CART 457 W2025
Michael Hemingway
40032015

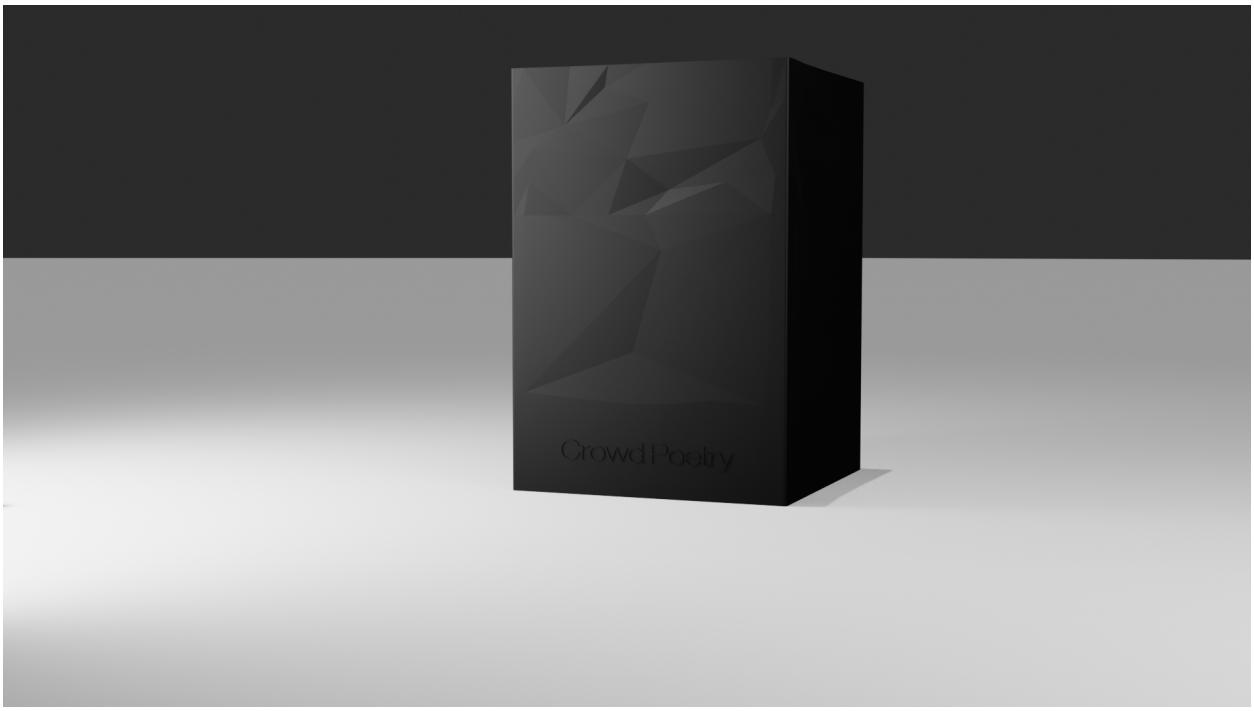
Crowd Poetry

https://github.com/stockhuman/crowd-poetry

Abstract

At the outset of this project, the intention was to develop a dynamic "crowd sampler" instrument that allowed users to input a poem from anywhere and have a device autonomously search for, download, and sample relevant material from the internet to create an evolving audio experience. This vision was broadly achieved, though the precise technical direction changed considerably from initial planning and the aesthetic product (the soundscape and physical device) were largely shaped by the technical constraints of the project itself.
Crowd Poetry ultimately makes dual use of user-generated inputs, though the process of creating poetry is that which gives it the most order and structure as a piece, where the sound clips as designed and implemented are decontextualized to a degree bordering on the abstract.

A key aesthetic goal for the project was capture a feeling for the pulse of the internet, and that was sonically achieved. Regardless of poem input, the raw data ingested into the soundscape never failed to evoke this imagery.



Render of the final hardware enclosure

Introduction

The initial vision framed this sampler as a standalone device, ideally self-contained, capable of interfacing with live data streams and supporting responsive interaction through embedded hardware. The project drew conceptual strength from prior work in audio hardware, UI/UX design, interactive fiction, and procedural generation, and it aimed to merge these disciplines into a poetic instrument that blurred the line between tool and performance.

Throughout development, however, the technical direction of the project changed significantly. Early constraints—such as limitations in official APIs, insufficient processing power on selected microcontrollers, and the complexity of real-time audio transformation—forced a shift away from a purely embedded solution. Instead, the project evolved into a hybrid architecture where a Raspberry Pi 4, connected via lightweight infrastructure and controlled by Python scripts, served as both the processing core and the host of a local SuperCollider-based audio engine. This transition preserved the artistic intent while allowing for a more flexible and iterative development cycle.

Despite these shifts, the central aesthetic aim remained intact: to capture the pulse of the internet in sonic form. While the final system does not fully implement real-time data stream integration, the audio compositions generated from poem inputs still evoke a visceral sense of internet culture—fragmented, layered, overwhelming, and fleeting. Each clip is decontextualized to the point of abstraction, yet collectively they carry a texture recognizable as “online.” The user, in shaping the poem, imposes a temporary structure on this noise, allowing the machine to build a sonic world that is both chaotic and curiously legible.

Crowd Poetry ultimately becomes a negotiation between user intention and algorithmic entropy. It highlights the tensions between authorship and automation, curation and randomness, signal and noise. The result is not just a device, but a speculative platform for exploring how personal expression and collective media collide in the digital age.

Research and Discovery

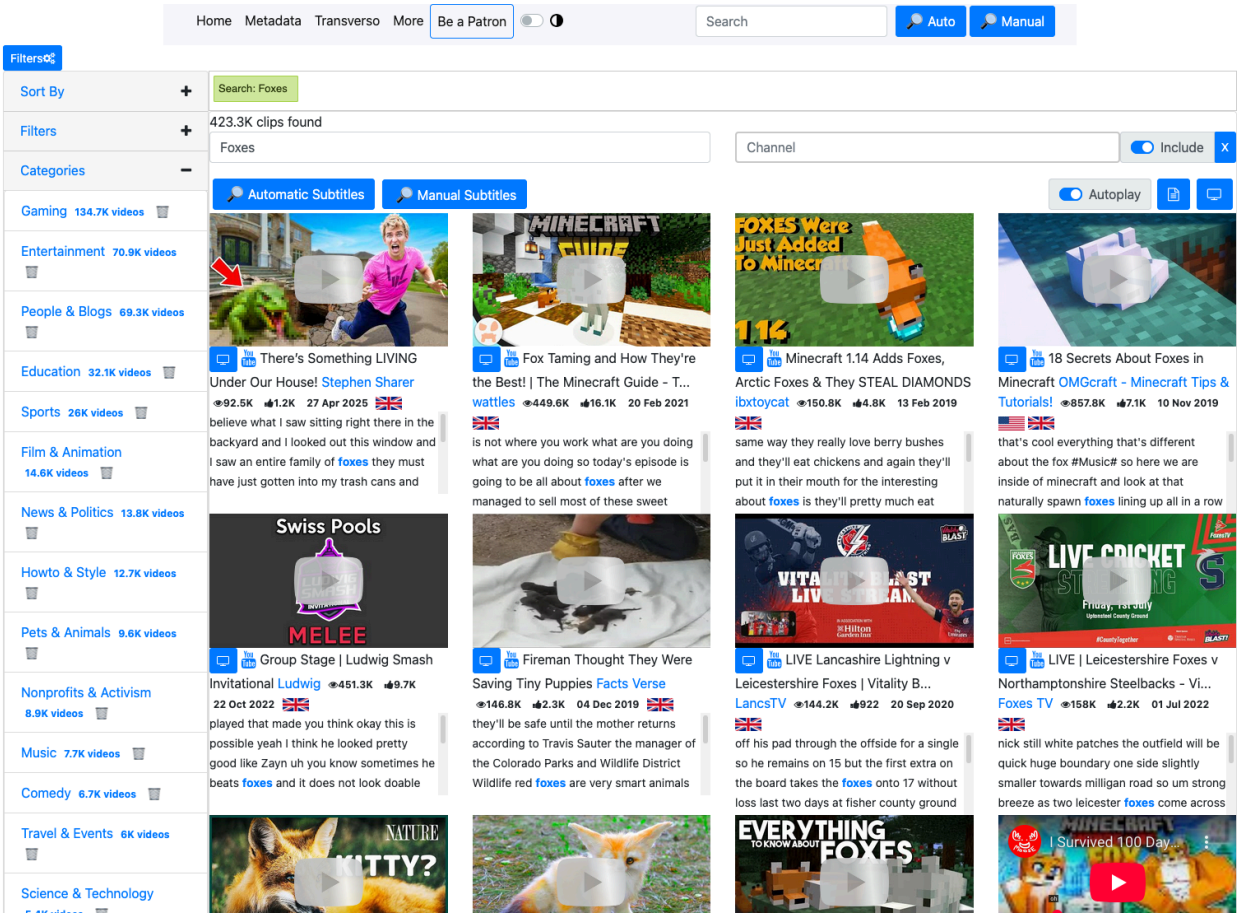
In the early stages of development, Crowd Poetry was envisioned as a device capable of sampling the web in real time, ingesting live data from major platforms such as TikTok, YouTube, and Twitter to generate soundscapes that reflected the constantly shifting dynamics of internet culture. As the idea progressed from concept to implementation, the technical feasibility of such integration became a central focus. Much of the early work revolved around understanding the architecture, access limitations, and (nominally) ethical considerations of these platforms.

Platform Capabilities and Limitations

Initial research began with TikTok, which was identified as a particularly compelling data source due to its audio-centric format and rapid, stream-like content delivery. However, attempts to access TikTok's official Research API were quickly thwarted. The API remains geofenced—inaccessible from Canada and limited to “Academic institutions in the US, EEA, UK or Switzerland” [1]—and access even then is further restricted by an application process whose timeline would not line up with the delivery of the project. This limitation effectively ruled out real-time TikTok scraping through sanctioned channels.

Workarounds were explored, including the use of third-party tools like PyTok [2], an unofficial Python library designed to navigate TikTok’s public-facing endpoints. While this allowed limited retrieval of captioned videos, and the interface left much to be desired. Raw calls to TikTok though undocumented APIs [3] similarly yielded sub-par results as the TikTok search algorithm is functionally worthless without significant time spent on the platform to inform preferences, and even then, it has been described by users as ineffectual even though sanctioned channels. [4] TikTok search returned low-value results often with no bearing on the search term and wholly polluted by tag spam. The intended keyword was only rarely found in the audio itself, if at all.

YouTube proved more reliable due to the mature ecosystem of tools surrounding it. Crowd Poetry makes extensive use of *yt-dlp* [5], a robust and well-maintained fork of *youtube-dl* [8]. This allowed for precise downloading of video content and audio extraction using scriptable options, which proved crucial for automated sampling. In tandem, *Filmot* [7] was selected as a tool to discover those clip previews and manage source metadata. Filmot is a closed-source passion project with the goal of indexing YouTube videos through their auto-generated captions. Filmot does not provide an API, but the service itself was invaluable resource to the project and the primary weight tipping the scales towards electing to use YouTube as a media source.



Filmot search results, which are ultimately scraped via BeautifulSoup.

The Role of Captioned Media

A major discovery during this phase was the utility of captioned content—particularly material with embedded subtitles or user-added transcripts. Captions served as both an entry point for linguistic parsing and as a way to semantically filter audio material. In early tests, captioned TikToks and YouTube videos were algorithmically prioritized based on their lexical overlap with user-submitted poems. This concept—aligning raw audiovisual material with poetic input via caption matching—emerged as a technically achievable and aesthetically rich compromise, allowing the system to generate a form of "semantic resonance" without needing full NLP-based video comprehension.

The Legal, Ethical, and Practical

The project encountered early and persistent tensions between artistic intent and the limits imposed on the closed platforms hosting content for this project's use. Scraping publicly available media for non-commercial artistic transformation arguably falls under fair dealing (in Canada) or fair use (in the U.S.), particularly when clips are heavily decontextualized and repurposed. However, no firm legal precedent exists for autonomous systems that remix internet content in real time for generative art purposes, and no provisions are made in any public API.

Videos were never stored in their entirety. Instead, a small number of seconds—often between 3 and 7—were sampled, processed, and sonified without visual display. As a result, what remained was sonic texture, not representation. This detachment became part of the project's aesthetic signature: samples were heard but not seen, detached from their original meaning, turned into anonymous fragments of the broader internet pulse.

The use of YouTube downloading transparently runs afoul of Alphabet Inc.'s ToS, and the platform actively fights against this library and others like it. As such, the technical implementation requires constant updates to its libraries to keep functional. There is nothing illegal about this practice, but as it stands it remains burdensome to technically creative efforts such as this one.

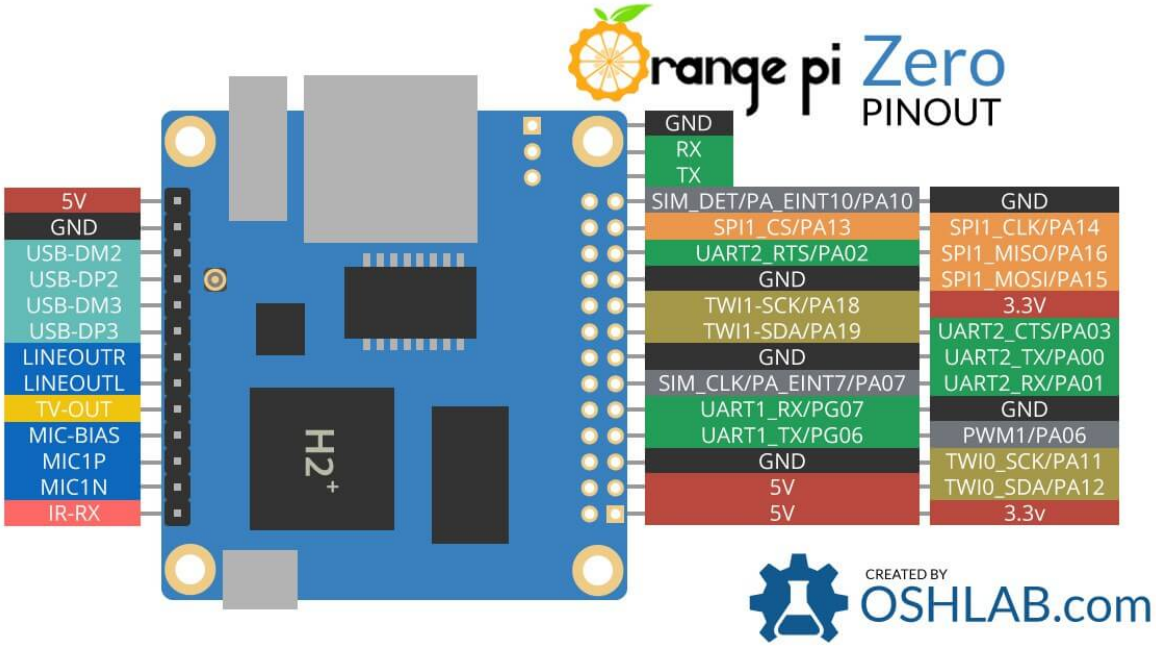
Summary

This phase of the project uncovered two central themes: first, that technical feasibility is deeply intertwined with platform politics, and second, that sampling from the internet requires constant negotiation, especially when operating in an automated, generative mode. Later in development the very location of the requests (hosted remotely in an effort to reduce SBC overhead) were algorithmically blocked. Despite barriers, workable pathways were established using reliable tooling like *yt-dlp*, lightweight caption analysis, and principled limitations on data retention and reuse. These solutions informed both the system's architecture and its broader aesthetic, helping ensure the final instrument remained playable, provocative, and sustainable.

Initial hardware mockups, featuring an embedded speaker and ePaper display readout of the current poem.

Hardware Exploration and Evolution

Taking stock of the devices available to me for immediate development, I initially planned to use the Olimex A20-OlinuXino-LIME2 [8], an industrial-applications-rated Linux SBC, but it quickly proved incapable of running *PyTok*, which bode poorly for the rest of the planned features. The decision to use a Linux machine came about as a response to the early realization that heavy API use was going to require some sort of server anyway (when the project was still entertaining an *ESP32-ADF* embedded audio platform). Had that approach been possible, the software/firmware development of the project would have leaned closer to an initial notion of developing an embedded instrument not unlike a sampler with extra web functionality. As uncomfortable as it was, I had worked with Linux audio before, and so the task was not as daunting as it may otherwise appear to be. Realtime effects and processing were however still a challenge.



The Orange Pi zero, which communicated over serial to the Teensy

Teensy + Orange Pi

Also at hand was a first-generation Orange Pi Zero, a small Linux board with WiFi, which was desirable as I still had some notions of developing this as a portable instrument, and the diminutive form factor was very attractive for enclosure considerations.
Unfortunately, considerable time was spent trying to get the board to first interface with a Teensy 4.0 + Audio shield [9], as I was already aware that offloading audio processing was going to be the ideal path for performance of the underpowered device.

Getting the Zero to load audio samples onto the Teensy was *not* progressing at any real rate, and so out of consideration for the timeline at hand was abandoned. The Teensy Audio Shield was attractive as an option as I also had experience using it, and it came with an integrated

DAC and a very intuitive audio pipeline interface [10]. This could've achieved bare-metal performance and separated concerns between data-fetching and playback to a very clean degree.

When that failed, I first thought to move the audio output to the Zero completely, which whilst that device was not ultimately used, it informed the final arrangement of concerns. I switched tack to developing software implementations of the audio pipeline.

In switching, I ran into further hurdles related to the Zero not having a realtime kernel, and JACK2 (an audio driver standard) support being spotty at best. Even with on-device compilation, exploring web and rust options, hardware limitations left me again moving towards the board I initially Envisioned but didn't have access to from the start: a Raspberry Pi. Glicol [11], Node Web Audio [12], raw ALSA calls, SuperCollider [13], and PureData [14] were tried on the Zero before settling on PureData on the Pi.

The Pi 4B is a capable device with an integrated audio jack, USB 3 and enough horsepower to run realtime audio (with numerous tweaks). The lack of integrated WiFi changed the nature of the expected enclosure somewhat, where I was then closer to envisioning a stationary installation. Upon getting a rather total crash on the SD card running headless Armbian, I switched to an SSD, further growing the minimum size of the device.

Audio Backend and Processing Pipeline

One of the most technically demanding and creatively defining aspects of *Crowd Poetry* was the design and stabilization of its audio backend. The system needed to support flexible, dynamic sample playback driven by user-submitted poems and live content from the web, while remaining lightweight enough to run on constrained hardware such as a Raspberry Pi. This required a hybrid architecture combining Python for control logic and text processing, with real-time audio managed by Pure Data (Pd) and eventually SuperCollider (SC).

Speech-to-Text: Whisper vs. Vosk
At the heart of the backend was the need to transcribe internet-sourced video and audio clips, transforming spoken content into text that could be matched against user-submitted poems. The initial plan was to use OpenAI's Whisper [15], which provides highly accurate transcription across many languages. However, Whisper's heavy dependency on the PyTorch ecosystem or its paid, cloud counterpart became a roadblock on the Raspberry Pi 4. Even using the smallest models, performance lagged and system compatibility was inconsistent.

```
s.waitForBoot {
    ~maxBuffers = 10;
    ~buffers = Array.fill(~maxBuffers, { Buffer.alloc(s, 1, 1) });
    ~bufIndex = 0;

    // Granular SynthDef
    SynthDef(\grainPlay, {
        |buf, rate = 1, dur = 0.1, density = 15, amp = 0.5, pan = 0.0|
        var trig, pos, sig;

        trig = Impulse.kr(density);
        pos = TRand.kr(0, BufDur.kr(buf), trig);

        sig = TGrains.ar(
            2,
            trig,
            buf,
            rate,
            pos,
            dur,
            pan,
            amp
        );

        Out.ar(0, sig);
    }).add;

    SynthDef(\bedlayer, {
        |buf, amp = 0.2, rate = 0.05, cutoff = 500|
        var sig;

        sig = PlayBuf.ar(1, buf, BufRateScale.kr(buf) * rate, loop: 1);
        sig = LPF.ar(sig, cutoff);  // Low-pass to unify tone
```

SuperCollider synth script

Instead, Vosk [16] was adopted as a pragmatic alternative. While slightly less accurate in raw transcription, Vosk offered offline capability, lightweight footprint, and compatibility with ARM systems, which allowed real-time operation on the Pi without GPU acceleration. The trade-off proved acceptable: since the project emphasized aesthetic transformation over linguistic fidelity, occasional mis-transcriptions were re-contextualized as poetic texture rather than seen as failures.

Audio Slicing and Labeling

Once caption-aligned or transcribed content was acquired, it needed to be sliced into small, meaningful audio segments. A series of Python routines were built to:

- Convert downloaded clips into mono WAV format;
- Use heuristics to slice audio into 2–7 second samples;
- Export each sample as its transcription target word and source video, saving it to an internal SQLite database.

These labeled samples were then evaluated for lexical overlap with the user's poem. A simple lookup for matching words or phrases then queued these for playback or triggered a search for a new sample if absent. This process allowed the soundscape to remain tightly thematically coupled to user input while still retaining a degree of entropy and abstraction.

OSC Control Flow: Python to Pd/SC

For real-time interaction, Open Sound Control (OSC) was used to bridge the gap between Python (logic, text processing, file management) and the audio engines (first Pure Data, later SuperCollider). This interface allowed Python to:

- Dynamically assign file paths to buffers or tables;
- Trigger sample playback with positional/volume parameters;
- Queue or rotate samples on command.

In the Pure Data prototype, Python handled all non-audio logic and sent OSC messages to a headless Pd instance running a looped dispatcher, which loaded samples into a set of named tables (sample0 through sample4) and handled timed triggering. Python could also mute, crossfade, or replace tables in real time, creating a feeling of responsiveness without overloading the audio engine.

Migration from Pure Data to SuperCollider

While Pure Data proved invaluable for prototyping—especially when using PlugData on macOS—it introduced several headaches on the Zero and then the Raspberry Pi; The Pi Zero required custom compilation of Pd from source due to missing dependencies in available package builds and transitioning from PlugData patches (which often bundled GUI features and nonstandard objects) to bare Pd required significant refactoring at every step. Lastly, Pd's audio engine, while robust, exhibited clicking artifacts and latency issues under load, particularly when triggering multiple tables simultaneously with varied pitch/volume envelopes.

SuperCollider was ultimately adopted as the final audio backend. Though the learning curve for SuperCollider's sclang language was steeper and less visually intuitive than PureData, it offered vastly improved audio quality, sample streaming, and granular control over envelopes, buffers, and playback curves. Once the OSC interface was re-established in SC, real-time triggering from Python became more precise and expressive. The scripting environment also made it easier to implement higher-level features like looper subroutines and dynamic crossfades.

Final JACK Configuration and Integration

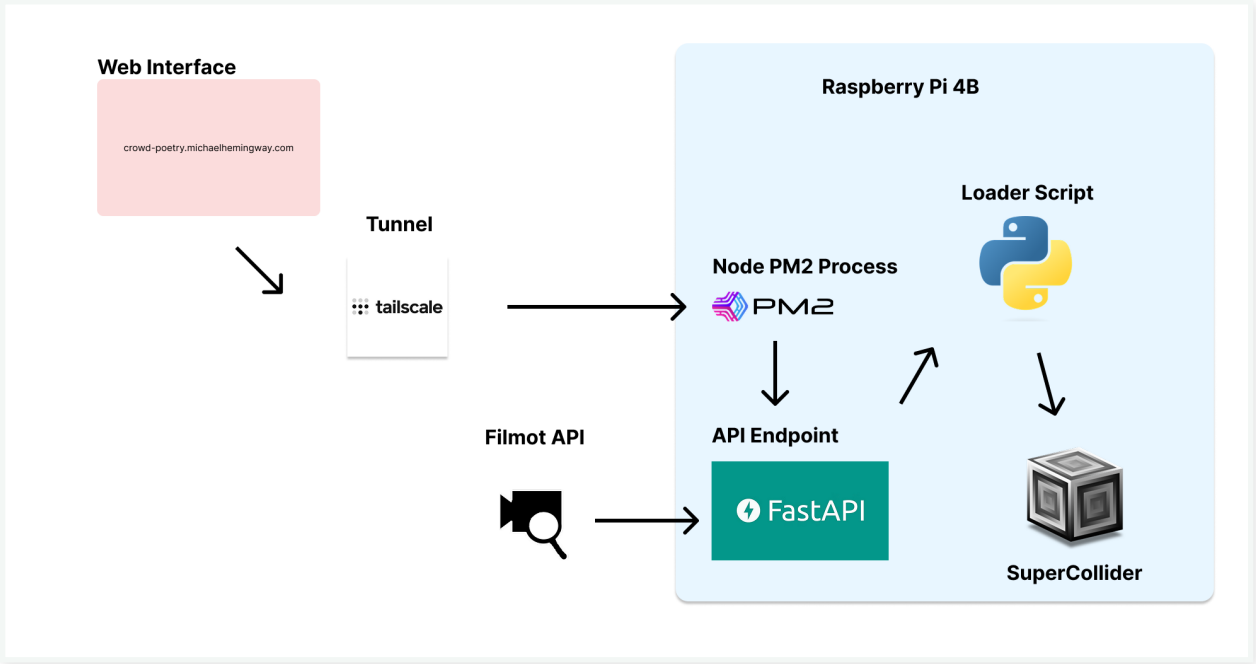
To coordinate audio routing across the Pi's lightweight Linux environment, JACK (Jack Audio Connection Kit) was used as the master audio server. After experimenting with alsa_in and alsa_out, a stable configuration was established using jackd as the primary server, auto-launched via systemd, SuperCollider configured to boot its server headless and connect to JACK and finally Python handling all high-level event timing, running as a service.

This final setup allowed the system to run headlessly without the overhead of a desktop interface. An OLED or e-paper display was planned but ultimately not implemented. Once booted, the Pi would listen for poem input, generate and trim samples, and trigger their playback in real time, all orchestrated through OSC with minimal overhead.

Final 3D printed enclosure

Results and Observations

The final audio system proved robust and expressive. The use of captioned samples, processed and dynamically layered, produced audio compositions that often felt surprising and emergent. The switch to SuperCollider notably improved sound fidelity and responsiveness, particularly under load. Perhaps most importantly, the control scheme allowed the system to remain modular: additional interaction models (e.g., CV input, touch, or MIDI) could be easily added by patching into the OSC layer, extending the lifespan and creative reach of the instrument.



The rough system architecture, noting the wealth of embedded services. (Not pictured: SQLite & BeautifulSoup)

Final Architecture

By the end of the development cycle, Crowd Poetry had evolved into a distributed, modular system integrating web-based interaction, real-time backend processing, and live audio synthesis—all orchestrated on lightweight hardware with cloud-accessible control. The final architecture was shaped as much by the constraints of embedded systems and public APIs as by the aesthetic and poetic goals of the instrument.

Overview: From Poem to Playback

The system was designed with this interaction path:

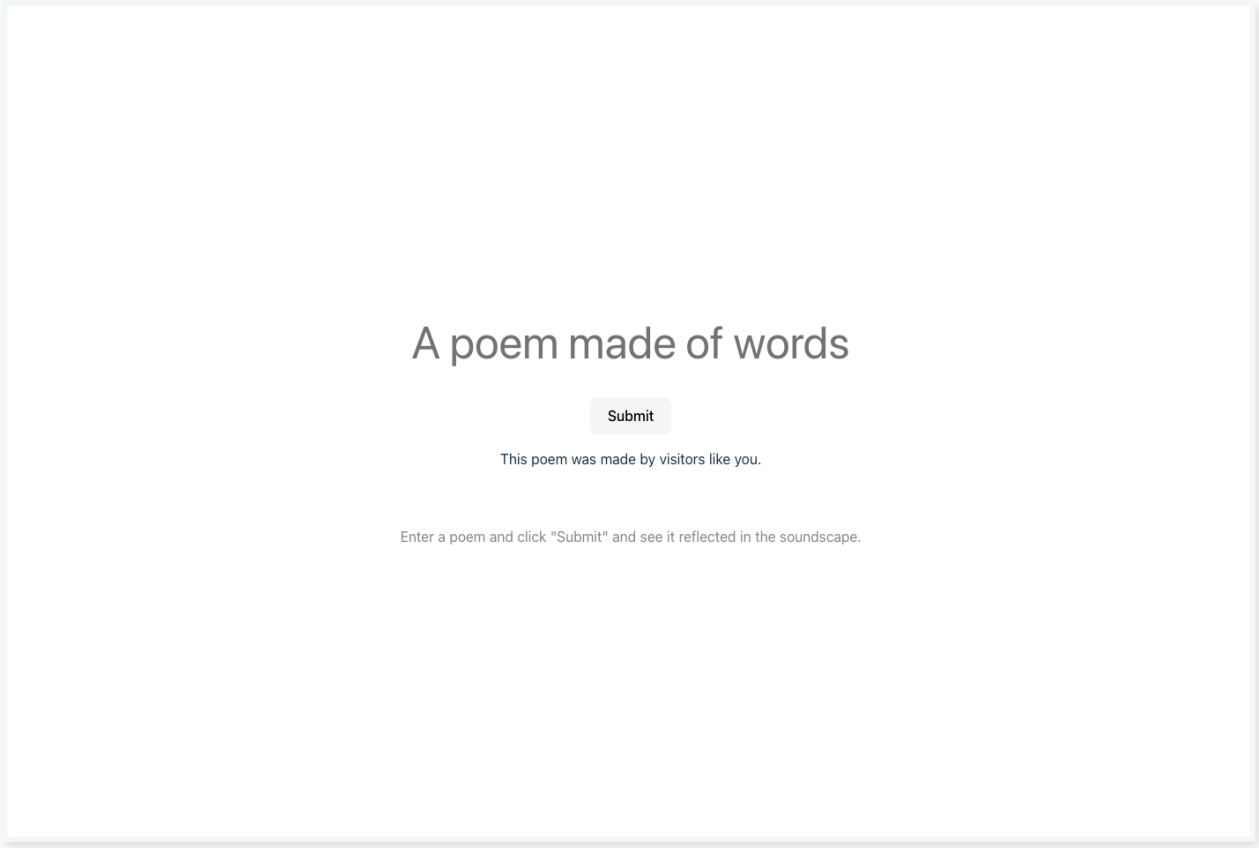
1. Poem Submission: A user accesses a simple web interface and submits a short poem.
2. Keyword Extraction and Fetch: The backend parses the poem, slicing every word into a Filmot search.
3. Download and Transcription: Relevant media is downloaded using yt-dlp or other scraping tools (e.g., pyktok, filmot), then transcribed using Vosk unless it already exists in the known words database
4. Sample Serving and Playback: Relevant samples are served locally and queued for real-time playback via OSC in SuperCollider.

Frontend and Interaction Model

The control surface for the system was a minimal web frontend built with HTML and JavaScript (React & Vite), allowing users to submit text and trigger new sessions. This web app communicated with a FastAPI backend, which served multiple roles:

- Handling poem input and initiating fetch/transcribe pipelines;
- Serving audio samples over HTTP for review/debugging;
- Exposing endpoints for triggering playback externally (e.g., /current, /search?id=…).

To keep the device responsive and always-on, the API and worker services were launched and supervised using pm2 [17], a Node.js process manager that provided logs, restarts, and status monitoring via a simple command line interface.



Screenshot of the live website, without an active poem loaded.

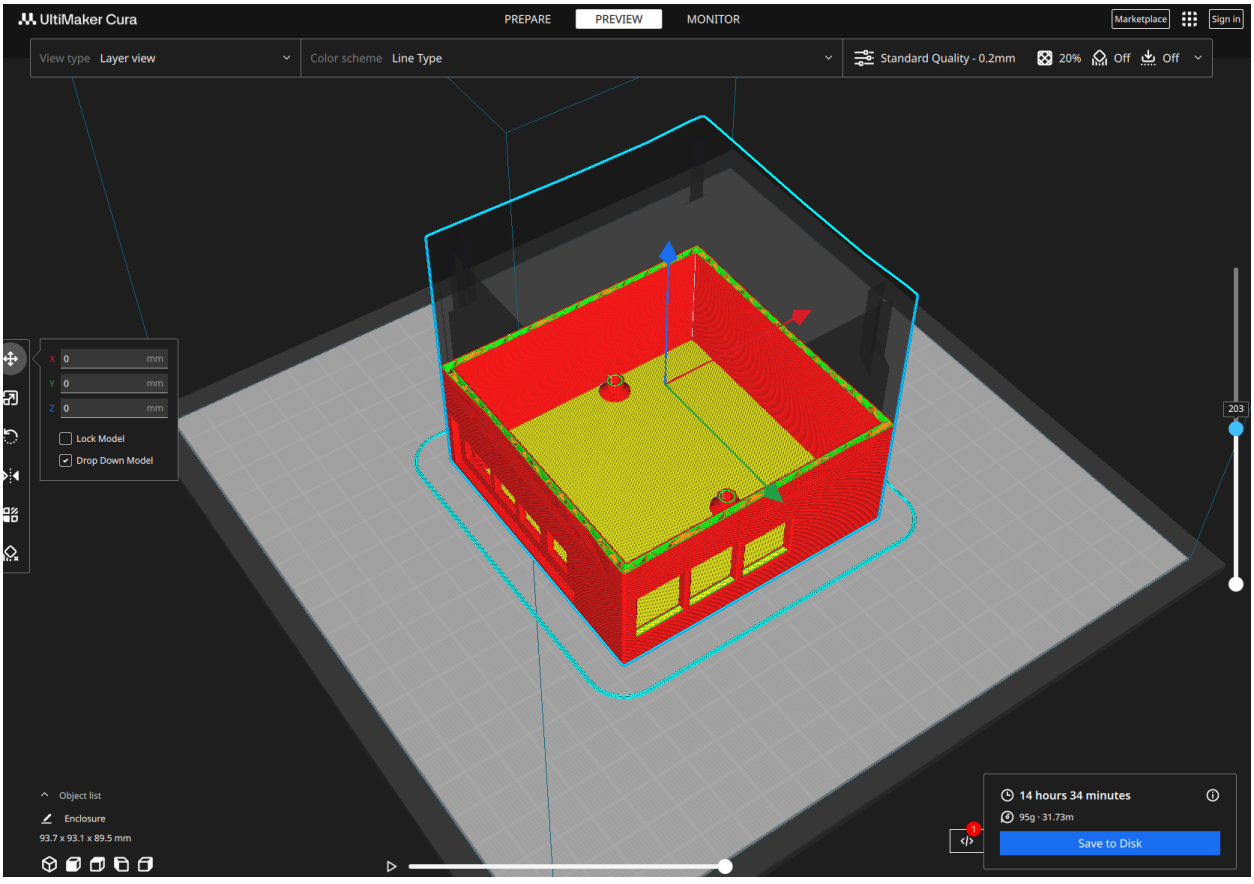
To enable the Pi to act as a remote server where ever it was, the board was enrolled in a Tailscale VPN “Funnel” [16], which allowed secure access to an exposed port 8000 to its web interface and SSH without otherwise exposing public ports. In practice, this arrangement was the key to getting past Cloudflare IP blocks when hosted elsewhere, provided it was connected to a residential ISP. YouTube downloading and *Filmot* would not work otherwise when hosted remotely.

Design Thinking and Iteration

The visual and functional design of *Crowd Poetry* was deeply intertwined with its sonic and conceptual identity. From the start, the goal was to create a device that felt as much like a sculptural object as it did a tool—part archive, part oracle. This guided early decisions around form factor and interaction design: a minimalist enclosure, a small OLED display for sparse textual feedback, and a single illuminated control that pulsed in time with the audio, offering rhythm as its primary feedback mechanism.

Aesthetic vs. Technical Fidelity

A key design goal was to maintain poetic ambiguity in how samples were presented. To this end, samples were layered with variable delay and pitch, partially randomized, and often triggered from unexpected segments within the source material. The result was a collage-like texture, intended to feel emergent rather than programmed. Despite the complexity of underlying systems—fetching, transcription, pruning—the final auditory result needed to remain gestural, not precise.



3D Printing the enclosure, seen here within the Cura slicer software

This aesthetic stance occasionally clashed with technical realities. For instance, using short, intelligible samples required tighter transcription and slicing tools than initially expected. Several iterations of sample labeling were tested before the match quality felt satisfactory.

Iterative Detours and Abandoned Tools

Several promising technologies were explored and ultimately set aside. *Glicol*, a graph-based live coding language, offered expressive audio graph construction but struggled under CPU constraints on the Pi and lacked OSC integration at the time. *PlugData* proved invaluable for prototyping on macOS, with its tight visual feedback loop, but proved brittle when transitioning to vanilla Pure Data on the Raspberry Pi. Incompatible externals, GUI reliance, and startup lag made PlugData ill-suited for deployment. Audio transfers between Python and Pure Data introduced latency and instability early in the build. Some delays were ultimately never overcome and the system isn't technically truly realtime, but file serving and playback coexisting on the same device was a takeaway, as serving files remotely during hosted attempts was not fruitful.

Reflection: Design Freedom vs. Technical Risk

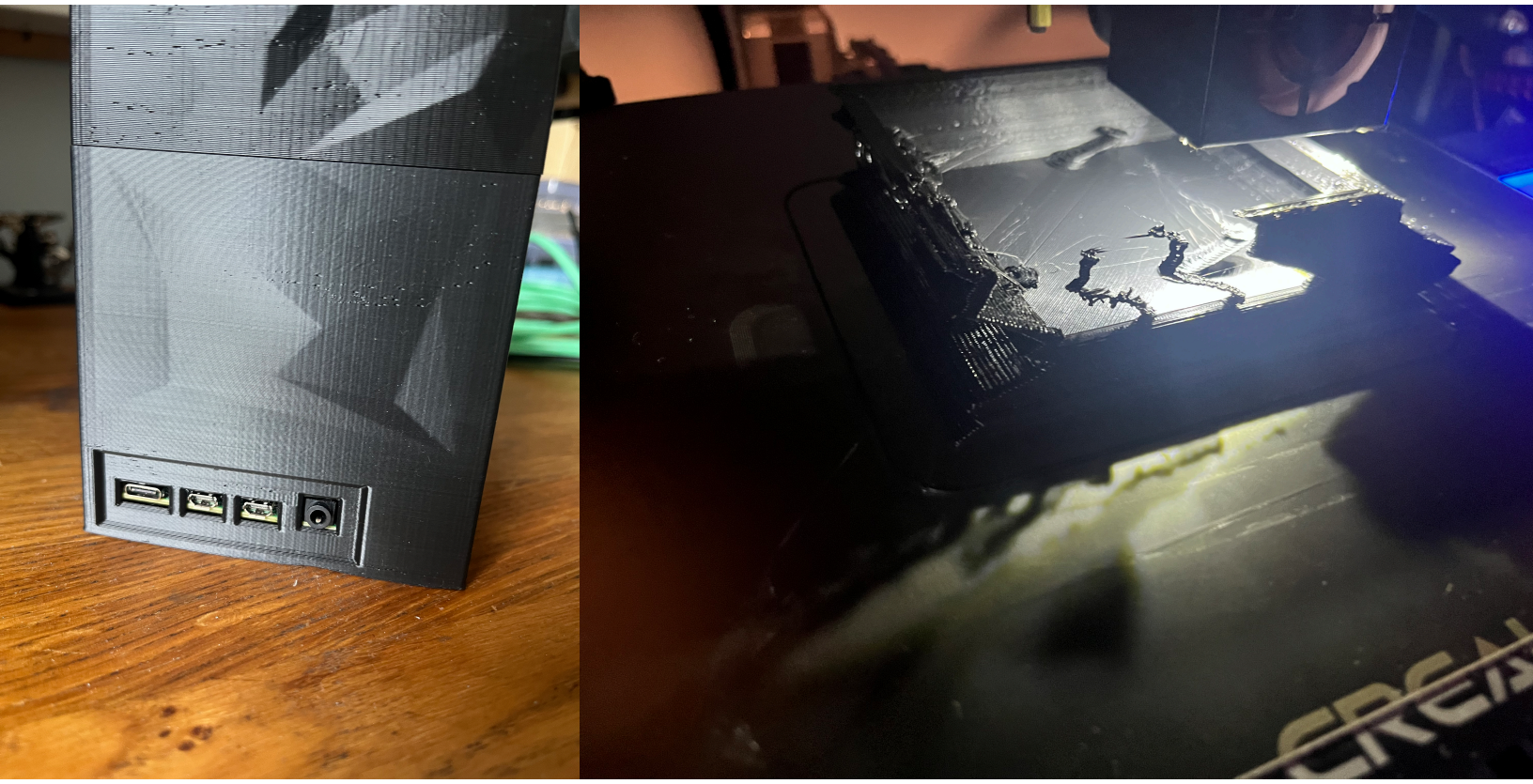
Throughout the build, there was a consistent tension between openness to poetic interpretation and the rigidity of embedded systems. Several moments required stepping back from ideal designs to preserve system reliability. However, these limitations also became a kind of creative constraint: the decision to simplify the frontend, limit playback to a few simultaneous samples, or reduce reliance on external APIs all reinforced a sense of containment.
This project affirmed that the most interesting work often happens at the edge of what's possible with the tools at hand. It also underscored the importance of iteration and realistic scope, as regrettably much of the artistic and wholly deliberate portion of this work was available to me only after a monumental technical challenge.

Conclusion and Future Work

Crowd Poetry was a study in constraints—technical, aesthetic, and conceptual. Working at the intersection of real-time media, embedded systems, and poetic interface design surfaced recurring lessons about fragility, latency, and the unpredictability of both networks and language. Creating a self-contained instrument that navigated these challenges in real time—ingesting web data, transforming it, and outputting audio with minimal user intervention—required not only a layered technical stack but also a philosophy of improvisation and approximation.

One of the most valuable outcomes of this project was a deeper understanding of the limits and affordances of real-time embedded systems. Building for the Raspberry Pi meant rethinking assumptions around memory, concurrency, and audio stability—choices that would be invisible in a cloud-based or desktop environment became central design constraints. This forced a tight coupling between backend logic and expressive outcome, leading to a kind of functional minimalism: every part of the pipeline had to earn its place.

From a poetic and interface perspective, Crowd Poetry also posed an open question: how much context does a user need for an experience to feel meaningful? The audio textures generated by the device are not always legible, but they are evocative.



LEFT: The enclosure from the side with an audio jack; the structure currently requires external output. RIGHT: An unfortunate print error, later discovered to be due to an overloaded Octoprint.

Future Directions

There is significant room for expansion and refinement. A more sophisticated search and content ranking system, potentially using embeddings or semantic similarity models, could dramatically improve the relevance of sampled material. Enabling multi-user or networked input, where poems or data streams from multiple sources modulate the same soundscape, would heighten the collective aspect of the piece. Similarly, live audience interaction via websockets or the like could further dissolve the boundary between performer and listener, turning Crowd Poetry into a true platform for participatory soundmaking.

Hardware-wise, integrating a more powerful compute platform or offloading heavier tasks to the cloud could enable real-time video slicing, richer audio layering, and dynamic sample morphing without sacrificing responsiveness, but will surely run up against bot mitigations faced during development without clever workarounds.

Looking Ahead

More broadly, Crowd Poetry invites speculation on what networked poetic instruments might become. Could they serve as archival witnesses, capturing cultural rhythms in real time? Could they function as generative collaborators, remixing and re-contextualizing language across time zones and dialects? The current device is an artifact, bounded and specific. But the concept has extensibility. In an era where language, media, and memory are increasingly mediated by machines, Crowd Poetry is a gesture toward making that mediation audible, and perhaps, in its own way, legible.

When the Pi is online, the poem will be editable at ***crowd-poetry.michaelhemingway.com***

References

1. TikTok Research API; https://developers.tiktok.com/products/research-api/
2. PyTok 1.0.0; https://pypi.org/project/PyTok/
3. How to Access And Use the TikTok API; https://geekblog.net/blog/how-to-access-and-use-the-tiktok-api
4. TikTok's suggested search problem; https://embedded.substack.com/p/tiktoks-suggested-search-problem
5. yt-dlp; https://github.com/yt-dlp/yt-dlp
6. YouTube-dl; https://en.wikipedia.org/wiki/Youtube-dl
7. Filmot; https://filmot.com/about
8. Olinuxino LIME2; https://www.olimex.com/Products/OLinuXino/A20/A20-OLinuXino-LIME2/open-source-hardware
9. Teensy Audio Shield; https://www.pjrc.com/audio-shield-for-teensy-4-0/
10. Audio Design on the Teensy; https://cdn.hackaday.io/files/8292354764928/workshop.pdf
11. Glicol; https://glicol.org/
12. Node Web Audio API; https://github.com/audiojs/web-audio-api
13. SuperCollider; https://en.wikipedia.org/wiki/SuperCollider
14. PureData; https://en.wikipedia.org/wiki/Pure_Data
15. OpenAI Whisper; https://openai.com/index/whisper/
16. Tailscale Funnel; https://tailscale.com/blog/introducing-tailscale-funnel
17. PM2; https://pm2.io/

Thanks!