# Agentic Radiostation: An Industry Track Case Study in Autonomous, AI-Operated Radio Broadcasting with Rust

Shyamal Chandra
Crankshaft News
crankshaftnews@gmail.com

June 7, 2026

**Abstract**

Agentic Radiostation is an autonomous, AI-agent-powered radio station built as a single Rust application. It combines a typed domain model for songs, playlists, libraries, and schedules with three cooperating broadcast agents: a DJ agent that creates personality-driven voice breaks and selects tracks, a producer agent that builds day-part schedules, and a music director agent that manages listener requests and rotation. The runtime streams a continuous HTTP audio feed through Axum and Tokio, decodes local media with Symphonia, synthesizes fallback audio when source files are missing, and can call a local Ollama model for contextual DJ banter. A command-line interface, HTTP control plane, and terminal dashboard make the station operable without a separate control service.

This paper presents the system as an industry track case study. It focuses on engineering choices rather than model novelty: why the system uses Rust for agent orchestration and real-time audio, how bounded pre-buffering makes live streaming practical, where local LLM integration improves the product, and what trade-offs remain before such a system becomes production broadcast infrastructure. The case study argues that agentic media systems benefit from systems-language discipline: explicit state, typed boundaries, graceful degradation, and clear operational surfaces.

**Keywords:** autonomous agents, internet radio, Rust, audio streaming, local LLMs, media generation, systems engineering

## 1 Introduction

Radio is a durable product form because it solves a simple problem well: a listener presses play and receives a continuous, curated, human-feeling audio experience. The operational model behind that simplicity is not simple. A station needs music ingestion, programming, host identity, scheduling, metadata, listener interaction, monitoring, and reliable delivery. Traditional stations solve this with teams, studio tools, and broadcast automation. Internet radio reduced distribution cost, but it did not remove the need for constant programming labor.

Agentic Radiostation explores a more aggressive question: what does a radio station look like when the operational roles are implemented as software agents inside the streaming stack itself? The project replaces three familiar human roles with agent types:

- a **DJ** that presents the station, introduces songs, responds to listener context, and makes selection decisions;
- a **Producer** that structures the broadcast day into shows and generates playlists for each day part;
- a **Music Director** that handles requests, rotation, discovery, and mood-based recommendations.

The implementation is intentionally small enough to inspect yet broad enough to behave like a station. The Rust crate exposes modules for core domain types, agents, audio processing, HTTP streaming, command-line operation, talk-show content generation, and a live terminal dashboard. At runtime the station can generate a topic-specific synthetic library, schedule shows, select tracks, produce short spoken interstitials, stream PCM audio over HTTP, and expose control endpoints for skip, pause, resume, shuffle, playlist changes, DJ selection, show selection, metrics, and now-playing state.

This is not a paper about inventing a new large language model. It is about the systems work required to make LLM-shaped behavior useful in a media product. Recent agent research such as ReAct [7] and multi-agent frameworks such as AutoGen [8] show the value of combining language models with tool-using loops. Industry adoption, however, depends on mundane properties: state must be inspectable, failure must degrade cleanly, media timing must not fall apart, and operators need controls they can understand. Agentic Radiostation is a compact case study in those constraints.

## 2 Industry Context and Motivation

The radio industry has already adopted automation, playout systems, scheduling tools, and internet distribution. What has changed recently is the cost of generating voice, scripts, synthetic shows, and station identity. Local model runtimes such as Ollama make it practical to run LLM-assisted workflows on a developer machine or commodity server [6]. Text-to-speech engines make spoken station breaks cheap. HTTP streaming and commodity clients make global distribution accessible through ordinary network infrastructure.

This creates a product opening for long-tail radio:

- **Hyperlocal or topic-specific stations.** A school, community group, conference, retail environment, game server, or fan community can run a station around a narrow theme.
- **Low-labor programming.** AI-generated voice breaks and schedule construction reduce the need for continuous human operation.
- **Adaptive identity.** A station can change topic, voice, mood, and schedule without rebuilding its infrastructure.
- **Private media environments.** Internal company radio, classroom audio, or museum audio can use the same pattern without public broadcast distribution.

The hard part is not a single prompt. The hard part is turning those capabilities into a running station. Agentic Radiostation treats the broadcast as a systems problem: typed state machines, bounded queues, explicit interfaces, and simple failure modes.

# 3 Design Goals

The project was built around six practical goals.

## 3.1 Single-process operability

The core station should start with one command, bind an HTTP server, and expose a playable stream. External services may enrich the experience, but they should not be required for the basic station loop.

## 3.2 Agent roles as domain objects

The agents should not be anonymous prompt calls hidden in controller code. Each role should have explicit state and behavior. The DJ owns personality and presentation. The Producer owns schedule construction. The Music Director owns requests and rotation.

## 3.3 Graceful degradation

If real audio files are missing, the station should produce synthetic audio. If Ollama is unavailable, DJ introductions should fall back to template text. If text-to-speech fails, the track should still play. These choices make the system demoable and debuggable even when media or AI services are absent.

## 3.4 Low-latency listener experience

Live streaming cannot wait for model inference, TTS synthesis, and audio decode on the request path. The station must prepare future tracks ahead of playback and feed the HTTP stream from an already-prepared queue.

## 3.5 Ordinary web interfaces

The station should be controllable by CLI users, scripts, dashboards, and web clients. The implementation therefore exposes both command-line actions through Clap and HTTP endpoints through Axum [3].

## 3.6 Systems-language constraints

The project chooses Rust to keep concurrency, ownership, data modeling, and audio performance in one language. Rust's ownership model and type system reduce classes of memory and data-race bugs that are especially painful in streaming software [1]. Tokio provides the asynchronous runtime for the HTTP and background work [2].

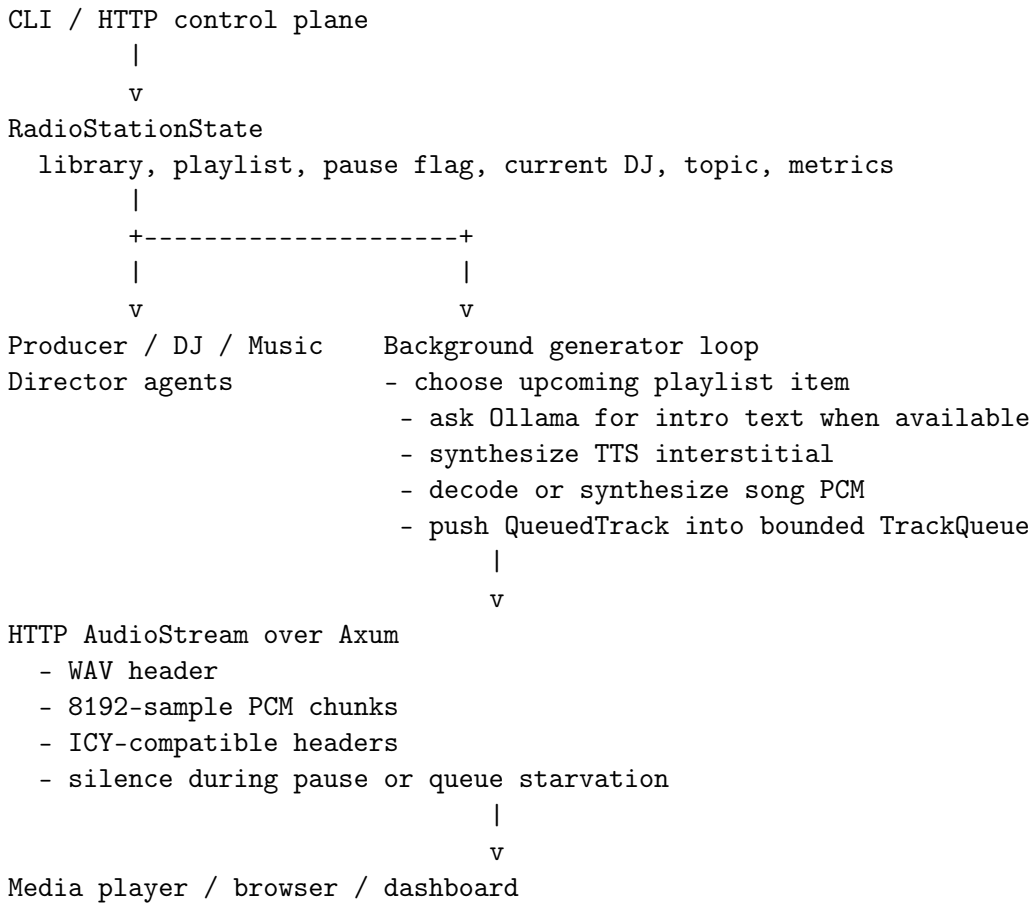


Figure 1: Runtime data flow in Agentic Radiostation.

# 4 System Overview

Figure 1 summarizes the runtime architecture. The server starts by generating or loading a music library for a topic, building a schedule, selecting the current show, creating a playlist, and spawning a background generator loop. The generator loop stays ahead of playback by preparing interstitial audio and song PCM into a bounded queue. The HTTP audio stream consumes that queue and emits chunks to clients.

## 4.1 Core domain model

The core module defines the station vocabulary. A `Song` contains title, artist, album, genre, mood tags, duration, file path, BPM, play count, skip count, energy, and release year. A `Library` indexes songs by identifier, genre, mood, and artist. A `Playlist` stores ordered entries and tracks the current index. A `Schedule` maps six day parts to shows: `EarlyMorning`, `Morning`, `Midday`, `Afternoon`, `Evening`, and `LateNight`.

The topic parameter drives both synthetic library generation and schedule construction. For example, a rock topic produces higher-energy songs and rock or metal show names; an ambient topic produces lower-energy moods and late-night ambient programming. This makes the station feel configured even when no real media library exists.

### 4.2 Control surfaces

The system exposes three operator surfaces.

| Surface | Purpose |
|---|---|
| CLI | Start the station, inspect status, list the library, list DJs, view the schedule, manage playlists, simulate listener messages, request songs, generate talk-show segments, and launch the dashboard. |
| HTTP API | Stream audio, inspect now-playing state, inspect the playlist and schedule, change shows or DJs, pause/resume, skip, shuffle, add or clear tracks, and read metrics. |
| TUI dashboard | Poll the metrics endpoint and display station status, current DJ, uptime, served bytes, bitrate, queue depth, and a waveform sparkline. |

Table 1: Operator-facing interfaces.

This shape is useful in industry settings because the station can be driven by a human, a script, a web UI, or another agent without changing the core runtime.

## 5 Agent Design

The agent design is intentionally simple. Each agent is a Rust struct with explicit fields and methods. This keeps behavior inspectable and testable. It also avoids making the LLM the owner of all business logic.

### 5.1 DJ Agent

The DJ role has two layers. `DJPersonality` stores the name, catchphrase, style, and energy level. `DJAgent` stores the selected personality, on-air state, track count, and listener greeting. The personality generator creates six DJs per topic. It chooses names, styles, energies, and catchphrases from topic-specific tables. A jazz station gets smooth, lower-energy DJs; an electronic station gets pulse, neon, cyber, bass, and synth-themed identities.

The DJ agent owns presentation methods:

- `generate_intro` creates a time-aware introduction for a song.
- `generate_outro` creates a style-specific post-song break.
- `generate_transition` adapts to energy shifts between songs.
- `respond_to_listener` generates a short listener response.
- `pick_next_song` scores candidates by style fit, energy, artist/album repetition, BPM proximity, play count, and randomness.

The important design choice is that track selection is not delegated entirely to the language model. The LLM may improve the wording of a break, but the scoring logic remains deterministic Rust code. That gives operators a place to reason about rotation policy and music fit.

### 5.2 Producer Agent

The Producer builds the station schedule and show playlists. It maps the topic to six day-part shows, assigns show names, picks genre focus, and chooses DJ names. For a given show, it filters candidates by genre and mood, estimates the target track count from average track duration, samples the pool, and sorts the result by energy. This produces a coarse radio programming pattern: mornings and prime time can be higher energy, late night can be calmer, and each topic gets a recognizable programming grid.

The Producer is the most obviously "radio" part of the system. It turns a pile of songs into an operating day.

### 5.3 Music Director Agent

The Music Director manages listener requests, rotation counts, mood recommendations, and new music discovery. Its request queue stores listener name and song identifier pairs. Its rotation map counts plays per song. It can find songs by title, recommend songs for named moods, surface underplayed music, and generate short announcements when a requested song is added.

In a production station, this role would own compliance rules, sponsor constraints, explicit-content policy, duplicate-artist spacing, request fairness, and payola/rights controls. The current implementation is small, but the role boundary is the right place for those concerns.

### 5.4 LLM integration

The optional Ollama client calls `/api/generate` on a local Ollama server, defaulting to the `llama3.2` model. It builds prompts for song introductions, outros, transitions, station IDs, and topic-aware interstitials. At startup the server attempts to connect to Ollama for up to roughly one minute. If the service is not reachable, it logs that DJ scripts will use template text instead.

This is a pragmatic LLM boundary. The system treats model output as optional content, not as required control flow. If model generation fails during pre-buffering, the generator loop falls back to a short deterministic intro. That keeps model latency and model availability from becoming station availability.

## 6 Audio and Streaming Pipeline

The audio pipeline is built around normalized PCM. The engine decodes source media with Symphonia [4], converts samples to interleaved `f32`, mixes mono to stereo when needed, resamples when asked, generates synthetic songs when files are absent, creates a WAV header, and encodes `f32` samples to signed 16-bit little-endian PCM.

### 6.1 Decoding and fallback synthesis

For real files, `AudioEngine::decode_song` uses Symphonia's probe, format, and codec APIs to decode packets into sample buffers. If the configured file path does not exist or decode fails, the station does not stop. It calls `generate_synth_song`, which creates a simple stereo composition from song metadata: genre chooses a base frequency, BPM determines beat spacing, and energy

controls amplitude. The result is not a replacement for real music, but it preserves the end-to-end radio behavior.

This fallback matters for developer experience. A new user can run the station before collecting a media library, and the streaming path can be tested without licensed audio assets.

### 6.2 Text-to-speech

DJ interstitials use the macOS `say` command to generate temporary AIFF files. The station then decodes those bytes through the same audio engine and mixes them to stereo. Talk-show generation uses Ollama for scripts, `say` for voice output, and `ffmpeg` to write MP3 files into `assets/music`. The streaming core therefore remains Rust, while some media-generation conveniences use host tools.

This is a useful industry compromise for a prototype. The OS TTS path is fast to ship and easy to inspect, but it is not cross-platform. A production version would likely use a portable TTS engine or a hosted voice service behind an adapter.

### 6.3 Pre-buffered queue

The station avoids doing slow work on the stream request path. A background `generator_loop` prepares upcoming tracks into `QueuedTrack` values containing:

- interstitial PCM;
- song PCM;
- song identifier;
- queue generation number;
- playlist index.

The `TrackQueue` is bounded, with a default maximum size of three tracks. When an operator changes the playlist, skips, shuffles, changes topic, clears the playlist, or switches show, the system clears the queue and increments a generation counter. The audio stream discards queued tracks from old generations. This small mechanism prevents stale pre-buffered audio from playing after a control action.

### 6.4 HTTP stream

The Axum route `/stream` and alias `/stream.mp3` return a response body backed by a custom `Stream`. The stream sends a WAV header once, then emits 8192-sample PCM chunks. The response includes ICY-style headers such as `icy-name`, `icy-genre`, `icy-br`, `icy-pub`, and `icy-description`, making it recognizable to many internet-radio clients and media players. The broader lineage is Icecast and Shoutcast-style HTTP audio delivery [5].

The current stream is technically WAV/PCM even though one route is named `/stream.mp3`. That compatibility alias is convenient for users, but the paper should name the trade-off directly: production software should either serve real MP3/AAC/Opus on that route or rename the route to avoid confusing clients and operators.

### 6.5 Pause and starvation behavior

When paused, the stream emits zeroed audio bytes. When the queue is empty, it also emits silence. This keeps client connections alive and avoids treating temporary starvation as a fatal stream error. The choice favors continuity over strict truth. A future implementation could expose explicit stream-state metadata so clients can distinguish silence, pause, buffering, and actual audio.

## 7 Implementation Characteristics

Table 2 lists concrete properties of the implementation. These are not benchmark results; they are static design facts observed in the code.

| Property | Implementation |
|---|---|
| Language and runtime | Rust 2024 edition with Tokio async runtime. |
| HTTP framework | Axum 0.8 with route handlers and typed shared state. |
| Audio decode | Symphonia with all codec features enabled. |
| Default output format | 44.1 kHz, stereo, 16-bit little-endian PCM in WAV container. |
| Stream chunk size | 8192 samples per emitted chunk. |
| Pre-buffer depth | Three prepared tracks in `TrackQueue`. |
| Queue invalidation | Atomic generation counter plus queue clear on control actions. |
| Metrics | Bytes served, chunks served, bitrate estimate, uptime, and waveform peaks. |
| Dashboard | Ratatui and Crossterm terminal UI polling `/metrics`. |
| LLM integration | Local Ollama HTTP client with template fallback. |
| TTS | macOS `say` for interstitials; `ffmpeg` for generated MP3 shows. |

Table 2: Implementation characteristics of the current system.

### 7.1 Concurrency model

The server uses asynchronous tasks for the HTTP server, the generator loop, and the interactive console. Shared station state is stored in `Arc` and `Mutex` wrappers, with atomic flags and counters for pause state, metrics, and queue generation. This is simple and readable. It also comes with the usual caveat: synchronous mutexes inside async programs can block executor threads if held across slow work. The current code keeps locks small in most paths, but a production refactor could move to `tokio::sync` primitives or actor-style ownership for playlist, library, and stream state.

### 7.2 State invalidation

The queue generation counter is one of the most important reliability details. Without it, an operator action could update the playlist while old prepared tracks remain in the queue. The next listener-visible item might then contradict the now-playing state or selected show. Incrementing

generation on mutation turns a broad class of stale-work bugs into a local check in the stream consumer.

### 7.3 Observability

The metrics module records bytes served, chunks served, instantaneous waveform peaks, uptime, and an estimated bitrate. The dashboard consumes those metrics over HTTP and renders a terminal view. For production, this should be extended to structured logs, Prometheus/OpenTelemetry metrics, listener counts, per-client disconnects, generator latency, TTS failures, LLM latency, queue starvation counts, and content-policy events. Still, the current metrics are enough to validate that audio is moving and that the queue is alive.

## 8 Operational Walkthrough

The station can be operated from the command line:

```
cargo run -- start --topic "late night electronic"
cargo run -- djs
cargo run -- schedule
cargo run -- library --genre Electronic
cargo run -- control now-playing
cargo run -- dashboard --server http://127.0.0.1:8000
```

Once the server is running, a listener can open:

```
http://127.0.0.1:8000/stream
```

or inspect JSON state:

```
GET  /status
GET  /now-playing
GET  /schedule
GET  /playlist
GET  /dj
GET  /metrics
POST /control/skip
POST /control/pause
POST /control/resume
POST /control/shuffle
POST /playlist/add
POST /playlist/clear
POST /show
POST /dj
```

This split is useful because broadcast products often have multiple operators: humans at a terminal, web dashboards, automation scripts, and monitoring agents. A small HTTP control plane makes those integrations possible without coupling them to Rust internals.

# 9 Evaluation

This paper does not claim a production benchmark. Instead, it evaluates the prototype against the design goals from Section 3.

## 9.1 Functional completeness

The implementation covers the minimum viable loop for an autonomous station: library generation or loading, schedule generation, DJ identity, AI or template interstitials, audio decode or synthetic fallback, continuous HTTP stream, operator controls, metrics, and dashboard display. The loop can run without Ollama and without real MP3 files, which is important for onboarding.

## 9.2 Latency containment

The most important latency decision is moving LLM generation, TTS, and decode into the background generator loop. Stream clients consume already-prepared PCM. This does not eliminate latency; it moves latency to queue fill. If model or TTS latency exceeds playback headroom, the listener receives silence. That is a better failure mode than blocking the HTTP stream, but production readiness requires measuring queue starvation and generator latency under load.

## 9.3 Modifiability

The code has clear seams for station behavior: topic inference in the library and schedule modules, presentation in the DJ agent, playlist construction in the Producer, request policy in the Music Director, and stream mechanics in the streaming module. This is a strong property for a prototype because most future work can be attached to a natural role.

## 9.4 Product realism

The station feels like a product because it includes operator controls and a dashboard, not only a generator script. However, it still lacks several features that industry operators expect: authentication, persistent configuration, content rights management, real listener accounting, stream codec negotiation, metadata intervals, observability export, and cross-platform TTS.

# 10 Lessons for Building Agentic Media Systems

## 10.1 Keep model calls outside the critical path

Model calls are variable-latency dependencies. If they sit on the listener's request path, the product inherits their worst-case behavior. Agentic Radiostation keeps model calls in the pre-buffer path and provides deterministic fallback text. This pattern generalizes to podcasts, live news, educational audio, and personalized briefings.

### 10.2 Use agents as ownership boundaries

The DJ, Producer, and Music Director are not just personas. They are ownership boundaries. This makes it easier to ask where a change belongs. Track scoring belongs to the DJ. Day-part programming belongs to the Producer. Rotation and requests belong to the Music Director. In production, this same boundary can hold policy, metrics, and tests.

### 10.3 Fallback content is a developer feature

Synthetic audio is not the final product experience. It is a developer accelerator. It makes the end-to-end stream testable without media files. This is especially useful in open-source and demo settings where copyrighted audio cannot be checked into the repository.

### 10.4 Typed systems help agent products

LLM applications often start as scripts. Scripts are good for learning, but media systems have long-lived state, concurrent tasks, binary data, and external clients. Rust's type system and ownership model are a good fit for the parts of agentic products that must be deterministic and safe [1]. The language does not replace model quality, but it can make the non-model system more reliable.

### 10.5 Name the mismatch between routes and media

The prototype exposes `/stream.mp3` while serving WAV/PCM. That may help some users guess the right endpoint, but it is operationally misleading. This is a small example of a broader industry lesson: AI products should be especially clear about the difference between product affordances and technical truth.

## 11 Security, Safety, and Rights Considerations

An autonomous radio station touches several sensitive areas.

### 11.1 Content rights

The project can generate synthetic placeholder songs and talk-show segments, but real radio operation requires rights to music, voices, and generated content. Production use should include asset provenance, license metadata, usage limits, and audit logs.

### 11.2 Prompt and output policy

If the station uses an LLM for DJ banter or talk-show scripts, it needs content policy checks. A local model can produce off-brand or unsafe output. The current prototype constrains prompts to short, conversational sentences, but it does not implement moderation, blocklists, or post-generation review.

### 11.3 Control-plane access

The HTTP control plane can pause, skip, shuffle, clear playlists, change DJs, and change shows. It is currently designed for local operation, not hostile networks. Public deployment requires authentication, authorization, request validation, rate limiting, and probably a separate listener-only stream origin.

### 11.4 Supply chain and host tools

The Rust dependency tree, Ollama runtime, `say`, and `ffmpeg` are part of the operational supply chain. Production packaging should pin versions, declare platform requirements, and separate optional media-generation tools from the streaming runtime.

## 12 Limitations and Future Work

The current system is a strong prototype, not a finished broadcaster.

- **Codec mismatch.** The stream emits WAV/PCM, including on the `/stream.mp3` route. Production should add MP3, AAC, or Opus encoding.
- **No persistent station state.** Playlists, requests, rotations, and schedules are in memory.
- **No listener accounting.** The `/listeners` endpoint is simulated.
- **Limited stream metadata.** ICY headers are present, but timed ICY metadata blocks are not.
- **Cross-platform TTS gap.** The interstitial path depends on macOS `say`; talk-show generation also expects `ffmpeg`.
- **Simple resampling.** The audio engine includes nearest-neighbor resampling, which is not production audio quality.
- **No moderation pipeline.** LLM-generated scripts are not checked before playback.
- **No auth on controls.** The control plane assumes trusted local access.
- **No formal load testing.** The project needs benchmarks for concurrent listeners, queue starvation, model latency, TTS latency, and memory usage.

Future work should focus on four tracks. First, split the audio backend into codec adapters and add real internet-radio formats. Second, add persistent station state with a migration-friendly storage layer. Third, add policy: rights metadata, moderation, auth, and audit logs. Fourth, add a web control surface over the existing HTTP API so non-technical operators can run the station.

## 13 Related Work

Agentic Radiostation sits at the intersection of internet radio, local AI inference, and systems programming.

Icecast documents the long-running model of HTTP-based internet broadcasting and source/client separation [5]. The project borrows the familiar listener experience of opening a stream URL, but implements a compact in-process stream generator rather than a full relay server.

Rust, Tokio, and Axum provide the systems foundation [1, 2, 3]. Symphonia provides pure-Rust media decoding [4]. This matters because agentic systems often begin in Python, while media streaming benefits from predictable memory behavior, compiled binaries, and strong concurrency rules.

On the AI side, ReAct shows the usefulness of interleaving reasoning and acting with external tools [7], while AutoGen explores multi-agent conversation patterns [8]. Agentic Radiostation applies a related idea to a media product, but uses smaller domain-specific agents instead of a general conversation framework. The agents communicate through typed station state, not free-form chat.

The end-to-end argument in system design remains relevant [9]. Some guarantees, such as "the listener hears the selected show after a skip," cannot be solved solely by a network stream, an LLM prompt, or an audio decoder. They require state invalidation across the whole station. The generation counter in `TrackQueue` is a small example of that principle.

## 14 Conclusion

Agentic Radiostation shows that an autonomous radio station can be built as a compact Rust system rather than a pile of scripts around a model endpoint. Its most important contribution is architectural: agent roles are explicit, model calls are optional and off the critical path, audio work is normalized through a single engine, and operator controls are exposed through ordinary CLI and HTTP interfaces.

The prototype is honest about its gaps. It needs production codecs, persistence, authentication, moderation, observability, cross-platform TTS, and load testing. But the core pattern is strong: treat AI-generated media as one component in a typed, observable, degradable streaming system. That pattern can extend beyond music radio to personalized news, educational audio, internal company radio, game-world broadcasters, event channels, and other continuous media products.

## References


[1] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2019. Online edition: https://doc.rust-lang.org/book/.

[2] Tokio Contributors. *Tokio: An asynchronous Rust runtime*. https://tokio.rs/.

[3] Tokio-rs Project. *Axum web application framework*. https://docs.rs/axum/latest/axum/.

[4] Pdeljanov and contributors. *Symphonia: Pure Rust audio decoding and demuxing*. https://github.com/pdeljanov/Symphonia.

[5] Xiph.Org Foundation. *Icecast Documentation*. https://icecast.org/docs/.

[6] Ollama. *Ollama: Get up and running with large language models locally*. https://ollama.com/.

[7] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations*, 2023. https://arxiv.org/abs/2210.03629.

[8] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework. arXiv:2308.08155, 2023. https://arxiv.org/abs/2308.08155.

[9] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984. https://doi.org/10.1145/357401.357402.