%.

Workaround: Switch to the "small" or "medium" Whisper model for critical monitoring tasks.

Planned resolution: Fine-tune Whisper on RF-degraded audio samples.

16.5 Edge Node Latency

Description: Edge nodes (Raspberry Pi) introduce 200–500 ms of additional latency for spectrum data due to network transport and the sparse IQ batching interval.

Workaround: Run the full backend on the edge node if the hardware supports it (Pi 5 with 8 GB RAM).

Planned resolution: No current plan. The latency is acceptable for the target use case (unattended monitoring, not real-time interception).

17 Future Work

- **Wideband channeliser**: simultaneous observation of multiple sub-bands within a single SDR tuning, enabling concurrent missions on one device.
- **Anomaly detection**: an "unknown modulation" class that flags signals not matching any known pattern.
- **ONNX-on-device for edge**: run the modulation classifier directly on Raspberry Pi, reducing central server load.
- **Geolocation**: expand TDOA from proof-of-concept to operational capability with calibrated Kraken arrays.
- **RF denoising**: deploy the experimental U-Net autoencoder for pre-classification noise reduction.
- **Signal separation**: use the experimental FastICA module to separate overlapping emissions before classification.

A DuckDB Schema Reference

Listing 8. Complete Deep Intelligence schema

```
-- Signal hits: every persisted detection
CREATE TABLE signal_hits (
    id              VARCHAR PRIMARY KEY,
    timestamp_ms    BIGINT NOT NULL,
    center_freq_hz  DOUBLE NOT NULL,
    sample_rate_hz  DOUBLE NOT NULL,
    freq_start_hz   DOUBLE NOT NULL,
    freq_end_hz     DOUBLE NOT NULL,
    bandwidth_hz    DOUBLE NOT NULL,
    peak_db         DOUBLE NOT NULL,
    mean_power_db   DOUBLE NOT NULL,
    device_name     VARCHAR NOT NULL,
    emitter_id      VARCHAR,
    label           VARCHAR,
    confidence      DOUBLE,
    artifact_path   VARCHAR
);

-- Emitters: unique hardware fingerprints
CREATE TABLE emitters (
    id                  VARCHAR PRIMARY KEY,
    fingerprint_vector  FLOAT[],
    label               VARCHAR,
    last_seen           TIMESTAMP
);

-- Decoded content: protocol payloads linked to hits
CREATE TABLE decoded_content (
    id       VARCHAR PRIMARY KEY,
    hit_id   VARCHAR NOT NULL,
    protocol VARCHAR NOT NULL,
    payload  JSON
);
```

B TLV Protocol Quick Reference

Byte offset	Size	Field	Description
0	1	`channel`	Channel type (see §15)
1	1	`flags`	Bit 0: COMPRESSED; bits 1–7 reserved
2	2	`seq`	Sequence number (LE u16), wraps at 65535
4	4	`length`	Payload length in bytes (LE u32)
8	*N*	`payload`	Channel-specific data

B.1 Spectrum Payload Header (16 bytes)

Byte offset	Size	Field	Description
0	8	`center_freq`	Centre frequency in Hz (LE i64)
8	8	`sample_rate`	Sample rate in Hz (LE i64)
16	$N-16$	`bins`	Power values as u8 (0–255 mapped to dB range)

C API Endpoint Quick Reference

Endpoint	Method	Auth	Returns
`/api/health/deep`	GET	None	JSON: system health
`/api/census/deep`	GET	None	JSON: signal hit array
`/api/archive/stats`	GET	None	JSON: modulation rarity index
`/api/emitters`	GET	None	JSON: emitter array
`/api/briefing`	GET	None	JSON: narrated summary
`/api/nlq`	POST	None	SSE: streaming prose answer
`/api/tdoa`	POST	None	JSON: geolocation result
`/api/edge/register`	POST	None	JSON: registration ack
`/api/edge/sparse-ingest`	POST	None	JSON: ingest ack
`/ws/stream`	WS	None	TLV binary frames