ChatGPT

Local Podcast Automation Platform – Architecture & Design

Introduction

This report presents a comprehensive technical and architectural plan for a **local server-based podcast automation platform** built with open-source software. The system will allow users to **edit podcast audio via a drag-and-drop web UI**, apply common audio edits (trim, intro/outro, volume gain, pitch/time shifts), and automatically generate an **audiogram waveform video** for sharing on platforms like YouTube. It will also incorporate **AI-driven transcription and content generation** (using OpenAI Whisper for speech-to-text and an Ollama-managed LLM for titles/summaries) and support automated publishing (e.g. uploading to YouTube or updating an RSS feed). All components are intended to run locally in Docker containers with support for GPU acceleration (NVIDIA CUDA or AMD ROCm) where applicable. The following sections detail the key components, their integration, resource requirements, and the end-to-end architecture (with Docker Compose setup, directory structure, and workflow). The goal is to provide a blueprint for a development team to assemble and extend this system.

System Components Overview

To meet the requirements, we propose integrating several open-source tools, each fulfilling a specific role in the pipeline. Below is an overview of the major components and their purpose:

- **Web-Based Audio Editor UI (Drag-and-Drop):** We recommend using **AudioMass**, an open-source web audio editor that runs entirely in the browser. AudioMass provides a waveform timeline interface and supports operations like cutting, pasting, trimming audio, volume adjustment, fades, and even effects like reverb, compression, and pitch shift [1] [2]. It's a lightweight JavaScript app (~65 KB) that requires no backend processing (all editing is done client-side) [3]. This will serve as the user-facing **drag-and-drop editor** where creators can import `.wav` (or other) files, perform edits, and arrange their podcast episodes visually. The UI will allow adding an intro/outro by simply concatenating audio segments (e.g. the user can load an intro clip, copy and paste or drag it to the start of the main audio track, and similarly for outro at the end). AudioMass can export the edited audio (e.g. as WAV or MP3) when the user is done [4]. We will integrate it by hosting the AudioMass static files in our web container and possibly extending it to send the exported file to our backend for further processing.

- **Audio Processing Engine (Backend):** On the server side, we will use **FFmpeg** (and/or related libraries like SoX or PyDub) to handle final audio processing tasks. FFmpeg is a powerful open-source media processing tool capable of concatenating files (to join intro/main/outro), trimming or segmenting audio, adjusting volume (using filters or normalization), and even altering speed or pitch (via filters like `atempo` for tempo or `asetrate` for pitch shifting). Many of these edits can be directly done in the AudioMass UI, but using FFmpeg on the backend ensures consistent, high-quality output and offloads heavy processing from the browser. For example, after the user's editing

actions are collected (or after receiving the exported track), the backend can apply final normalization or encoding via FFmpeg CLI or a Python wrapper. FFmpeg will also be crucial for generating the waveform video (described below). **SoX** (Sound eXchange) or **librosa/pyrubberband** are alternative tools for specific effects (SoX has high-quality tempo/pitch shift algorithms), but FFmpeg alone can likely cover our needs (to avoid overlapping tools). In summary, the backend (preferably a Python service using libraries or subprocess calls) will orchestrate audio processing: merging tracks, encoding to the desired format, and preparing inputs for other stages.

- **Waveform Video Generator:** To create the **aesthetic waveform videos** for platforms like YouTube, we will leverage FFmpeg's filtering capabilities or an existing audiogram generator. An *audiogram* is a video combining audio with a waveform visualization [5]. FFmpeg has a built-in `showwaves` **filter** that can render an audio waveform as an overlay on an image or canvas. For example, using a command with `-filter_complex "[0:a]showwaves=s=1280x720:mode=line,format=rgba,colors=0x00ff00|0x0000ff[v]; [1:v][v]overlay=0:0[outv]"` would draw a multicolor waveform (green & blue in this case) over a background [6]. We can generate videos by combining the final audio with a background image (like a static podcast cover art) and overlaying the waveform animation. *Figure 1* below illustrates an example of a minimalist waveform style similar to the user's reference. FFmpeg is CPU-driven for such rendering (no specialized GPU use by default), so a **decent multi-core CPU** is recommended for faster processing [7]. For instance, rendering a 50-minute audio to video took about 25 minutes on an AMD Ryzen 5 3600 (6-core CPU) [8]. This is roughly 2× faster than real-time, so shorter episodes will render quickly. We can fine-tune waveform appearance (color, height, style) in the FFmpeg filter parameters [9]. As an alternative, the open-source **Audiogram** tool by NYPR [10] [11] could be containerized – it provides a Node.js server that takes an audio clip, a background image, and optional captions and outputs a polished video. However, using FFmpeg directly via Python gives us more control and simpler integration (avoids overlapping functionality, since Audiogram internally uses FFmpeg too). The plan is to create a Python function or script that, given an audio file and an image (and style settings), invokes FFmpeg to produce the waveform MP4. We'll ensure the output video meets YouTube specs (e.g. 1080p resolution as needed).

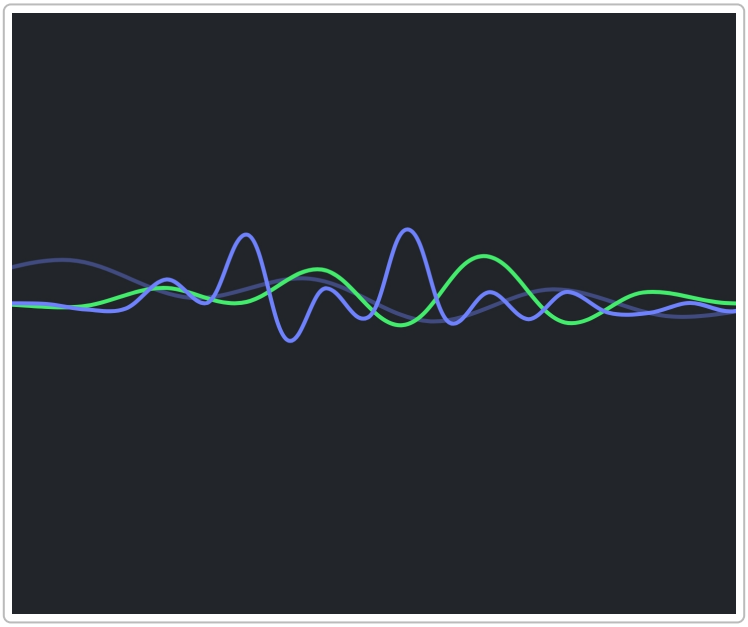

Figure 1: Example waveform visualization (stylized). The platform will generate similar waveform videos by overlaying an audio waveform on a background image.

- **Speech-to-Text Transcription (Whisper):** For automated transcription of podcast audio, we will deploy **OpenAI's Whisper** model. Whisper is an open-source, state-of-the-art ASR (Automatic Speech Recognition) model known for its accuracy on a variety of languages. We will use the Whisper **large-v2 model** for best accuracy, but also allow using smaller models for lower resource environments. Whisper can be invoked via its Python API (`openai-whisper` package) or via a CLI tool, and it supports GPU acceleration via PyTorch. On an NVIDIA GPU, Whisper large requires approximately **10 GB of VRAM** to run inference [12] (the medium model needs ~5 GB, small ~2 GB, etc. [12]). It's recommended to have at least that amount of GPU memory (or system RAM if running on CPU) to transcribe in reasonable time. The platform will attempt to use an NVIDIA GPU if available, or fall back to CPU (with a smaller model if necessary to avoid long runtimes). AMD GPU support is possible via ROCm – Whisper *does* work with ROCm on supported AMD cards [13] [14], though setup may be non-trivial; our Docker environment can include ROCm PyTorch for AMD compatibility. We might integrate **Faster-Whisper** (which uses CTranslate2) as it offers ~4× faster transcription and lower memory use for the same models [15], including 8-bit quantization for further efficiency [16]. The transcription service will likely run within the Python backend container (calling the library directly). For long episodes, transcription will be run asynchronously (so as not to block the web UI). The resulting transcript text can be stored (as a file or in a lightweight DB) for use in title/summary generation and potentially for inclusion in show notes. We will also consider using **Whisper's timestamped output** to generate subtitles if needed for the video, though that is an extension (not explicitly required, but feasible with the transcribed text).

- **Local LLM for Titles/Summaries (Ollama + model):** To generate episode titles and summaries, we will use an open-source **Large Language Model** served via **Ollama**. Ollama is a tool for running LLMs locally with ease – it acts as a lightweight server and package manager for models [17]. We will deploy Ollama in a container (there's an official image) and load a suitable model (for example, a fine-tuned Llama-2 or similar 7B-13B parameter model that's good at summarization). Ollama provides a simple REST API: once the service is running (by `ollama serve`), we can hit `http://ollama:11434/api/generate` with a JSON payload specifying the model and prompt [18]. The backend will send a prompt containing the transcript (or a condensed version if the transcript is very long) asking for a concise title and a paragraph summary. The model's output will be captured as the suggested title and summary. We will design the prompt to steer the model appropriately (e.g. "Provide a catchy title (max 10 words) and a 2-3 sentence summary of the following podcast episode transcript..."). In terms of resources, running a 7B model typically requires ~8 GB of RAM (for 4-bit quantized weights) and a modern CPU [19]; a 13B model might require ~16 GB [20]. If an NVIDIA GPU is present, Ollama can offload model inference to it (it supports 4-bit GPU acceleration, which can improve speed and reduce CPU load) [21] [22]. We will start with a 7B model to balance quality and performance. The platform should be configurable to use different models via Ollama (for instance, if the user wants to install a more capable model and has the hardware, they can do so). **Note:** Ollama itself will manage model files (which can be several GB each) – we'll mount a volume for Ollama's model storage so that downloads persist. The integration will be through HTTP calls from our backend to the Ollama service (no code overlap, since Ollama handles the LLM internally).

- **Publishing Integrations (YouTube & RSS):** Once the audio, video, and text (transcript/summary) are prepared, the platform will **publish content automatically**. We plan to use **n8n**, the open-source

workflow automation tool, to handle external integrations (described separately below). For YouTube, n8n includes a YouTube integration node that can authenticate via API and upload videos [23]. We will supply the video file, title, description (which can include the AI-generated summary and maybe a link to the transcript or website) to that node in an n8n workflow. The YouTube Data API will handle the upload to the user's channel (the user would need to provide API credentials or OAuth tokens via n8n's credential system). For podcast RSS feed publishing, there are a couple of approaches: (1) If the user self-hosts the audio and RSS, the platform can generate or update an RSS XML file that includes the new episode entry (with enclosure URL pointing to the audio file). We could use a Python library like **Feedgen** to construct the RSS feed programmatically [24]. The file can then be placed on a local or cloud server. (2) Alternatively, if the user uses an external podcast hosting, we might skip RSS generation and instead upload the audio to that service – but since the requirement explicitly mentions RSS, we'll assume self-publishing. The n8n workflow can, for example, SFTP the audio file to a web host and update an RSS file, or run a custom script (n8n has a Code node) to update the local RSS and commit it somewhere. In summary, publishing will be automated either by direct API calls (YouTube, possibly others) or by preparing files for distribution (RSS feed). These actions are orchestrated so that once the content is ready, minimal manual intervention is needed for the episode to go live.

- **Automation Orchestrator (n8n):** We incorporate **n8n** as a central workflow automation service. **n8n** is a fair-code (source-available) automation platform with a visual editor to connect various services and logic [25]. We will run n8n in a container (with its own web UI on a local port, e.g. 5678). The role of n8n in our system is to **automate multi-step tasks** and integrate with external APIs easily. For example, we will create a workflow in n8n that is triggered when a new episode is finalized – this workflow can: receive a webhook or API call from our backend containing episode data, then perform steps like calling the YouTube node to upload video, calling an HTTP Request node to update a remote RSS feed or running a small script to update a local file, and maybe even send a notification (email/Slack) that publishing is done. By using n8n, we offload the external integration logic from our core code – n8n already has 400+ integrations (including email, cloud storage, social media, etc.) [26] that the team can leverage without writing custom API calls. It also allows building complex logic with minimal code (though code nodes can be used for custom tasks as needed [27]). We will treat n8n as a microservice: the backend will trigger n8n workflows via its API (n8n provides a built-in **webhook trigger** node type, so our backend can `POST` to a specific URL to start the workflow with context data). n8n will need access to the output files (audio/video) – we can achieve this by sharing a volume between the backend and n8n containers (so n8n can read the video file from a known directory), or by passing the file data in the webhook (less ideal for large files). A shared volume approach is likely simplest for local deployment. **Security:** Because n8n has broad access and uses webhooks, we will secure those endpoints (n8n allows setting up API key or basic auth on webhooks, or we run it in trusted local network). In terms of resources, n8n is not heavy – it runs on Node.js and typically 1-2 CPU cores and ~1 GB RAM is enough for moderate workflows, though more complex flows or heavy integrations could raise that slightly.

The table below summarizes the chosen components, including where to find them, their purpose, and how we plan to integrate each into the system:

Component & Source (Link)	Purpose / Functionality	Integration Method	Notes / Alternatives
AudioMass (GitHub: [28])	In-browser waveform audio editor (cut/paste, trim, effects, etc.) [1]. Provides a drag-drop UI for building the episode timeline.	Served as static web app (HTML/JS). No backend required – edits run in client. Exported audio is uploaded to server.	Chosen for ease of use and rich features (pitch shift, EQ, etc.) [2]. Alternative: custom UI using WaveSurfer.js (would require more development).
FFmpeg (Website: ffmpeg.org)	Audio processing backend (concatenate intro/outro, re-encode audio, adjust gain) and **waveform video rendering** via filters [5] [9]. Also used to encode final media (MP4, MP3).	Installed in Python backend container. Invoked via CLI (subprocess) or via Python bindings (e.g. `ffmpeg-python`).	FFmpeg is a de-facto standard for media processing. No overlap – it's needed for tasks not handled by other libs.
OpenAI Whisper (GitHub: [29])	Speech-to-text transcription of audio. Converts podcast audio to text transcript with high accuracy.	Used as a Python library (`openai-whisper`) inside backend. Runs on GPU (CUDA or ROCm) if available, otherwise CPU.	We may use **Faster-Whisper** for better performance [15]. Models downloaded at runtime (Large ~1.5GB). No direct alternative with equal quality (Open-source) – this is the chosen ASR engine.
Ollama + LLM model (GitHub: [17])	Local LLM service for generating episode titles & summaries. Manages models (e.g. Llama 2 variants) and provides an API for text generation.	Runs as a separate container (Ollama's server). Accessed via REST HTTP calls from backend (to `ollama:11434`).	Model files (4–8+ GB) stored in volume. Ensures no external API needed for AI. Alternatives: direct use of `llama.cpp` or HuggingFace in Python, but Ollama offers simplicity and an OpenAI-compatible API.
n8n (Workflow Automation) (GitHub: [25])	Orchestrates automation: receives trigger from backend and handles publishing (YouTube upload, etc.) using built-in nodes. Also can schedule or chain other tasks (e.g. nightly batch processes if needed).	Runs as separate service (Docker). Workflows configured via n8n's UI; triggered by backend via webhook calls. Communicates with external APIs through its nodes.	Chosen for flexibility (400+ integrations) and visual workflow editing [26]. Alternatives: custom Python scripts or other automation tools (Node-RED, Airflow), but n8n is purpose-built for this use case.

Component & Source (Link)	Purpose / Functionality	Integration Method	Notes / Alternatives
YouTube API (Google API)	External service – used to publish the video on YouTube. Allows programmatic upload of videos to YouTube channels.	Accessed via n8n's YouTube node (pre-built integration) [23] or via HTTP Request node calling YouTube Data API. Requires API credentials.	YouTube Data API v3 is the underlying interface. n8n simplifies usage with an authenticated node.
Podcast RSS Feed (XML)	Enables distribution to podcast clients by syndicating episodes. The system will generate or update an RSS feed file including the new episode (title, description, media URL, etc.).	Handled via a custom script or n8n workflow. For example, use Python's **Feedgen** library to create the RSS XML [24], then save to a file or upload to hosting.	If the local server is accessible online, it could directly serve the RSS. Otherwise, integration with a hosting service or cloud storage (via n8n) may be configured.

Architecture and Workflow

The platform is designed as a set of cooperative services orchestrated via Docker Compose. *Figure 2* shows the high-level architecture and data flow between components. The design emphasizes a modular pipeline: the user interacts only with the Web UI, and all heavy processing/transfers happen on the server side in stages. Below is a description of the end-to-end flow:

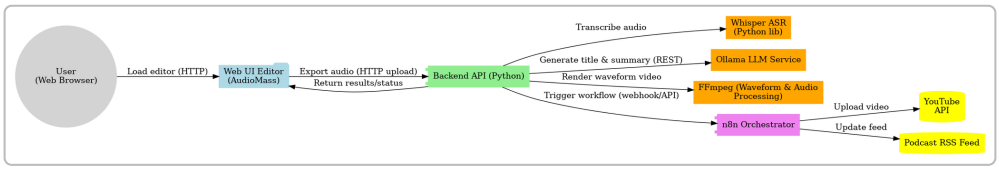

Figure 2: End-to-end architecture of the podcast automation platform. The user edits audio in the browser, and the backend coordinates transcription, AI summary, waveform video rendering, and publishing via n8n. Arrows indicate data or API flow between components.

Step 1 – Editing via Web UI: The user accesses the **Web UI Editor** (AudioMass) through their browser. The UI could be served at `http://localhost` (perhaps via an Nginx or directly by the backend server). The user imports their raw recordings (e.g. a WAV file of the main content) by dragging into the waveform editor. They can similarly import an intro/outro clip. Using the editor's timeline, the user trims unwanted parts, arranges the intro -> main -> outro sequence (for example, by copy-pasting or simply opening the main file and appending the intro at start, etc.), and applies any needed effects (volume normalization, fades, noise reduction, etc., as offered by the tool). They can play back the preview in the browser to verify the edits. The editing process is entirely client-side – no server resources are used yet, ensuring low latency for the user.

Step 2 – Exporting Edited Audio: Once satisfied, the user triggers an **export** from the web editor (e.g. clicking "Export" -> "Download/Send"). AudioMass supports exporting to MP3 (with a LAME WASM encoder) or to WAV [4]. In our integration, instead of (or in addition to) downloading the file to the user's machine, we will **upload the edited audio file to the backend**. This can be done by an AJAX call or form upload that the AudioMass UI triggers upon export (we might customize the AudioMass code to POST the audio blob to an endpoint). The backend provides an endpoint (e.g. `/upload`) to receive the final edited audio file (in WAV or high quality format to avoid generational loss). Along with the audio, metadata like the project name or desired episode title (if user provided one) could be sent.

Step 3 – Backend Processing: The **Python backend service** (likely a Flask or FastAPI app) receives the audio file. Now it orchestrates several tasks in sequence or in parallel:

- **(a) Audio Post-processing:** If any additional processing is needed server-side (for example, ensuring the intro and outro are properly attached if the user provided them separately, or applying final normalization), the backend will invoke FFmpeg. In many cases, if the user already combined the intro/outro in the editor, no further merging is needed. But suppose the UI only allowed editing the main audio and the intro was not appended; the backend could concatenate the intro + main + outro using an FFmpeg concat command or filter. Similarly, it might apply a loudness normalization filter (if we want to match broadcast LUFS standards) or ensure the output is in a target format (say MP3 at 128 kbps for RSS, and AAC in MP4 for video). These operations are scripted with FFmpeg's command-line and executed, producing a final mastered audio file (e.g. `episode-final.mp3`).

- **(b) Transcription (Whisper):** In parallel with (a) (or right after, depending on desired sequencing), the backend calls Whisper to transcribe the audio. Using the Python library, it will load the model (this might take a short time if not already loaded – we could keep the model in memory between runs to optimize) and run the transcription. If GPU is available, this step is quite fast (e.g. a few minutes for an hour of audio on a high-end GPU) [30]. If only CPU is available, transcription will be slower (possibly real-time or 2× real-time for large model on a 6-core CPU). We could choose a smaller model in that case to speed it up at the expense of some accuracy. The output will be a text transcript of the entire episode. We might save this to a file like `episode-transcript.txt` for reference.

- **(c) LLM Title/Summary Generation:** As soon as the transcript text is ready (or even as it's streaming, but we'll do it after full transcript for simplicity), the backend prepares a prompt and calls the **Ollama API** to generate the episode title and summary. The call is made via an HTTP POST to the Ollama server (running at `http://ollama:11434`). Ollama will load the chosen model (if not already in memory) and return the completion result. For example, the prompt might be: "`TRANSCRIPT: [full transcript text]`. *Now generate a catchy title (in 5-10 words) for this podcast episode and a 3-sentence summary highlighting the main points discussed.*" The model's response is parsed to extract a title and summary. These might be further trimmed or moderate (we can enforce character limits or remove any problematic content as needed). The result is stored, e.g. as `episode-title.txt` and `episode-summary.txt`.

- **(d) Waveform Video Rendering:** In addition to the audio file, we generate the video for YouTube. The backend calls an FFmpeg command (or uses a small Python automation) to produce the waveform visualization MP4. This requires a background image – we will either have a default template image (perhaps a stylized background with the podcast logo and an empty space where

the waveform will go) or allow the user to upload one. (The example in *Figure 1* simply had a dark background with a two-tone waveform; we might use something similar or more branded.) The command will take the final audio (`episode-final.mp3`) and the image (e.g. `background.png`) and output a video file `episode-video.mp4`. Using FFmpeg's `showwaves` filter we can customize color and style; e.g. a continuous line or mirrored waveform. We can match the waveform color scheme to the user's preference (perhaps even use two colors to distinguish left/right channel as in the example). This process is CPU-intensive but straightforward – FFmpeg will utilize multiple threads. We expect the video rendering to be one of the longer steps, but since it's happening on the server, the user can do other things meanwhile. (If performance is a concern, we could consider offloading this to a separate worker process or container to run in parallel with other steps, and/or notify the user when done.)

All these sub-tasks (a–d) can be initiated by the backend almost simultaneously after receiving the audio. We need to manage their outputs and ensure any dependencies: for instance, the transcription could use either the raw edited audio or the finalized audio (with intro/outro). Ideally we transcribe the entire final audio so the transcript covers the intro/outro as well (though those might just be music – we could skip transcribing the known music sections to save time). We will implement this logic in Python, possibly with asynchronous tasks or threads. If needed, a task queue (e.g. Celery with Redis) could be introduced, but given the scale (one job at a time, mostly), a simpler thread or sequential process is fine. The system will log progress so the user can be informed in the UI (e.g. showing "Transcribing…", "Generating summary…", etc.). The backend might expose an endpoint the UI can poll for status, or use WebSocket to push updates if we want a real-time update on progress.

Step 4 – Publishing Automation (n8n): Once the content is prepared (audio, video, title, summary, transcript), the final step is to publish. This is where **n8n** comes into play. The backend will trigger an **n8n workflow** dedicated to publishing the episode. We will create a workflow in n8n with a **Webhook Trigger** node, and our backend will `POST` to that webhook URL (which might look like `http://n8n:5678/webhook/podcast-publish`). The payload can include information like the file paths for the audio and video, the title and summary text, etc. Upon triggering, the n8n workflow will execute a series of nodes:

- **YouTube Upload:** Using the YouTube integration node in n8n, configured with the user's channel credentials, it will take the `episode-video.mp4` file and upload it to YouTube. The node will be set with the **Title** (from LLM output) and **Description** (perhaps the summary plus any standard info/links). If the podcast has an associated thumbnail, we could either include it in the video or upload a separate thumbnail via another node (YouTube API allows custom thumbnails). The successful upload returns a video ID or URL, which we can capture for record.

- **RSS Feed Update:** Next, the workflow updates the podcast RSS feed. If the RSS is hosted on the local server, this could be as simple as running a Code node that uses the Feedgen library: it might read an existing `podcast.xml`, insert a new `<item>` entry for the new episode (with title, description, pubDate, etc., and an `<enclosure>` with url pointing to the audio file and length/type), then save the file. If the RSS needs to be published externally, the workflow could SFTP the new audio (`episode-final.mp3`) to a static hosting and then update the XML there. Another approach is using a **GitHub integration** in n8n: for example, if the user hosts their site via GitHub Pages, n8n could commit the new audio and XML to a repository. The exact method can vary, but the key is the process is automated.

- **Notifications (optional):** We can include optional nodes, like an Email node to send the user a confirmation that "Episode X has been published to YouTube and RSS feed." Or a message to a Slack/Discord webhook if the user wants team notifications. n8n makes this straightforward if needed.

The n8n workflow runs these in order. Because n8n runs in a container on the same Docker network, it can access files via a shared volume mount. For instance, we mount a host directory (say `./output`) into both the backend container (where files are created) and the n8n container. The backend might pass just the filename, and the n8n YouTube node will look for that file in the shared path to upload. This avoids transferring the file over HTTP again. Once completed, n8n will respond to the webhook call (which our backend receives) with a success status. Our backend can then inform the UI that publishing is done.

Step 5 – User Feedback and Extendability: After the automation completes, the user could be presented with the results. The UI could display the generated title & summary for review (perhaps before publishing, the user might want to tweak the title or description – this could be a step we allow: e.g. show AI suggestions and let user override before hitting "Publish"). The transcript could also be offered as downloadable text or used for adding subtitles to YouTube (YouTube API allows uploading subtitles too, though not in our core scope). The system is designed such that each component can be replaced or upgraded: e.g. if a better ASR model comes out, it could replace Whisper; if the user prefers a cloud AI service, that could be hooked into n8n instead. The **modular architecture** ensures these substitutions don't break the overall pipeline as long as interfaces are respected.

In summary, the runtime flow is: **User edits audio -> Backend receives audio -> (FFmpeg merges intro/outro) -> Whisper transcribes -> Ollama generates text -> FFmpeg renders video -> n8n publishes to external services.** All of this happens on the user's local machine or server, preserving privacy and control. The use of Docker Compose means each service is isolated but networked, communicating over well-defined APIs or shared files.

Deployment Architecture (Docker & Directory Structure)

We will use **Docker Compose** to tie all services together, ensuring one-command setup and consistent runtime environment. The Compose file will define the following containers (services):

- `web` – Serves the frontend UI. This could be as simple as an Nginx container serving the AudioMass `index.html` and assets, or we could let the Python backend serve static files. For modularity, we'll assume a small Nginx that serves the UI on port 80 and proxies API calls to the backend. The AudioMass files (HTML/CSS/JS) would be baked into an image or mounted from the repository.
- `backend` – The Python REST API and processing service. This container includes Python 3 with all necessary libraries (ffmpeg, pydub, openai-whisper, faster-whisper, etc., PyTorch with CUDA/ROCm for GPU, Feedgen, etc.). It exposes endpoints for file upload and maybe status queries. It will also have access to the host's GPU (for NVidia, we'd use `runtime: nvidia` or the newer `--gpus` flag in Compose to pass the GPU; for AMD, we might have to ensure ROCm support is enabled). The backend container mounts a volume like `./data` to persist any outputs (or share with n8n). It also might mount a models volume if we want to cache Whisper models (though they typically download to ~/.cache by default).
- `ollama` – The LLM server. We'll use an official Ollama image (if available, e.g. from Docker Hub [31]) or build one from their GitHub. This container will need to access the GPU as well if we want

acceleration. It will also mount a volume for model storage (so that pulling a model persists). We can specify which model(s) to preload by either configuring Ollama's startup or by running `ollama pull <model>` via an entrypoint script. The container exposes port 11434 (the API).

- `n8n` – The automation service. We'll use the official n8n image. It needs a volume to save its workflow data (so that if restarted, workflows persist) – e.g., `n8n_data:/home/node/.n8n`. It will expose its editor UI on port 5678 (which we can keep accessible only locally). Environment variables for n8n can set things like execution mode, etc., but defaults are fine. The crucial config is to set up credentials for YouTube API within n8n's UI after deployment (this is a one-time user step). We will also mount the shared `./data` volume here (read-write) so that n8n's nodes (like the Upload video node or any file operations) can see the files produced by backend.
- (Optional) `db` – If n8n needs a database, by default it uses SQLite (which is stored in the n8n_data volume). We may not need a separate DB service unless scaling.
- (Optional) `proxy` – If we want a single entrypoint, we might run a reverse proxy that routes e.g. `/api` to backend, `/` to the UI, etc. But this is optional for local use.
- (Optional) `whisper-worker` – If we decide to offload Whisper to its own container (for example, a dedicated service that listens for audio via a queue and returns transcript), we could containerize that separately. However, likely we keep it within the backend to simplify.

Directory Structure: In the project repository, we can organize files as follows:

```
podcast-platform/
├── docker-compose.yml
├── frontend-ui/                # Frontend UI assets (AudioMass or custom UI
code)
│   ├── index.html
│   ├── audiomass.js, audiomass.css, etc.
│   └── ...
├── backend/                    # Backend service code
│   ├── app.py (Flask or FastAPI app)
│   ├── requirements.txt (e.g. flask, ffmpeg-python, whisper, etc.)
│   ├── processors/
│   │    ├── audio_edit.py (calls ffmpeg for concat/volume)
│   │    ├── transcribe.py (wraps Whisper)
│   │    ├── summarize.py (calls Ollama API)
│   │    └── video_render.py (ffmpeg waveform generation)
│   └── Dockerfile
├── workflows/                  # n8n workflow definitions (if exporting as
JSON)
│   └── publish_workflow.json
├── data/                       # Shared volume directory (or could be outside
repo)
│   ├── outputs/ (audio/video files)
│   └── rss/ (podcast.xml and possibly media files)
└── README.md (developer setup instructions)
```

Each major component (frontend, backend) has its own Dockerfile for reproducibility. The Compose file ties them and handles networking. For example, the `backend` service in docker-compose might include:

```
services:
  backend:
    build: ./backend
    ports:
      - "5000:5000"        # API port
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia  # for GPU support on NV
              capabilities: [gpu]
    volumes:
      - ./data:/app/data    # mount for outputs and model cache if needed
    environment:
      - NVIDIA_VISIBLE_DEVICES=all   # (if using NVIDIA runtime)
      - OLLAMA_URL=http://ollama:11434  # env to point to Ollama API
    networks:
      - default
```

And the `ollama` service might look like:

```
  ollama:
    image: ollama/ollama:latest
    ports:
      - "11434:11434"
    volumes:
      - ./models:/root/.ollama  # store models
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              capabilities: [gpu]    # if we want GPU for LLM
```

The n8n service:

```
  n8n:
    image: n8nio/n8n:latest
    ports:
      - "5678:5678"
    volumes:
```

```
    - n8n_data:/home/node/.n8n
    - ./data:/data        # to access files
  networks:
    - default
```

And possibly an `nginx` service for the UI:

```
web:
  image: nginx:alpine
  ports:
    - "80:80"
  volumes:
    - ./frontend-ui:/usr/share/nginx/html:ro
  networks:
    - default
```

In the above, we mount the `frontend-ui` folder directly to Nginx html folder for simplicity (in production one might bake it in the image). This serves the AudioMass application. We also might add an Nginx config to proxy API calls to `backend:5000` if needed (so the UI can call `/api/upload` without CORS issues). Alternatively, the UI could directly call the backend's address and we enable CORS in backend.

Message Brokering vs Direct Calls: The architecture predominantly uses direct HTTP calls and shared volumes to integrate services (the backend calls Ollama's REST API, triggers n8n via webhook, etc.). The question of using a **message broker** or an **MCP (Message/Model Control Protocol)** for inter-service communication is worth evaluating. In our design, since the number of services is small and the workflow largely linear, simple REST/HTTP is sufficient and easier to implement. For example, the backend knows exactly when to call the LLM or n8n and can do so synchronously over HTTP. Introducing a message broker (like Redis Pub/Sub, RabbitMQ, MQTT) could decouple components but would add complexity. A broker would be useful if we had many asynchronous events or wanted to queue multiple jobs (e.g., multiple users processing concurrently) – then the backend could drop tasks onto a queue and worker services (transcription worker, render worker) could pick them up. If scaling to a cluster or if we anticipate heavy load, that pattern is beneficial. In the current scope, a broker is **optional**. We can achieve asynchrony by spawning background threads or tasks for Whisper and rendering without blocking the API. However, we will keep the architecture flexible: by using n8n for orchestration, we already have a form of a control mechanism (n8n is like a higher-level orchestrator using its own internal message passing for nodes). If needed, we could integrate a lightweight messaging system: e.g., the backend could publish an event "transcription finished" that triggers the next step. But given that our pipeline steps are fairly sequential, direct function calls or HTTP calls suffice.

For completeness, **MCP (Model Control Protocol)** refers to emerging patterns for tool-LLM interaction pipelines [32]. In our case, an MCP could hypothetically manage the interactions between Whisper and the LLM – but since Whisper and the LLM don't directly talk to each other (the backend coordinates them), we don't require a specialized protocol. Should the architecture grow (say integrating multiple AI models in sequence), one could consider adopting a standardized message or context protocol for clarity [33]. For now, we will implement straightforward REST endpoints and use n8n's built-in scheduling for any timed jobs

(e.g., we could schedule n8n to check an “episodes” folder and automatically process new files, etc., if we wanted a fully hands-off pipeline – but since we have the manual UI step, real-time triggering is fine).

Resource and Performance Considerations

It’s important to size the hardware requirements for each component, especially since we aim to support GPU acceleration. Below is an outline of the expected CPU/GPU, RAM, and storage needs:

- **Web UI (AudioMass):** Runs in the user’s browser. Performance depends on the client machine – editing a typical hour-long audio (possibly represented in memory) is feasible on modern browsers. No server CPU/RAM usage (aside from serving static files). Ensure the user’s browser is fairly up-to-date for best results (AudioMass supports most modern browsers and even mobile devices [34]).

- **Backend (FFmpeg processing & glue):** This container will use CPU for FFmpeg tasks. FFmpeg scales with CPU cores – on a 4 or 8 core system it will be quite fast for audio operations and moderately fast for video rendering. Memory usage of FFmpeg is low (usually under a few hundred MB even for video encoding). The Python process for coordinating tasks will use minimal RAM (maybe a few hundred MB for loading Whisper model and handling data). **Storage:** The backend will produce output files – ensure there is enough disk space for large videos and audio. For example, a 1-hour WAV audio could be ~600 MB, MP3 maybe ~60 MB, and a 1-hour 1080p MP4 (waveform on static image) could be on the order of 100–200 MB (depending on compression settings). We should allocate at least a few GB of space for safety.

- **Whisper (ASR):** If using the **large model**, it’s ~1.5 GB on disk and requires ~10 GB of **VRAM or RAM** to run [12]. On GPU, it will occupy up to 10 GB VRAM during processing (and also use ~10 GB of system RAM while loading, as noted by OpenAI [30]). On CPU, it will use a lot of RAM as well (same order, 10+ GB) and be much slower. If the system has less memory, using the **medium model** (~769M params) requiring ~5 GB RAM [35] is an option, or even the small (~2 GB RAM) for quicker, albeit less accurate, transcription. We will allow model selection based on environment. Whisper will also use significant CPU when not using a GPU – it parallelizes across cores. For example, real-time transcription of an hour of audio on CPU might take 2-3 hours on a 6-core machine with the large model (just an estimate; smaller models can approach real-time). With a good GPU (e.g. an NVIDIA 2080 or 3080), it can transcribe faster than real-time [36]. We expect typical usage to have a GPU available for smooth performance. If AMD GPU, ensure ROCm is installed; if not available, default to CPU small model to avoid excessive load. In our Compose setup, we will include the necessary drivers for GPUs. **Note:** Consider using the Faster-Whisper implementation which can use INT8 and even 4-bit quantization to drastically lower memory (reducing large model to ~2.5 GB memory with 8-bit) [16]. This would also enable running on moderately powered machines without 10GB free.

- **Ollama/LLM:** Requirements vary by model loaded. For a **7B model (e.g. Llama 2 7B)**, recommended RAM is 16 GB (8 GB minimum) [37]. For **13B**, 32 GB recommended [20]. These are high because Ollama might keep model in memory for responsiveness. However, using 4-bit quantization, the actual model size in RAM is much smaller (a 7B 4-bit is ~4 GB). In practice, a system with 16 GB RAM can run a 7B or 13B quantized model as long as nothing else is consuming too much memory. The LLM inference will use CPU by default (possibly all cores during generation). If an NVIDIA GPU is available, and if Ollama supports offloading to it (which it does through GGML CUDA acceleration for 4-bit), then having a GPU with ~6-8 GB VRAM can accelerate the generation. LLM generation for a

title/summary (which is a relatively small output) is not very heavy – even on CPU it should only take a few seconds to produce a couple hundred tokens summary. So this is not a bottleneck. Disk space for models is significant: the download for a 7B model can be ~3–4 GB (for quantized) up to 13 GB (for full precision). We will budget ~10–20 GB of disk for storing LLM models if needed. If the user wants a larger 30B or 70B model for higher quality, GPU memory will become a limiting factor (we'd need ~20 GB VRAM for a 30B in 4-bit, which most consumer GPUs don't have, so it would be CPU-bound and require 32+ GB RAM, which is likely out of scope for most user machines). Thus, we'll default to 7B/13B class models which are within reach of an enthusiast's PC.

- **n8n (Automation):** n8n itself doesn't do heavy computations; it's I/O bound (making API calls, waiting for responses). A container with 1 CPU core and 1–2 GB RAM is generally enough for n8n in this context [38] [26]. Each active workflow will consume some memory (especially if large data is passed around in it). In our case, the main heavy data in the workflow is the video file – but we won't load the whole video into memory in n8n; instead, the YouTube node streams from file. So memory impact is small. There will be some overhead if we use the Code node to manipulate XML for RSS, but that's trivial. **Disk:** n8n's data volume (for SQLite DB or any saved credentials) will be just a few MB.

- **GPU Support (NVIDIA vs AMD):** We have accounted for NVIDIA via Docker runtime. For AMD, the situation is a bit more complex. To utilize AMD GPUs, the containers (backend and possibly Ollama) need to have ROCm-compatible builds. If the host is Linux with a supported AMD GPU, we could use a ROCm-enabled PyTorch in the backend. The Whisper library can then use `device="cuda"` (which under ROCm works by mapping to AMD) [13]. Alternatively, the **whisper.cpp** (C++ implementation of Whisper) could be used, which has OpenCL support that might run on AMD GPUs – but that's an alternative approach. For Ollama, as of current knowledge, it uses `llama.cpp` under the hood; `llama.cpp` can offload to GPU but primarily NVIDIA (via cuBLAS) or on Apple (Metal). There is experimental OpenCL support in llama.cpp which could, in theory, work on AMD OpenCL drivers. If that's viable, Ollama might gain AMD support. Otherwise, AMD systems would default to CPU for LLM, which is still functional. The bottom line: the design tries to accommodate AMD but the level of support may not match NVIDIA's out-of-the-box performance. We will note this in documentation and possibly provide container variants or flags for AMD (e.g. use ROCm PyTorch image, etc.).

Custom Development Requirements

Most components are off-the-shelf, but **glue code and custom integration** will be needed in a few areas:

- **UI-Backend Integration:** We need to modify or extend AudioMass to send the edited audio to our backend. AudioMass is open source and could be forked – we can add a feature where clicking "Export" triggers a JavaScript `fetch` POST of the audio data to the server (in addition to offering the download). This will require knowledge of AudioMass's codebase (which is fairly small, ~65kb JS). We may also create a simple UI around AudioMass (e.g. a parent page that embeds the editor and has additional controls like a "Publish" button). Also, if we want to allow the user to input episode metadata (like an episode number or manual title) or choose a background image for the waveform, we'll need to add form elements in the UI and send those inputs to the backend. Implementing multi-track support (for intro/outro) in the UI might be challenging if AudioMass doesn't support it natively. A workaround is to let the user handle intro/outro as part of the editing (e.g. instruct them to copy-paste audio files together in one track). If that's insufficient, we might custom-build a very

simple timeline that just allows reordering full tracks (this could even be a form where user uploads 3 files and we concatenate – but that loses the nice wave editing). For now, we assume the user can manage within the single-track editor environment. This integration work will be custom development.

- **Backend Logic:** We will write the Python code that orchestrates everything – receiving the upload, calling ffmpeg/whisper/ollama, tracking progress, saving files, and handling errors (e.g., if Whisper fails or takes too long, etc.). This includes crafting the prompt for the LLM (which might be tuned over time for best results). Also, generating the RSS feed (likely using feedgen) is a custom scripting task – though small, it requires coding and testing to ensure the XML meets podcast specs (RSS 2.0 with itunes podcast extensions perhaps). We will implement that within the backend or as part of the n8n workflow (maybe easier in backend using feedgen, and just have n8n call a webhook to retrieve the ready XML).

- **Workflow Creation in n8n:** Setting up the n8n workflow is a one-time configuration, but it's part of the development deliverables. We (the developers) will create the "Publish Episode" workflow with the necessary nodes. This isn't coding in the traditional sense, but it's a technical task to configure and test. We'll document how this workflow is set up so it can be deployed along with the system (n8n allows exporting workflows as JSON). Custom nodes or credentials might need to be configured (especially the YouTube OAuth credentials). This might require creating a Google Cloud project for YouTube API and storing the client secrets in n8n – a step to document for deployment.

- **Testing and Iteration:** We will likely write some helper scripts to test individual pieces (e.g., a script to transcribe a sample audio and get a summary, to validate Whisper+Ollama integration, etc.). These might not be part of final product, but are part of development.

- **Optional Enhancements:** In the future, custom development could extend the platform with features like: voice enhancement (using AI noise removal), multiple speaker detection in transcripts, chapter generation, social media clip generation, etc. These are out of scope for now but the architecture can accommodate them (likely via additional n8n workflows or additional AI models). The team should ensure the architecture is documented clearly so that new services (e.g. a "denoiser" container or an "NLU annotator") can be added if needed.

Conclusion

By combining powerful open-source tools in a containerized architecture, we achieve a **fully local podcast production pipeline**. The system enables a user to go from raw recordings to a polished published episode with minimal manual steps: intuitive drag-drop editing, one-click AI-generated summaries, and automated publishing. We have chosen components that are each best-in-class in their domain (AudioMass for editing [1], Whisper for transcription, Ollama for local LLM, FFmpeg for media processing, n8n for automation [25]) and detailed how they communicate and complement each other. The architecture is modular and extensible, using Docker Compose to allow deployment on any machine (with support for GPU acceleration when available). We have outlined hardware requirements to guide deployment (ensuring the host machine has adequate CPU/GPU and memory for a smooth experience). Table 1 below recaps the resource considerations for major components:

Component	CPU / GPU Needs	Memory (RAM/VRAM)	Storage
Web UI (AudioMass)	Uses **client CPU** (browser executes audio ops). No server CPU needed. GPU not used.	Depends on audio length; e.g. a few hundred MB RAM in browser for a long audio.	N/A (runs in browser). Server storage not used.
FFmpeg (Audio/Video)	**CPU intensive** (multithreaded); benefits from 4+ cores. GPU not required (no CUDA used in our filters).	Low RAM (< 0.5 GB typically). CPU usage high during renders.	Needs disk for temp files and outputs (e.g. waveform video ~100+ MB per hour of video).
Whisper ASR	**GPU highly recommended** for large model – needs ~10 GB VRAM [12] (NVIDIA or ROCm AMD). Can run on CPU (slow).	~10 GB RAM/VRAM for large; 5 GB for medium, 2 GB for small model [12]. Also ~100% of one GPU or heavy use of all CPU cores during processing.	Model file ~1.5 GB (large). Smaller models: 0.5 GB or less. Cache to disk for reuse. Transcripts (text) negligible size.
Ollama LLM	**GPU optional** (uses CPU if GPU not present; GPU (NVIDIA) speeds up 4-bit inference).	~8–16 GB RAM for 7B–13B models (quantized) [19]. CPU multithreaded during generation. If GPU used, ~6–8 GB VRAM per model offloaded.	Model files: 4–8 GB (quantized 7B/13B). Ensure volume for storing models. Output text is tiny.
n8n Orchestrator	CPU: minimal (lightweight Node.js). No GPU usage.	~1 GB RAM (plus overhead for any in-memory data, which is small for us) [38].	Few MB for database (workflows, credentials). Logs and workflow exports also minimal.
Publishing Outputs	CPU: YouTube upload uses negligible CPU (network-bound). RSS generation trivial.	Memory: negligible (RSS XML build in memory is tiny; YouTube upload stream buffered).	Video upload requires network bandwidth. RSS and media hosting requires storage if self-hosting files.

In conclusion, the proposed platform achieves a high degree of **automation and integration** while remaining self-hosted and open-source. The development team should follow this design to set up each service, implement the integration points, and test the entire flow with sample podcasts. The end result will streamline the podcast production workflow – from editing through AI-powered content generation to multi-channel publishing – all on local infrastructure under the user's control. By documenting each component's role and interfaces, we ensure the system is maintainable and ready for future enhancements by the team.

1 2 3 4 34 AudioMass - About
https://audiomass.co/about.html

5 6 7 8 9 Generating podcast audiograms with FFmpeg – Tales from Trantor
https://talesfromtrantor.com/behind-the-scenes/generating-audiograms-with-ffmpeg/

10 11 GitHub - nypublicradio/audiogram: Turn audio into a shareable video.
https://github.com/nypublicradio/audiogram

12 29 35 GitHub - openai/whisper: Robust Speech Recognition via Large-Scale Weak Supervision
https://github.com/openai/whisper

13 ROCm support · openai whisper · Discussion #55 - GitHub
https://github.com/openai/whisper/discussions/55

14 Use AMD GPU for speech recognition with Whisper - KDE Discuss
https://discuss.kde.org/t/use-amd-gpu-for-speech-recognition-with-whisper/28766

15 16 faster-whisper · PyPI
https://pypi.org/project/faster-whisper/

17 18 GitHub - ollama/ollama: Get up and running with Llama 3.3, DeepSeek-R1, Phi-4, Gemma 3, Mistral Small 3.1 and other large language models.
https://github.com/ollama/ollama

19 20 22 37 How Much Memory Does Ollama Need?
https://www.byteplus.com/en/topic/405436

21 What are the minimum hardware requirements to run an ollama ...
https://www.reddit.com/r/ollama/comments/1gwbl0k/what_are_the_minimum_hardware_requirements_to_run/

23 YouTube node documentation | n8n Docs
https://docs.n8n.io/integrations/builtin/app-nodes/n8n-nodes-base.youtube/

24 lkiesow/python-feedgen: Python module to generate ATOM feeds ...
https://github.com/lkiesow/python-feedgen

25 26 27 GitHub - n8n-io/n8n: Fair-code workflow automation platform with native AI capabilities. Combine visual building with custom code, self-host or cloud, 400+ integrations.
https://github.com/n8n-io/n8n

28 GitHub - pkalogiros/AudioMass: Free full-featured web-based audio & waveform editing tool
https://github.com/pkalogiros/AudioMass

30 OpenAI Whisper performance benchmarks - 1 QuBit
https://www.1qubit.de/en/ai/openai-whisper-performance-benchmarks

31 Docker Image - ollama
https://hub.docker.com/r/ollama/ollama

32 Introducing Model Control Protocol (MCP): A Key to Efficient Model ...
https://medium.com/@leela.kumili/introducing-model-control-protocol-mcp-a-key-to-efficient-model-coordination-in-ai-systems-5e7148913cae

33 Designing Model Context Protocols (MCPs) the Right Way - Reddit
https://www.reddit.com/r/ClaudeAI/comments/1ir77ab/designing_model_context_protocols_mcps_the_right/

36 GPU Recommendation for Whisper - OpenAI Developer Community
https://community.openai.com/t/gpu-recommendation-for-whisper/300608

38 n8n.io: The Rising Star in Workflow Automation Explained
https://www.atakinteractive.com/blog/n8n.io-the-rising-star-in-workflow-automation-explained