# Episode Book

EPISODE #5573

# Audio Post-Processing for AI Podcast Pipelines

| | |
|---|---|
| PUBLISHED | 26 September 2026 at 09:52 UTC |
| DURATION | 25:58 |
| LISTEN | myweirdprompts.com/episode/audio-post-processing-pipeline/ |
| HOSTS | Corn, Herman, Hilbert |
| SCRIPT MODEL | deepseek-flash |
| VOICES | chatterbox-regular (chatterbox-local) |
| PIPELINE | V5.2 |
| BOOK GENERATED | 19 October 2026 at 17:37 UTC |

## SYNOPSIS

The front end of an AI podcast pipeline gets all the attention — agents, prompts, retrieval. But the stage nobody films a documentary about is audio post-processing, and it's where the actual craft lives. This episode walks through four specific edits: silence truncation, loudness normalization, EQ, and speed normalization without pitch distortion. Along the way, we dig into why SoX and FFmpeg split the work, why a single wrong integer in FFmpeg's silenceremove filter can silently delete 90% of your episode, and why the order of operations between silence removal and loudnorm isn't a style preference — it's a correctness issue. Plus: when to preview with silencedetect, how EBU R128's double-pass mode works, and whether any of these jobs actually benefit from GPU inference.

# TRANSCRIPT



### Corn

And what I keep coming back to is that the front end of our pipeline is agentic, and the back end has quietly become a shell script. The script-writing agent, the metadata construction, the prompt cleanup, the research agent grounding everything in retrieved context, that's the part everyone wants to talk about. Then the finished script goes to Chatterbox running as a serverless container on Modal, the voices come out of voice embeddings so they stay stable, all the chunks and the standard elements get concatenated into one file, and a periodic deployment pushes the feed. And at the very end of that chain, there's a stage nobody films a documentary about, which is the audio post-processing.



### Herman

Which is where the actual craft is, if you ask me.



### Corn

Daniel sent in a prompt that's really about exactly that stage. He walks back through the pipeline, and then he asks which command-line tools are best suited to four specific edits. Silence truncation, for cutting dead air out of TTS output, with the caveat that an aggressive setting clips natural pauses. Loudness normalization, which he flags as the one that matters most for podcast distribution. EQ. And then speed normalization, which is the one he wants to revisit adding to our own pipeline, because he's noticed the TTS sometimes generates a passage too slowly or too quickly, the pace drifts, and the naive fix distorts pitch. So he wants to know what tool you reach for to keep pacing consistent across an episode, or across a single speaker, without modulating the pitch.



### Herman

That's the hard one.



### Corn

It is. He also makes a broader observation I think is right, which is that these CLIs are powerful and well established on Linux, but before AI agents showed up they were underused, because most people would rather drag a slider in a visual editor than read a man page. Now they can be dropped into an automated pipeline and nobody has to look at them. And he raises two meta-questions. First, if you're using several effects, how do you chain them across a thirty-minute file? Second, since our pipeline runs on Modal, do any of these jobs actually benefit from GPU inference, or do they run fine on server-side CPU?



**Herman**

Two of those four are easy, one is medium, and one is a genuine research project. So let's start with the two tools that keep coming up, SoX and FFmpeg, and why practitioners split between them.



**Herman**

SoX is Sound eXchange, and its own man page describes it as particularly suited to making quick, simple edits and to batch processing. Then it does something I find charming, which is that it points you at Audacity if you need an interactive graphical audio editor. The tool knows what it is. Its power is the effect chain. You apply echoes, filters, EQ, compression, silence trimming, pitch and tempo changes, the same way a DAW does, except it's one line in a terminal.



**Corn**

And FFmpeg is the other one, which is less an audio tool and more a general media framework that happens to have an enormous audio filter set bolted onto it.



**Herman**

Right, and the split people actually make is SoX for pure audio effect work, FFmpeg for container, format, and segmentation work. There's a comment from a user called klodolph on Hacker News from a few years back that sums it up better than any documentation. He says SoX is amazing, it's way easier to use for audio stuff than FFmpeg is, and then he lists what he actually does with it. High-pass, normalize, automatically trim audio files, add fade-in or fade-out, downmix to mono, then resample and dither. That's a whole mastering chain in one sentence.



**Corn**

And the real-world pattern shows up in that police-scanner transcription project, where dockerized FFmpeg processes slice the streamed audio into thirty-second wav files, and then SoX removes the silence. FFmpeg for the segmentation, SoX for the trim. That's the division of labor in the wild.



**Herman**

Except here's the tension. FFmpeg now has silenceremove, it has loudnorm, and it even has a rubberband filter. So a single FFmpeg graph can do the entire chain. One tool, one process, one pass. And that's attractive when you're automating, because every additional



### Herman (continued)

binary in the pipeline is another thing that can fail on a Sunday.



### Corn

Or another thing that can silently swap your left and right channels, which I have experience with.



### Herman

You have mentioned that once or twice.



### Corn

I noticed it after a week. Anyway. Let's take the four edits one at a time, starting with the one that has the nastiest failure mode, which is silence truncation.



### Herman

This is where the parameters don't mean what people think they mean, and the consequences are severe. FFmpeg's silenceremove filter is the workhorse here. The first counterintuitive thing is that start_periods equals one trims only leading silence. Plenty of guides say it trims both ends. They are wrong. If you want to shorten gaps in the middle of the file, you need stop_periods equals minus one. The negative value is what tells it to act on every silence after sound stops, not just the first one.



### Corn

And a positive stop_periods equals one is a trap.



### Herman

It's a catastrophe. It deletes everything after the first silence. There's a test documented in the FFmpeg cookbook, verified against version eight point one, where an eight-second test file collapsed to one point zero two seconds. You feed it a finished episode and you get a jingle and a half-second of breath.



**Corn**

So the sign of one integer is the difference between a trimmed file and a deleted one.



**Herman**

And it fails silently. That's the thing. It doesn't error, it doesn't warn, it just hands you back a file that's ninety percent shorter and exits cleanly.



**Corn**

What's the parameter that actually protects the natural pauses?



**Herman**

stop_duration. That's the one that says shorten any silence longer than a second down to a second. The cookbook's recommendation for natural-sounding conversation is stop_duration equals one with stop_silence equals zero point three. For tight editing, you drop stop_duration to zero point three. And stop_silence is additional silence kept, so if you set stop_duration to one and stop_silence to zero point five, you're leaving one and a half seconds in each gap. Which is a lot, but for a dramatic pause it might be exactly right.



**Corn**

And the failure pattern Daniel described, the over-aggressive clipping, maps directly onto the threshold and the duration.



**Herman**

For digitally generated audio, and that includes TTS output, you want a threshold between minus sixty and minus seventy dB. For a microphone in a quiet room, minus forty-five to minus fifty. Set it too high, minus thirty and up, and quiet speech or word endings get classified as silence and cut. And then there's the other failure, words chopped mid-sentence, which happens when stop_duration is too short and the pauses between words get treated as silence. The fix there is to lengthen it to half a second or a full second.



**Corn**

So the two failure pattern pull in opposite directions.



**Herman**

They do. Too aggressive and you're cutting the ends off words. Too conservative and you've removed nothing and you've added a processing pass for no reason.



**Corn**

What's the discipline that prevents this?



**Herman**

You preview with silencedetect before you cut anything. You run FFmpeg with the silencedetect filter at the threshold and duration you're considering, output to null, and it prints the exact silence start and end ranges it found. Then you feed those same values to silenceremove and you know exactly what it's going to do before it does it. That's the whole verification step. It costs you one pass over the file and it saves you from shipping an episode that's four minutes long.



**Corn**

And trailing silence? Because start_periods only handles the front.



**Herman**

Trailing silence needs what people call the areverse sandwich. You run silenceremove with start_periods equals one, then areverse, then silenceremove with start_periods equals one again, then areverse again. You flip the file, trim what is now the leading silence, and flip it back. The caveat is that areverse holds the entire file in memory. Multi-hour recordings cost gigabytes. Anything around an hour is fine, which for us is comfortably true.



**Corn**

And SoX has its own silence effect, which does beginning, middle, and end in one go.



**Herman**

It does, and its syntax is silence, optionally minus l, then above-periods with a duration and threshold, then below-periods with a duration and threshold. It's more capable in one invocation than silenceremove is. It's also famously confusing. There's a blog post from two thousand nine called The SoX of Silence that's still circulating, and the entire post is a person working out what the parameters mean. That's a fifteen-year-old complaint about the same four arguments.



**Corn**

Once the silences are right, the next question is how loud the whole thing is, and that's where EBU R128 comes in.



**Herman**

Loudnorm. EBU R128 is the broadcast loudness standard, and FFmpeg implements it in the loudnorm filter, originally written by Kyle Swanson, merged in twenty sixteen, built on libebur128. It targets three things. Integrated loudness, which is the overall perceived level. Loudness range, which is how much the level varies across the program. And maximum true peak, which catches inter-sample peaks that a sample-peak meter misses.



**Corn**

And there are two modes.



**Herman**

Single-pass, which is for livestreams where you can't look ahead, and double-pass, which is for files and is the recommended approach. The first pass measures the audio properties of the source. Then you take those measurements and feed them back in as input to the second pass. The reason it matters is that single-pass loudnorm is guessing at the dynamics, and on a file you don't have to guess.



**Corn**

There's a Python utility that automates that, ffmpeg-normalize.



**Herman**

It does EBU R128 normalization, two-pass by default with an option for one-pass dynamic, plus RMS-based and peak normalization, batch processing, and Docker support. If you're scripting a pipeline and you don't want to write the two-pass dance yourself, that's the tool.



**Corn**

And the podcast target?



**Herman**

The cookbook's example uses I equals minus sixteen, true peak minus one point five, LRA eleven. Broadcast targets are minus twenty-three LUFS for EBU in Europe and minus twenty-four for ATSC A slash eighty-five in North America. Podcasts generally land hotter than broadcast, and minus sixteen is the common number.



**Corn**

Now here's the ordering question, and this one has a real trap in it.



**Herman**

It does. The cookbook's combined example puts loudnorm after silenceremove, and the warning is explicit. If you normalize after removing silence, put loudnorm after silenceremove. Normalizing first changes what counts as silence. That's the whole thing in one line. If you raise the level first, your threshold is now measuring a different signal, and the silences you were going to cut are suddenly not below the threshold anymore, or worse, quiet passages that weren't silence get pushed up and then classified as sound.



**Corn**

So the order isn't a style preference, it's a correctness issue.



**Herman**

Silence removal operates on level. Loudness normalization changes level. Therefore silence removal goes first. It's almost tautological once you say it out loud, and it's the kind of thing that gets discovered by shipping a weird-sounding episode.



**Corn**

So silence and loudness are handled. Now the two harder ones. EQ, which is easy, and speed normalization, which is not.



**Herman**

EQ is easy because the tools are mature and the parameters are intuitive. SoX has a rich filter set. Equalizer, which is an RBJ peaking EQ biquad. Bass and treble, which are RBJ shelving biquads. Highpass and lowpass. Bandpass and bandreject. Allpass. And firfit, which does FFT convolution with a FIR filter built from a frequency response you supply. Those are the same building blocks Audacity exposes in its GUI, which is why the skills transfer.



**Corn**

And FFmpeg's equivalents.



**Herman**

Equalizer, bass, treble, highpass, lowpass, firequalizer, and anequalizer for multiband work. For a podcast voice chain, the standard move is a high-pass around eighty hertz to get rid of rumble and plosive energy, plus a presence shelf up in the intelligibility range. That's it. You're not mastering a record.



**Corn**

Which is why EQ is the one where the answer is boring.



**Herman**

It's boring because it's solved. Speed normalization is where it gets interesting, because the naive approach distorts pitch, or rather, the naive approach is designed not to, and it still doesn't sound right.



**Corn**

Explain the naive approach.



**Herman**

FFmpeg's atempo and SoX's tempo. Both are time-stretching algorithms, time-domain or phase-vocoder based, that preserve pitch by design. That's the whole point of them. If you speed up a file with atempo, the pitch doesn't change. So on paper, the naive approach already solves Daniel's problem.



**Corn**

But it doesn't sound good.



**Herman**

It sounds acceptable on some material and bad on others. Phase vocoders smear transients. Speech has a lot of transients, plosives, fricatives, the attack of consonants. Stretch a phase vocoder too far and you get that watery, phasey quality, that artifact everybody recognizes but nobody can name. So the naive approach is pitch-preserving but not artifact-free.



**Corn**

Which brings us to Rubber Band.



**Herman**

Rubber Band, from Breakfast Quay, written by Chris Cannam. It's a library and a utility program that permits changing the tempo and pitch of an audio recording independently of one another. That independence is the whole point. You can change the tempo without touching the pitch, or change the pitch without touching the tempo, or both.



**Corn**

And the CLI.



**Herman**

Minus t or minus minus time with a value stretches to that multiple of the duration. Minus T or minus minus tempo changes the tempo by a multiple, or with two values separated by a colon, changes the tempo from one value to another. Minus D or minus minus duration stretches or squashes to exactly that many seconds. And then the fine-tuning is where it gets interesting.



**Corn**

The fine-tuning being the knobs that matter for speech.



**Herman**

The FFmpeg filter form of Rubber Band exposes tempo, pitch, transients with options crisp, mixed, or smooth, detector with compound, percussive, or soft, formant with shifted or preserved, and pitchq with quality, speed, or consistency. Formant preservation is the under-discussed one.



**Corn**

Define formant for the listener who hasn't thought about it since a linguistics class.



### Herman

Formants are the resonant frequencies of your vocal tract. They're what make a voice sound like a specific body rather than a pitch. If you shift pitch without preserving formants, you get the chipmunk effect, because the formants move up with the pitch. Formant preservation keeps the vocal tract resonances where they were, so the voice sounds like the same person at a different pitch. For TTS character voices, that's the difference between nudging a voice and breaking it.



### Corn

So if Daniel ever wants to adjust how fast a character speaks, or nudge their pitch, formant preserved is the setting that keeps the character recognizable.



### Herman

It's the setting that keeps the character from sounding like a different, smaller animal. And it's the thing nobody mentions when they write about time-stretching, because they're all comparing algorithms on music.



### Corn

There's a comparison that's actually on speech, though.



### Herman

Justin Salamon benchmarked SoX against Rubber Band for pitch shifting and time stretching on both speech and music. He ran SoX with tempo minus s zero point eight three three against Rubber Band with time one point two and six channels. And his verdict is the most honest thing in this whole space. He says, which one's better? Like most things in life, it depends. For my application I have a relatively clear winner, but I won't bias you with your opinion.



### Corn

Which is a researcher declining to hand you a conclusion.



**Herman**

He found Rubber Band generally better, and he refused to declare a universal winner, because the material matters. That's the correct answer and it's the least satisfying one. Both have Python wrappers, pysox and pyrubberband, so you can drive either from a pipeline.



**Corn**

And then there's the licensing question, which is the part that actually kills projects.



**Herman**

Enabling Rubber Band binds the resulting binary to GPL or to Rubber Band's commercial license. That's a real constraint if you're shipping a binary. The alternatives are SoundTouch, which is LGPL, and Signalsmith Stretch, which is MIT. There's a benchmark harness from this year, timepitch-bench, that documents exactly this, and its README says if that's a problem for you, build with the Rubber Band flag disabled. So the community has already internalized that this is a build-time decision, not an afterthought.



**Corn**

So the best tool on quality grounds may not be the tool you can ship.



**Herman**

Which is a sentence that applies to a lot more than audio.



**Corn**

So you've got four effects and a thirty-minute file. How do you actually chain them?



**Herman**

FFmpeg chains them in a single filtergraph, comma-separated, in one pass. Silenceremove, loudnorm, equalizer, rubberband, in that order, and the order matters for the reason we already covered. SoX chains effects as space-separated arguments after the output file, and its man page notes that the positions of the output and the effects are



**Herman (continued)**

swapped relative to the logical flow, which is a gotcha that catches everyone once.



**Corn**

SoX puts the output filename before the effects.



**Herman**

It does, and it reads backwards until you internalize it. SoX in dot wav out dot wav effect one effect two. The file you're writing comes before the things you're doing to it. It's a small thing that costs people an afternoon.



**Corn**

And for a thirty-minute file, is single-pass fine?



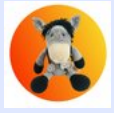
**Herman**

Single-pass is fine. The only memory-heavy step is areverse for the trailing trim, and that buffers the whole file, but thirty minutes is nothing. Two-pass loudnorm is the one structural complication, because it requires a scan pass and then an apply pass. So a full pipeline is effectively two passes. Pass one does silencedetect and the loudnorm measurement. Pass two applies everything in a single graph.



**Corn**

And the last question Daniel raised, does any of this need a GPU?



**Herman**

No.



**Corn**

That was fast.



**Herman**

I looked, and I want to be honest that this is a negative finding, which is a different thing from a positive one. Silence trimming, loudness normalization, EQ, and Rubber Band time-stretching are all CPU-bound C and C plus plus DSP. FFmpeg's loudnorm and silenceremove are pure CPU filters. Rubber Band is a CPU library. These are single-threaded-ish workloads doing arithmetic on samples. There's no matrix multiplication to offload. Nobody claims these are GPU-accelerated because there's nothing to accelerate.



**Corn**

For a Modal pipeline, the answer is CPU containers for this stage.



**Herman**

CPU containers, and that's the cheaper choice as well as the correct one. The GPU belongs to the neural stages. Chatterbox, the voice embeddings, any neural vocoder. Those are the parts that actually need a GPU. The post-processing is arithmetic.



**Corn**

There is one GPU thing in the space, though.



**Herman**

There is, and it's a research direction rather than a production need. TorchFX, a paper from April of last year, presented at DAFx. It's a GPU-accelerated Python library for DSP built on PyTorch, with FIR and IIR filters and a pipe operator for chaining. Their benchmarks show significant efficiency gains over traditional libraries like SciPy, particularly in multichannel contexts. But the paper itself notes current limitations in GPU compatibility.



**Corn**

Multichannel being the operative phrase.



**Herman**

Multichannel. If you're processing sixty-four channels of something, moving it to a GPU starts to make sense. If you're processing a stereo podcast, you're paying for a GPU to do what a CPU core does in a fraction of the time it takes to schedule the job. The interesting frontier isn't putting the DSP on a GPU. It's integrating the DSP with the neural stages, so the same framework does both.



**Corn**

There's a gap I want to flag before we get to Hilbert, which is that Daniel asked about speed normalization specifically, and there is no dedicated tool for it.



**Herman**

There isn't. There's no CLI that detects pacing drift and corrects it. What you'd have to build is a detector plus a corrector. The detector measures per-segment duration, either through forced alignment or by analyzing chunk durations, and computes a ratio. Then you apply Rubber Band or atempo per segment with that computed ratio. The pieces exist. The assembled tool doesn't.



**Hilbert**

I had the above-periods and below-periods arguments backwards for a week.



**Corn**

Say that again.



**Hilbert**

SoX silence effect. I was doing audio restoration for a small archival label, old radio dramas onto CD, and my whole job was running SoX and FFmpeg scripts over tape transfers. And I had the above-periods and the below-periods the wrong way round. I



### Hilbert (continued)

spent five days convinced the tool was broken. Five days. I rewrote the script twice. I filed a bug report that I'm still slightly embarrassed about. The parameters are named from the perspective of the signal crossing the threshold, and I was reading them as first and second.



### Corn

The tool was right the whole time.



### Hilbert

The tool was right the whole time. That's the lesson. You verify before you cut, because the failure is silent and the tool will not tell you.



### Herman

That matches what we were saying about silencedetect.



### Hilbert

It does. And on the areverse sandwich, I'll push back slightly, if that's the word. I never used it. For a podcast where you control the TTS output, you know where the tail is. You measure the file once, you trim a fixed duration off the end, and you're done. The sandwich is clever and it's the right answer for a tape transfer where you have no idea what's on the end. For our pipeline it's overkill.



### Corn

Fair.



### Hilbert

The other thing. I tried Rubber Band in a commercial pipeline once and legal told me to take it out. GPL. So I swapped in SoundTouch and I could not tell the difference on speech. Not on a single file. Maybe a trained ear could on music. On speech, no.



**Herman**

That's the licensing point landing in practice.



**Hilbert**

It landed in a meeting. It cost me a day and a half of rework. And the thing I remember is that the swap was easy and the decision was not. Nobody could tell me whether we were shipping a binary. It took four people and two weeks to answer a question that determined which library we used.



**Corn**

The tool you can ship isn't always the tool that sounds best.



**Hilbert**

It never is. The one I'd reach for on my own time is Rubber Band. The one I shipped was SoundTouch. Both of them worked. I still have the scripts.



**Corn**

That's a good reminder that the tool you can ship isn't always the tool that sounds best. Let's close on the gap that's still open.



**Herman**

The gap is speed normalization. Everything else on Daniel's list has a mature answer. Silence truncation is silenceremove with the right sign on stop_periods and a threshold that matches your source. Loudness is loudnorm in two-pass, after the silence removal, at minus sixteen. EQ is a high-pass and a shelf. Speed is the one where you have to assemble the tool yourself, and the assembly is the interesting part, because the detector is where the judgment lives.



**Corn**

The frontier isn't the GPU.



**Herman**

The frontier isn't the GPU. TorchFX is interesting for multichannel and for pipelines where the DSP lives in the same framework as the model. For a stereo podcast, the compute was never the problem. The parameter semantics were the problem. Which is why an agentic pipeline needs to verify before it cuts. The tools are old, the parameters are confusing, and the failures are silent, and that combination is exactly what an automated system is worst at catching.



**Corn**

If you're building an automated audio pipeline, the CLIs are there, they're cheap on CPU, and the hard part isn't the compute. It's knowing what the arguments mean. Thanks as always to Hilbert Flumingtop, our producer. This has been My Weird Prompts. If you're enjoying the show, a review helps more than you'd think.



**Herman**

We'll be back soon.