Deepwave

Deploying GNU Radio in Agentic AI Systems

2026 GNU Radio Conference

Daniel Bryant

Lead Engineer

deepwave.ai

The task is the interface

“Summarize the speech on this radio channel.”

The agent selects and configures a workflow.

“What kind of task is this? What kind of radio modulation is this? Which sensor should I use and how do I set it up? Where should I send the results?”

From intent to execution

01 **Interpret the requested task**

The user describes the desired result in natural language.

02 **Select and configure a supported workflow**

The agent works through the exposed MCP interface.

03 **Run the RF AI workflow on a remote sensor**

The agent selects a sensor, configures it, and starts data flowing.

04 **Return results to the central platform**

The application receives the workflow output for reporting or further analysis.

Two ways to use GNU Radio

AI IN A PYTHON BLOCK

GNU Radio hosts the application

The flowgraph contains DSP and a Python block that invokes the AI model.

Agents can configure anything that GNU Radio exposes.

GNU RADIO IN AN INFERENCE WORKFLOW

Inference server hosts the workflow

A Python backend model wraps the flowgraph.

Ensembles connect multiple DSP and AI models and provide structured configuration.

Example: Speech inference inside a GNU Radio block

01 **Demodulate the RF stream**

Soapy source, low-pass filter, power squelch, and NBFM receiver

02 **Buffer 16 kHz audio**

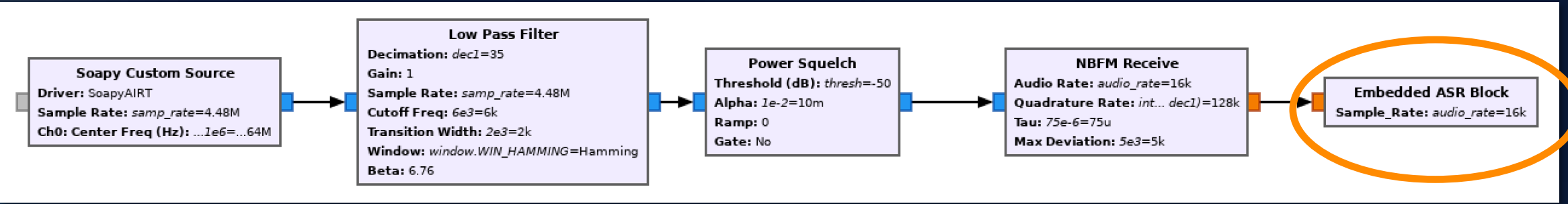
An embedded Python sink accepts float32 samples and fills a model frame.

03 **Transcribe from `work()`**

The block calls `ASRModel.transcribe()` when the frame is full.

The inference call runs inside the block's `work()` callback.

GNU Radio Hosted Implementation



ASR Block initialization

```python
import numpy as np
from gnuradio import gr
from radio_asr.online import SpeechInference


class blk(gr.sync_block):  # sinks are derived from sync blocks

    def __init__(self, sample_rate=16e3):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Embedded ASR Block',  # will show up in GRC
            in_sig=[np.float32],  # input is a float32 stream
            out_sig=[]  # no output port, this is a sink
        )
        # Allow user of block to pass in the sample rate (since this is set in
        # the flowgraph itself).
        self.sample_rate = sample_rate
        self.engine = SpeechInference(sample_rate=sample_rate)
        self.buffer = np.zeros(self.engine.n_frame_len)
        self.pos = 0
        if sample_rate != 16000.0:
            raise ValueError("Only 16kHz sample rates supported by this model")

        self.buf_duration = self.engine.n_frame_len / self.sample_rate
        self.benchmark = False  # For performance debugging

```

ASR Block work(): buffering and inference

```python
    def work(self, input_items, output_items):
        data = input_items[0]
        n_samples = len(data)
        if self.pos + n_samples < self.engine.n_frame_len: # Buffer data
            self.buffer[self.pos:self.pos+n_samples] = data
            self.pos += n_samples
            return n_samples
        else:
            n_processed = self.engine.n_frame_len - self.pos
            self.buffer[self.pos:] = data[:n_processed]
            # Measure transcription time
            t_start = time.time()
            text = self.engine.transcribe(self.buffer)
            t_end = time.time()
            t_duration = t_end - t_start
            ratio = self.buf_duration / t_duration
            text = text.strip()
            if len(text):
                self.new_line = False
                print(" " + text, end='', flush=True)
            elif not self.new_line:
                print("", end='\n', flush=True)
                self.new_line = True
            if self.benchmark:
                now = datetime.datetime.now()
                print(f"[{now}] Transcribed {self.buf_duration:.2f}sec in"
                      f"{t_duration:.2f}sec ({ratio:.2f}x): '{text}'")
            self.pos = 0
            return n_processed
```

What the GNU Radio hosted design makes difficult

Entire model runs within one monolithic block

work() runs the model synchronously

```
def work(...):
    buffer.extend(audio)
    if frame_ready:
        text = self.engine.transcribe(buffer)
    return consumed
```

Backpressure, buffering, and failure handling now live inside the flowgraph.

The flowgraph now also owns:

01 Model runtime and Python environment

02 GPU scheduling and memory pressure

03 Batching, retries, and result formatting

04 Re-use tied to one flowgraph

05 Configuration exposed to agents

The architecture works. Its ownership boundary limits composition and independent deployment.

What if a flowgraph becomes a reusable RF component?

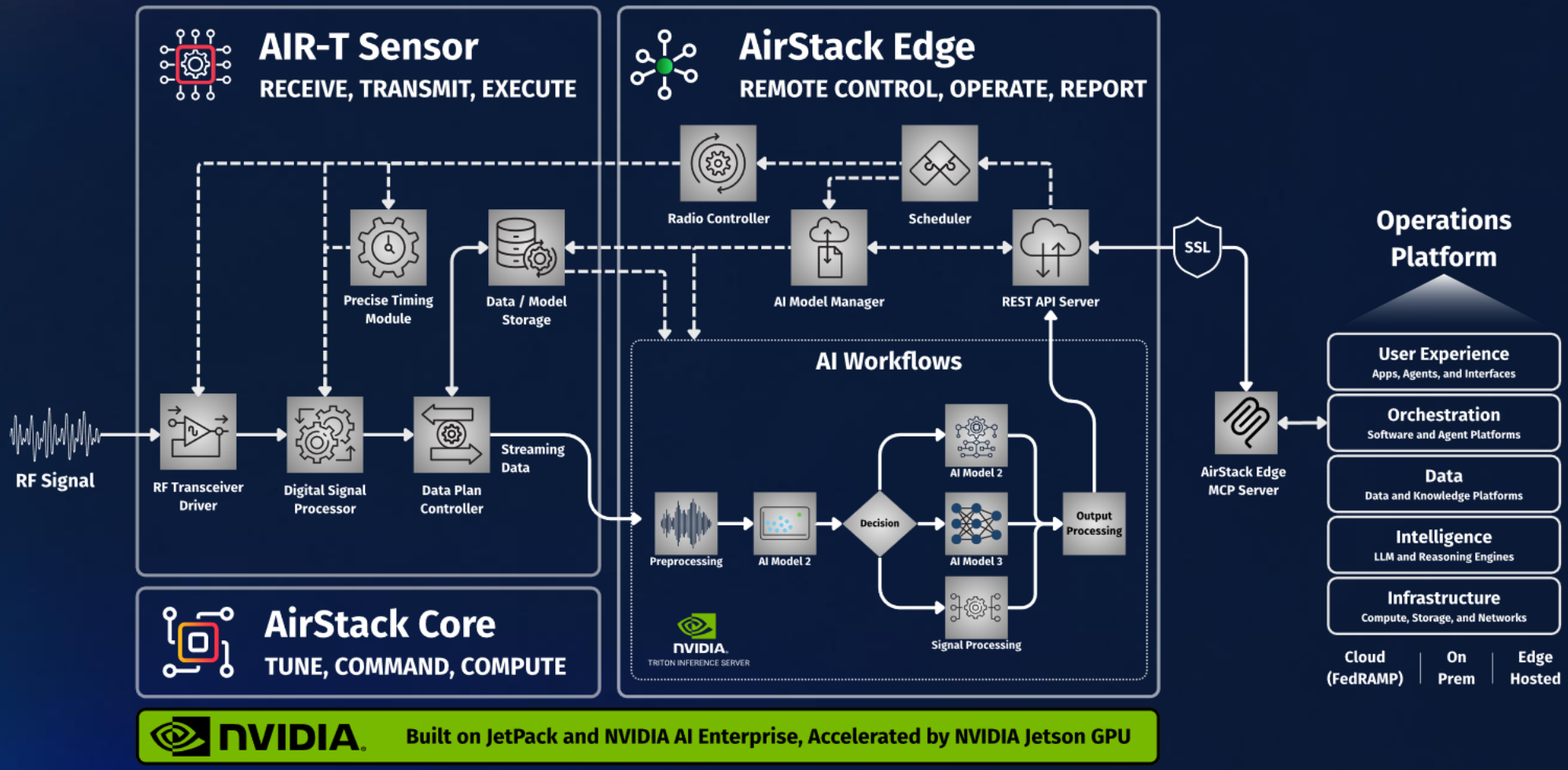
01 **Accept a tensor request**
Interleaved FP32 I/Q plus optional metadata

02 **Run the GNU Radio adapter**
Backend reuses a running flowgraph and waits for processing to settle.

03 **Return a model response**
Output and pass-through metadata feed the next model.

04 **Compose with other models**
Ensemble models connect the DSP component to AI models and downstream processing.

Deepwave's Agentic Radio AI System



GNU Radio runs in AI Workflows within AirStack Edge

Agentic Radio AI Technology Stack

Operator interface	**Radio Intelligence Agent (RIA)** Turns a natural-language task into workflow requests.
Sensor control	**MCP and AirStack Edge** Exposes what this sensor can do, manages configuration and execution.
Workflow execution	**NVIDIA Triton Inference Server** Hosts DSP and AI components and connects their tensor interfaces.
Signal processing	**GNU Radio and AI runtimes** Process RF samples, interpret signals, and produce structured results.

Inference Server: A higher-level flowgraph engine?

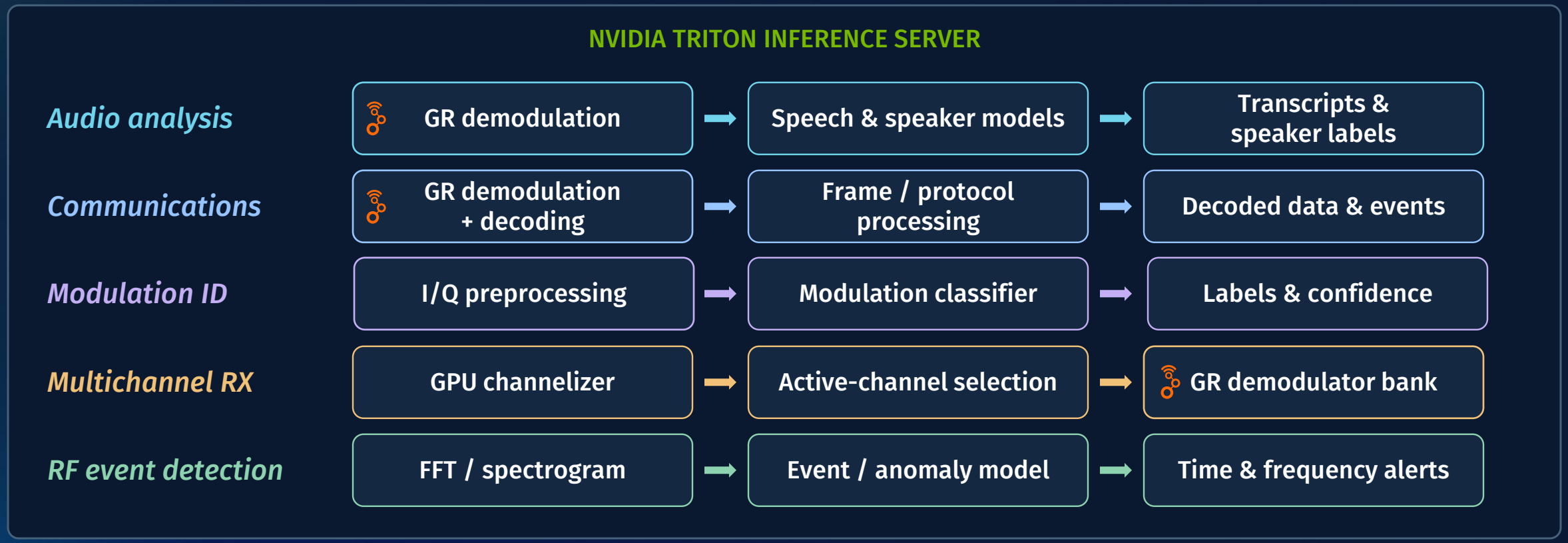
A familiar composition model, operating at a different scheduling boundary

GNU Radio	Triton	What changes
Flowgraph scheduler	Inference scheduler	Schedules requests on model instances
Flowgraph	Ensemble model	Connects models through named tensors
Block and its implementation	Model and its backend	Backend executes a configured model
Ports, sample buffers, PMTs	Named tensor interfaces	Exchanges typed arrays per request

Triton schedules model requests. GNU Radio schedules streaming work inside the DSP model.

One inference server, independent RF pipelines

DSP modules and AI models share a serving layer, with a separate processing path for each task.



Load multiple models at once and route inputs based on sensor's schedule and task.

Triton models and tensor interfaces

A named unit of computation

A model accepts named input tensors and returns named output tensors.

Its implementation can run a neural network, ordinary DSP code, or custom algorithms.

Interfaces

`config.pbtxt` defines the interface.

Each tensor's input name and shape is defined.

Each configuration parameter & value is specified.

The model runs in a single backend.

What is a Triton backend?

The execution engine behind a model: it loads model data, and handles the equivalent of work()

Backend family	Examples	What it runs
Neural network runtimes	ONNX Runtime, TensorRT, PyTorch, TensorFlow, OpenVINO	Exported models and optimized engines
LLM serving	TensorRT-LLM, vLLM	Language models and token generation
Python	TritonPythonModel in model.py	Custom Python logic, including GNU Radio
Custom C / C++	Triton Backend API	Native DSP, libraries, and custom runtimes

A Triton “model” can execute ordinary DSP code. Our flowgraph uses the Python backend.

An ensemble model connects other models

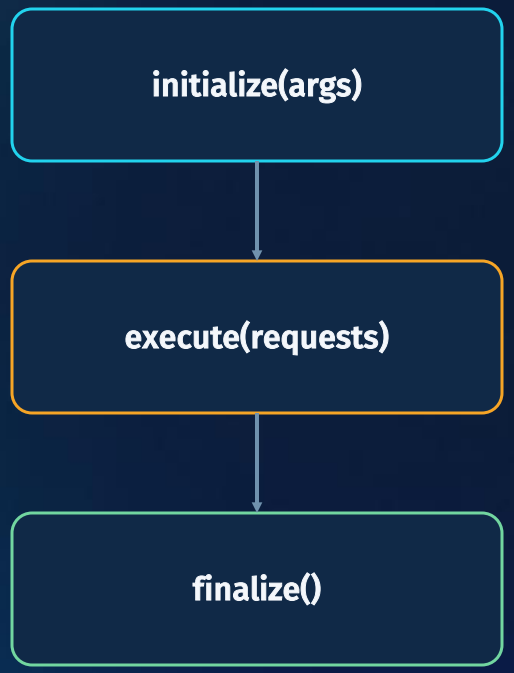
Each individual model contains explicit input/output interfaces and runs in its own backend.

Input tensor	Component model	Output tensor
I/Q samples	Demodulator (Python + GNU Radio)	Audio samples
Audio samples	Speech recognition (PyTorch)	Transcript text chunks
Transcript text chunks	Document formatter (Python)	Document JSON

This ASR ensemble contains a GNU Radio flowgraph, a PyTorch model on GPU, and custom text post-processing

One model instance owns one running flowgraph

The model lifecycle becomes the resource lifecycle for GNU Radio.



Parse config
Build flowgraph
Start the scheduler

Convert each request
Run the existing graph
Return one response

Stop and join workers
Release instance state

TritonPythonModel

```
def initialize(self, args):
    config = json.loads(args["model_config"])
    self.demod = DemodFlowgraph(config, ...)
    self.demod.start()


def execute(self, requests):
    responses = list()
    for request in requests:
        ...
        responses.append(response)
    return responses


def finalize(self):
    self.demod.stop()
    self.demod.wait()
```

Small amount of glue code adapts from Triton API to a running flowgraph

execute() adapts each request to the running graph

The existing flowgraph processes I/Q while metadata travels alongside the output

TritonPythonModel

```
def execute(self, requests):
    responses = []
    for request in requests:
        iq = triton.get_input_tensor_by_name(
            request, "IQ_IN").as_numpy()
        meta = triton.get_input_tensor_by_name(
            request, "META_IN")
        metadata = (meta.as_numpy() if meta is not None
                    else np.array([""], dtype=object))
        audio = self.demod.process(iq.view(np.complex64))
        responses.append(triton.InferenceResponse(
            output_tensors=[
                triton.Tensor("AUDIO_OUT",
                    audio.astype(np.float32)),
                triton.Tensor("META_OUT", metadata),
            ]))
    return responses
```

Read the request

Interpret interleaved FP32 I/Q as complex64. Preserve optional metadata.

Reuse the flowgraph

process() feeds samples and waits for the resulting output.

Return named outputs

One response per request, in the same order.

Custom block turns a tensor into a streaming source

Triton supplies an array. GNU Radio still uses a scheduler to pull samples.

NumpySourceBlock

```
def push(self, arr):          (called from outer TritonPythonModel)
    self.data = arr
    self.offset = 0

def general_work(self, _, outputs): (called from GR scheduler)
    out = outputs[0]
    n = min(len(out), len(self.data) - self.offset)

    end = self.offset + n
    out[:n] = self.data[self.offset:end]
    self.offset = end
    return n
```

Why a custom source + sink?

- vector_source works for a single-run flowgraph
- But starting/stopping a flowgraph is expensive compared to processing a few samples
- Custom source keeps the flowgraph alive, accepts new request payloads on demand, and lets GNU Radio consume it in chunks.
- Custom sink collects the corresponding output for each request

Why Chunked Processing Is Awkward

Flowgraph

```
def process(self, iq_data):
    self.sink.reset()
    self.src.push(iq_data)
    self.src.started_event.wait()
    self.sink.wait_until_idle()
    out = self.sink.get_data()
    stereo = out.reshape(-1, 2)
    return stereo
```

- Missing in GR 3.x: a way to know when one input chunk has fully traversed the graph
- Custom source + sink blocks created our own completion boundary
- Source signaled when general_work() injected the first sample
- Sink accumulated output and monitored arrival timing
- “Done” inferred when no new sink samples arrived for a timeout

What does deployment look like?

Models have a name, numeric version, configuration, and optional runtime environment

- This is the basic layout of any Triton model
- AirStack Edge extends this with MCP resources to describe the model to an agent

Model Tarball

```
audio_demod/
├── config.pbtxt
├── env.tar.gz
└── 0/
    ├── model.py
    └── …
```

```
name: "audio_demod"
backend: "python"

input [{ name: "IQ_IN" data_type: TYPE_FP32 dims: [-1] }]
output [{ name: "AUDIO_OUT" data_type: TYPE_FP32 dims: [-1] }]

parameters [
  { key: "sample_rate“
    value: { string_value: "1920000" } },
  { key: "EXECUTION_ENV_PATH"
    value: { string_value: "$$TRITON_MODEL_DIRECTORY/env.tar.gz" } }
]

instance_group [{ kind: KIND_CPU }]
```

AirStack Edge MCP makes models discoverable to agents

Extends Triton with model purpose, capabilities, and configuration guidance

What the model declares

Purpose

When an agent should select it

Capabilities

What it accepts and produces

Configuration

What can change, with units and limits

Illustrative example: FM demodulator

Recover audio from broadcast FM I/Q
Use this model for analog FM, not HD radio

Interleaved I/Q in, stereo audio out

Required radio settings (i.e. I/Q sample rates)
De-emphasis options and defaults

Agents use these descriptions as resources to select and configure supported workflows.

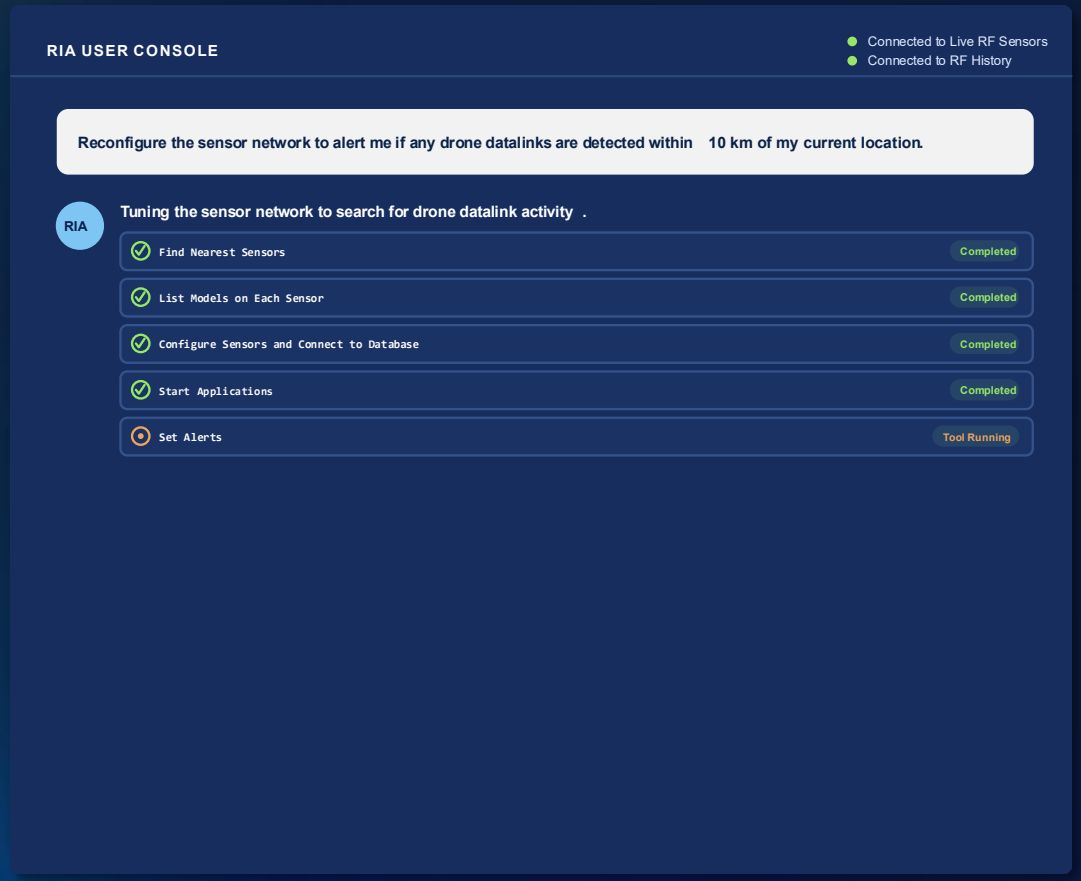
AirStack Edge MCP also exposes workflow-level control

Agent-facing interface	Implementation
Discover available SDR workflows	Each sensor provides catalog of currently deployed models
Select a workflow	Choose from the available catalog, find a suitable sensor (has model loaded, has frequency coverage, search by location)
Configure exposed settings	Apply the workflow's supported configuration
Execute and receive results	Run the underlying DSP and model pipeline

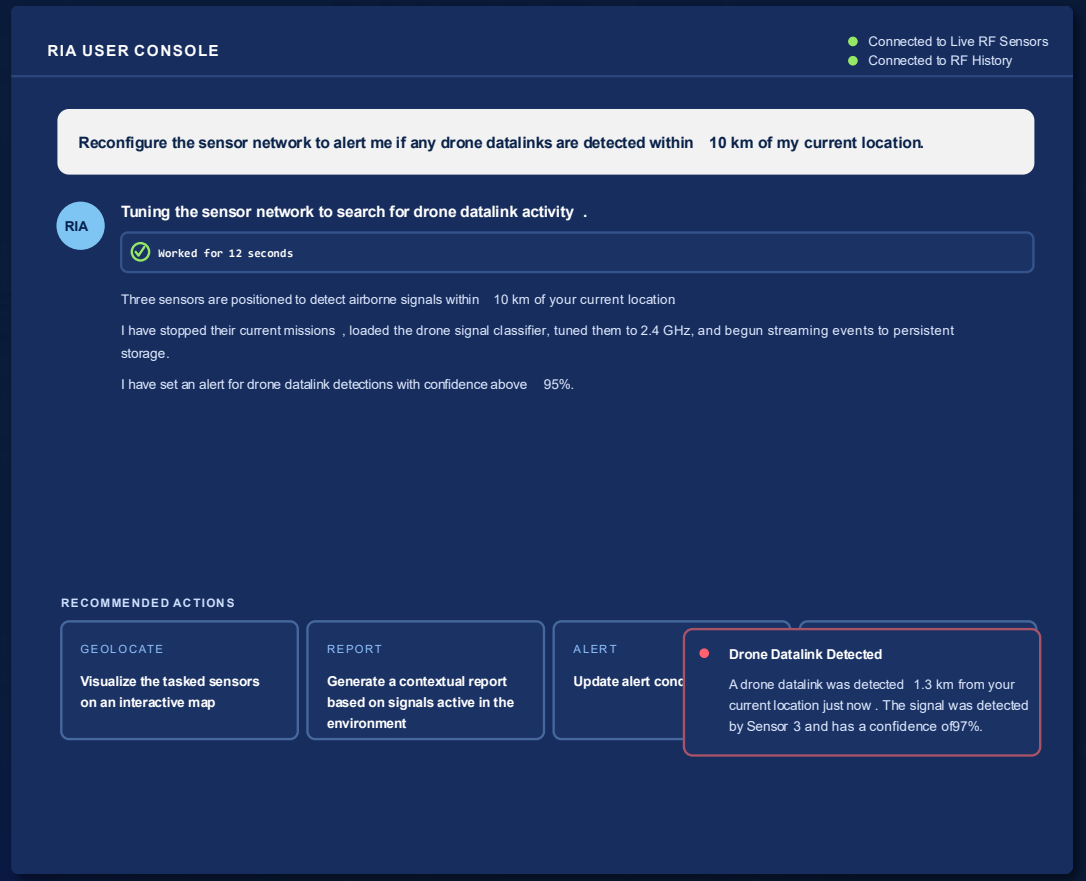
Exposing specific settings that an agent can control, without asking each agent to design a radio

Radio Intelligence Agent (RIA)

Agent Workflow and Tool Calling



Response and Alerts



GNU Radio processes RF.

Triton composes the models.

MCP lets an agent control workflows.

RIA provides the agent, memory, and analysis.

Questions?

Live demonstration

See the agent in action at the Deepwave booth

For more information

https://deepwave.ai/ria

Open-source code and models

https://github.com/deepwavedigital/airstack-edge-model-vault

DEEPWAVE

deepwave.ai