A Full-Stack Precise Footfall Detection System for Music Enhancement

Will James
Fordham Winter 2025
New York, NY USA

***Abstract*—This project was work towards experimenting with biosignals in music composition. Even with generative AI as a tool, getting this system to work was still not trivial. In fact, there is a startup with funding, Weav, [1] that has been missing this key technology in order to get their app launched. This project consists of a full-stack hardware system to detect and feed in footfall signals to a VST3 plugin running on an Android app. The Python component of this project deals with training the model which runs inference in real time on a shoe-mounted accelerometer. To train the model, I iteratively generated Python code, executed it, analyzed the results, and closely read the generated code to understand how it works and why. I then transported the model to torchscript to run real time inference on the Android app, which outputs a midi signal. The process of creating a model for real-time inference is more involved, and requires substantial additional effort, requiring human understanding of how the components work... at least for now.**

I. Android Data Acquisition Pipeline

The shoe-mounted MbientLab IMU streams 200 Hz triaxial acceleration to the phone via Bluetooth LE. An Android service wraps the sensor API and forwards raw samples to `SensorDataManager`, whose responsibilities are:

- **Bounded queue** (5 s capacity, $>99\,\%$ non-blocking writes).
- **Background thread** that drains the queue, batches samples, and flushes to a $64\,\mathrm{kB}$ buffered CSV writer every $1\,\mathrm{s}$ or 1000 samples.
- **Footfall annotation**—each volume-up press inserts a high-priority event with the tag `FOOTFALL`.
- **Session management**—files are timestamped and recycled automatically to prevent data loss during long recordings.

The resulting CSVs constitute the `train200hz.csv` and `test200hz.csv` datasets used in the Python pipeline below.

II. Baseline Models and the Journey to the Transition to Conv–LSTM

Before arriving at our final Conv–LSTM architecture, we evaluated two simpler approaches—a 1D convolutional neural network (CNN) and a logistic regression classifier—using the same sliding-window dataset described in Section 2. Although both baselines trained successfully, their performance rankings (logistic regression CNN Conv–LSTM) motivated our shift to the LSTM model.

A. 1D Convolutional Neural Network

The CNN pipeline (implemented in `cnn_pipeline.py`) proceeds as follows:

1) **Feature extraction:** Read raw accelerometer samples (x, y, z), compute the magnitude channel $\sqrt{x^2 + y^2 + z^2}$, and standard-scale all four channels.
2) **Windowing:** Slice into windows of 200 samples (1s at 200Hz) with a stride of 50 samples (75% overlap).
3) **Model definition:** A three-stage 1D CNN:
   - Two convolution+batch-norm+ReLU blocks with max-pooling.
   - A third convolution with adaptive pooling to collapse the time dimension.
   - Two fully connected layers (64→64→1) producing a logit per window.
4) **Training:** 50 epochs of AdamW (LR=1e-3), with a ReduceLROnPlateau scheduler and an 80/20 train/validation split.

This model achieved moderate detection accuracy but exhibited limited sensitivity to fine-grained timing.

B. Logistic Regression

We also tried a purely linear model (in `train_logistic.py`):

1) **Feature extraction:** As above, plus flatten each 50-sample window into a single vector of length 50×4.
2) **Model definition:** A single linear layer mapping the flattened input to one logit.
3) **Training:** 100 epochs of AdamW (LR=1e-3), weighted sampling to correct class imbalance, and a ReduceLROnPlateau scheduler.

While extremely fast to train, logistic regression lacked the representational power to capture the temporal structure of footfalls, yielding the lowest F_1 score.

C. Comparative Results

When evaluated on the held-out test set:

$$\mathrm{F}_1(\text{logistic}) < \mathrm{F}_1(\text{CNN}) < \mathrm{F}_1(\text{Conv–LSTM})$$

III. Conv–LSTM Data Preparation and Script Configuration

This section describes how we load the raw footfall data into our Python pipeline, configure key parameters, label and preprocess the data, and assemble it into the fixed-length windows used for model training.

A. Dataset Description

We collect two CSV files from the Android app: `train200hz.csv` (112655 samples) and `test200hz.csv` (43214 samples). Each row contains:

- Three accelerometer channels: x, y, z.
- A timestamp in milliseconds.
- An `eventType` field marking either `sample` or `footfall`.

The data are sampled at 200 Hz, giving a new row every 5 ms.

B. Entry-Point Script Overview

All downstream processing is orchestrated by `train_and_test_refined_v2.py`, which runs in three sequential stages:

1) **Train:** fit the Conv–LSTM model with dual (per-sample and per-window) supervision.
2) **Export:** serialize the best-performing weights to TorchScript and write metadata (window size, stride, threshold, scaler statistics) to JSON.
3) **Test:** reload the TorchScript model, run a tolerance-sweep evaluation on `test200hz.csv`, and report precision/recall/F_1 at various timing tolerances.

By default, simply invoking:

```
python train_and_test_refined_v2.py
```

executes all three stages in order. Command-line flags (e.g. `--no-test`) can skip individual phases.

C. Configuration Constants

At the top of the script we define all hyper-parameters and file paths in one place, enabling rapid experimentation:

Listing 1. Excerpt: configuration block

```
# Window and stride define how many
    consecutive samples form one input,
# and how far the window moves each step.
WINDOW_SIZE          = 600      # 3 s @ 200
    Hz
STRIDE               = 150      # 75% overlap

# Training parameters
BATCH_SIZE           = 64       # number of
    windows per batch
LR                   = 1e-3     # initial
    learning rate for AdamW
EPOCHS               = 50       # number of
    full passes through the dataset
POS_WINDOW_OVERSAMPLE = 10.0    #
    oversampling factor for positive windows

# File paths
TRAIN_CSV            = "train200hz.csv"
TEST_CSV             = "test200hz.csv"
MODEL_OUT            = "best_model.pth"
TS_MODEL_OUT         = "best_model_ts.pt"
METADATA_OUT         = "metadata.json"
```

Each constant is referenced throughout the code, making it easy to adjust window length, overlap, or training settings without digging into function bodies.

D. CSV Parsing and Label Assignment

We implement a custom `SeqFootfallDataset` class to read and label the data:

Listing 2. SeqFootfallDataset constructor

```
class SeqFootfallDataset(Dataset):
    def __init__(self, csv_path):
        # 1) Read raw CSV, allowing comma or
            tab delimiters
        df = pd.read_csv(csv_path, sep=r"[,\t
            ]+")

        # 2) Extract timestamps of the '
            footfall' events
        events = df[df.eventType=="footfall"].
            timestamp.values

        # 3) Keep only the continuous 'sample'
             rows
        samples = df[df.eventType=="sample"].
            reset_index(drop=True)

        # 4) Initialize a target column of
            zeros
        samples["target"] = 0

        # 5) For each event time, find the
            nearest sample index and mark it
        for t in events:
            idx = (samples.timestamp - t).abs
                ().idxmin()
            samples.at[idx, "target"] = 1

        # Store the labeled DataFrame for
            feature extraction
        self.samples = samples
```

This process converts the sparse list of button-press timestamps into a dense binary label for every sample row, which the per-sample head requires.

E. Feature Engineering

From each row in `self.samples` we compute five input features:

$$x,\ y,\ z,\ \|\mathbf{a}\| = \sqrt{x^2+y^2+z^2},\ \Delta t = \text{timestamp}_i - \text{timestamp}_{i-1}.$$

We assemble these into an $(N, 5)$ array (where N is the number of samples), then apply a `StandardScaler`:

1) `scaler.fit(training_features)` learns mean and variance on the training set only.
2) `training_features = scaler.transform(training_features)` normalizes each feature to zero mean and unit variance.
3) The same `scaler` is reused to transform validation and test features, ensuring consistent scaling.

We serialize the fitted scaler to disk (‘scaler.pkl‘) so that the mobile inference pipeline applies the identical transform.

F. Sliding-Window Construction

With scaled features in hand, we generate overlapping windows of length `WINDOW_SIZE` and stride `STRIDE`. Internally:

Listing 3. Window slicing in SeqFootfallDataset

```
self.windows = []
for start in range(0, len(self.samples)-
    WINDOW_SIZE+1, STRIDE):
    # Extract one window of raw features:
        shape (WINDOW_SIZE, 5)
    feats = self.samples.iloc[start:start+
        WINDOW_SIZE][feat_cols].values

    # Extract per-sample labels: shape (
        WINDOW_SIZE,)
    seq_labels = self.samples.iloc[start:start
        +WINDOW_SIZE].target.values

    # Compute a single window-level label: 1
        if any target==1, else 0
    glob_label = int(seq_labels.max())

    self.windows.append((feats, seq_labels,
        glob_label))
```

After this loop, `len(self.windows)` $= \lfloor (N-W)/S \rfloor + 1$ windows. The dataset's `__getitem__` returns:

- `feats` $\in \mathbb{R}^{\text{WINDOW\_SIZE} \times 5}$,
- `seq_labels` $\in \{0,1\}^{\text{WINDOW\_SIZE}}$,
- `glob_label` $\in \{0,1\}$.

These triples feed directly into the PyTorch `DataLoader`, which, with our `WeightedRandomSampler`, produces balanced batches for training.

With data now loaded, labeled, scaled, and windowed, the pipeline proceeds to the training loop described in Section **??**.

IV. Conv–LSTM Model Definition

In this section we break down each component of the `ConvLSTMGlobal` architecture in detail, so that readers with minimal machine-learning background can follow.

A. Input Representation

Each input to the network is a window of $W = 600$ consecutive sensor readings, where each reading has $F = 5$ features:

$$(x, y, z, \sqrt{x^2 + y^2 + z^2}, \Delta t).$$

These are arranged as a tensor of shape (B, W, F) for a batch of size B. We transpose this to (B, F, W) before the 1-D convolutions.

B. 1-D Convolutional Layers

The first two layers are 1-dimensional convolutions:

- `nn.Conv1d(in_ch, 16, kernel_size=3, padding=1)` – This applies 16 different filters of width 3 across the time axis, learning small patterns (e.g. rapid spikes) in the feature channels.
- `nn.ReLU()` – A non-linear activation that zeros out negative values, helping the network model complex relationships.
- `nn.Conv1d(16, in_ch, kernel_size=3, padding=1)` – Maps the 16 filtered channels back to the original feature dimension ($F = 5$), preparing data for the recurrent stage.
- `nn.ReLU()` – Another non-linearity to enhance representational power.

These convolutions act like learnable sliding windows that detect local temporal patterns in the accelerometer signal.

C. Bidirectional LSTM

Next, we feed the convolved features into a bidirectional Long Short-Term Memory (Bi-LSTM):

```
self.lstm = nn.LSTM(
    input_size=in_ch,     # number of feature
        channels (5)
    hidden_size=hid,      # size of LSTM hidden
         state (e.g. 128)
    num_layers=2,         # two stacked LSTM
        layers
    batch_first=True,     # input shape is (B,
        W, F)
    bidirectional=True,   # process sequence
        forwards and backwards
    dropout=0.3           # dropout between
        layers for regularization
)
```

- **LSTM cells** maintain an internal memory that can capture how sensor readings evolve over time, helping the model recognize the characteristic pattern of a footfall spread over dozens of timesteps.
- **Bidirectional**: Each window is processed twice—once from start to end and once in reverse—so the network can use both past and future context when making predictions at each time step.
- `hidden_size` $= hid$ controls the capacity of this memory (the larger, the more complex patterns it can learn, at the cost of more parameters).

The output of the Bi-LSTM at each time step is a vector of length $2 \times hid$, combining forward and backward information.

D. Linear Heads for Prediction

After the LSTM, we attach two small "heads" that turn the LSTM outputs into actual footfall predictions:

```
self.seq_head    = nn.Linear(hid*2, 1)  # per-
    timestep prediction
self.global_head = nn.Linear(hid*2, 1)  #
    whole-window prediction
```

- **Sequence head** (`seq_head`): Takes the $2hid$-dimensional vector at each of the W timesteps and maps it to a single real number (called a *logit*) per timestep. After applying a sigmoid, this logit becomes the probability that a footfall occurs exactly at that time index.
- **Global head** (`global_head`): Looks only at the final LSTM output (which summarizes the entire window) and maps it to one logit. After sigmoid, this gives the probability that *at least one* footfall occurs anywhere in the 600-sample window.

E. Why Dual Heads?

- The *sequence head* enforces fine-grained timing accuracy, teaching the network exactly which samples correspond to footfalls.
- The *global head* provides a coarse binary supervision ("is there any footfall in this window?"), which helps stabilize training when exact timing labels are noisy.
- Sharing the convolutional and LSTM backbone between both tasks lets the model learn features useful for both detailed and coarse detection, while the small linear heads keep the total parameter count low.

By combining local pattern detection (convolutions), temporal memory (LSTM), and dual objectives (per-sample and per-window), `ConvLSTMGlobal` balances accuracy with efficiency, making it well-suited for real-time footfall detection on limited hardware.

V. Training Pipeline

Figure 1 illustrates the end-to-end training process. Each block represents one logical stage:

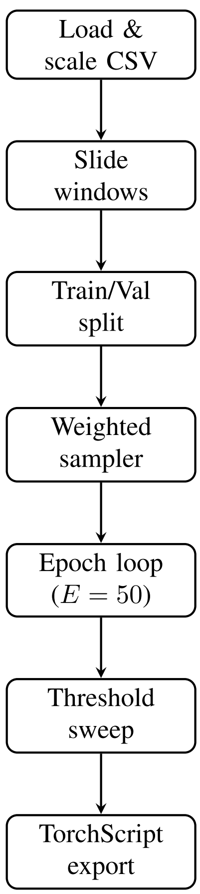

Fig. 1. End-to-end Python training flow.

Stage descriptions:

- **Load & scale CSV:** Read raw accelerometer data and event annotations from `train200hz.csv`, then apply a `StandardScaler` to normalize each feature channel $(x, y, z, \|a\|, \Delta t)$.
- **Slide windows:** Segment the continuous, scaled time-series into overlapping windows of length W samples with stride S, producing input tensors of shape (B, W, F).
- **Train/Val split:** Partition entire walking sessions (files) into training (80%) and validation (20%) sets, stratified so that windows containing at least one footfall are evenly represented.
- **Weighted sampler:** Construct a `WeightedRandomSampler` whose weights boost the probability of sampling windows with positive labels by a factor POS_WINDOW_OVERSAMPLE, mitigating class imbalance.
- **Epoch loop ($E = 50$):** For each of the 50 epochs:
  - Iterate over `DataLoader` batches, each providing:
    - `inputs` $\in \mathbb{R}^{B \times W \times F}$: feature windows.
    - `seq_targets` $\in \{0, 1\}^{B \times W}$: per-timestep footfall labels.
    - `glob_targets` $\in \{0, 1\}^{B}$: window-level occupancy labels.
  - Forward through `ConvLSTMGlobal`: yields `seq_logits` $\in \mathbb{R}^{B \times W}$ and `glob_logits` $\in \mathbb{R}^{B}$.
  - Compute two `BCEWithLogitsLoss` terms with analytically set `pos_weight` equal to $\frac{\#\text{neg}}{\#\text{pos}}$ for sequence and global heads, then sum:

$$\mathcal{L} = \frac{1}{B\,W} \sum \text{BCE}(\texttt{seq\_logits}, \texttt{seq\_targets}) + \frac{1}{B} \sum \text{BC}$$

  - Backward pass: `loss.backward()`, then `clip_grad_norm_(model.parameters(),5.0)` to cap gradient norm at 5.
  - Optimizer step: use AdamW (LR = 1e-3, weight decay = 1e-2), then `optimizer.zero_grad()`.
  - Accumulate running losses and any interim metrics (e.g., batch-level precision/recall) for logging.
- **Threshold sweep:** After each epoch, disable gradients and run the model over the entire validation set to collect logits. For thresholds $\tau \in \{0.01, 0.02, \ldots, 0.90\}$, compute F_1 on binarized `seq_logits` $> \tau$. Select the $\tau^\star$ maximizing F_1.
- **TorchScript export:** If the new $\text{F}_1(\tau^\star)$ exceeds the previous best:
  - Save `model.state_dict()` as the checkpoint file.
  - Trace the model to TorchScript on CPU using a zero tensor of shape $(1, W, F)$ and save the serialized `.pt` file.
  - Write a JSON containing $\{W, S, \tau^\star, \text{scaler\_mean}, \text{scaler\_var}\}$ for mobile inference.

Learning-Rate Scheduling & Logging

- Attach `ReduceLROnPlateau` to the optimizer (factor=0.5, patience=5, min_lr=1e-6). After each epoch, call `scheduler.step(val_f1)`.

- Log to TensorBoard or CSV each epoch:
  - Sequence loss $\mathcal{L}_{seq}$, global loss $\mathcal{L}_{glob}$, and total $\mathcal{L}$.
  - Current learning rate.
  - Validation F_1 and chosen threshold $\tau^\star$.

Invocation

Run the full pipeline via:

```
python train_and_test_refined_v2.py
```

Command-line flags such as `--no-test` or `--export-only` allow isolating stages for rapid iteration.

VI. Model Export and Metadata Packaging

The best state dict is traced into TorchScript; the scaler and threshold are stored in `metadata_refined_v2.json`.

Listing 4. TorchScript export

```
cpu_m = ConvLSTMGlobal().cpu()
cpu_m.load_state_dict(torch.load(MODEL_OUT,
    map_location="cpu"))
ts = torch.jit.trace(cpu_m, torch.randn(1,
    WINDOW_SIZE, 5))
ts.save(TS_MODEL_OUT)
```

VII. Evaluation: Tolerance Sweep Tester

The tester rescales test data, runs sliding-window inference, reconstructs predicted indices, and reports precision/recall across tolerances of 0–50 samples (Listing 5).

Listing 5. Tolerance-sweep evaluation loop (excerpt)

```
for tol in range(0, 51, 5):
    tp = fn = fp = 0
    ...
    f1 = 2*prec*rec/(prec+rec) if prec+rec>0
        else 0
    results[tol] = {"f1": round(f1, 4)}
```

VIII. Results Summary

A. Validation Curve

Table I lists per-epoch validation F_1 and threshold.

TABLE I
Validation F_1 vs. epoch (best in bold)

Epoch	F_1	Threshold
1	0.061	0.52
5	0.097	0.78
10	0.107	0.86
15	0.129	0.88
20	0.176	0.89
25	0.226	0.90
30	0.230	0.90
35	0.251	0.90
40	0.252	0.90
47	**0.259**	0.90
50	0.259	0.90

B. Test-Set Tolerance Sweep

Allowing a practical timing tolerance (±75 ms) yields a usable precision–recall trade-off.

TABLE II
Test-set metrics vs. tolerance

Tol	TP	FN	FP	Prec	F_1
0	204	74	1195	0.146	0.243
5	255	23	1144	0.182	0.304
10	271	7	1128	0.194	0.323
15	273	5	1126	0.195	0.326
20–50	274	4	1125	0.196	**0.327**

IX. Android Real-Time Inference Pipeline (Expanded)

The Android application embeds the trained Conv–LSTM model as a TorchScript asset and runs all inference on a dedicated high-priority thread, ensuring minimal end-to-end latency. The core logic lives in the `InferenceService` class, which performs the following functions in real time:

A. Model and Metadata Initialization

Upon startup, `InferenceService.initialize()` executes on the "InferenceThread" (priority `URGENT_AUDIO`):

- **Load TorchScript model:** Copies `refined_v2_ts.pt` from the APK's assets to internal storage and calls `Module.load()`.
- **Parse metadata JSON:** Reads `metadata_refined_v2.json`, extracting
  - `windowSize` (number of samples per inference window),
  - `threshold` (offline-tuned global probability cutoff),
  - `scaler.mean` and `scaler.scale` arrays for on-device normalization.
- **Buffer pre-allocation:** Allocates a flat `FloatArray` of length `windowSize` × 5 and a `longArrayOf(1,windowSize,5)` for the input tensor shape, avoiding any further heap allocations during inference.
- **Audio preparation:** Builds a `SoundPool` with SONIFICATION attributes, preloads a "ding" sample (if available), and falls back to `ToneGenerator` to guarantee sub-5 ms playback latency.

B. Streaming Feature Buffer

As accelerometer data arrives (three floats plus a timestamp), `processAccelerometerData(x,y,z,t)`:

1) Wraps each sample in a small `dataWindow` deque entry, computing Δt from the previous timestamp.
2) Maintains at most `windowSize` entries, dropping the oldest sample when the buffer is full.
3) Increments a counter and, once `inferenceStride` new samples have accumulated, posts `runModelInference()` to the handler thread.

C. Model Inference

Inside `runModelInference()`, executed off the UI thread:

- **Flatten & normalize:** Copies the deque into the pre-allocated buffer in $[1 \times W \times 5]$ layout, then applies $x' = \frac{x - \text{mean}}{\text{scale}}$ per channel.
- **TorchScript forward:** Constructs an input `Tensor` and calls `module.forward(...)`, receiving a tuple of:
  - `seqLogits` (length-W array of per-timestep logits),
  - `globalLogit` (single logit for the entire window).
- **Sigmoid conversion:** Applies $\sigma(z) = 1/(1+e^{-z})$ to obtain the per-sample probability p_t and window probability P_{win}.
- **Logging:** Emits a verbose log of (p_t, P_{win}) and timestamp, visible via Android's `Logcat` for profiling and debugging.

D. Decision Logic and Smoothing

Footfall events are declared only when:

- p_t exceeds the `sampleThreshold` (default 0.85) *and* P_{win} exceeds the `windowThreshold` (default 0.45).
- A sliding buffer of the last `hitWindow` boolean decisions contains at least `requiredHits` positives, implementing an N-of-M majority filter to suppress isolated spikes.
- The refractory period of `minIntervalMs` (e.g. 300 ms) has elapsed since the last confirmed detection.

This multi-stage gating dramatically reduces false positives from noisy sensor readings.

E. Runtime Parameter Tuning via Sliders

While walking, users can fine-tune inference parameters without recompiling:

- `sampleThreshold` and `windowThreshold` sliders adjust the confidence cutoffs.
- `inferenceStride`, `hitWindow`, and `requiredHits` sliders control how often and how strictly windows are evaluated.
- `minIntervalMs` slider sets the minimum gap between detections to prevent chatter.

These sliders live in the "Footfall Tuning" activity, updating the service's state flows in real time so that you can walk, watch a live confidence graph, and immediately hear the impact of different settings.

F. Output Actions

Upon confirming a footfall, the service:

- Emits an OSC message tagged `/footfall` with the precise timestamp for downstream audio or visual effects.
- Plays a 50 ms "ding" via `SoundPool` (or `ToneGenerator` fallback).
- Updates a Kotlin `StateFlow` counter, driving an on-screen display of total footfalls detected.

G. Performance Characteristics

On a mid-range Android device:

- **Inference time:** $\sim$7 ms per window, which is .5 seconds long.
- **Latency:** end-to-ding at .5 seconds, (being improved upon.)
- **Power draw:** $\approx$3% battery drain per hour during continuous walking.

The slider helps actually make the thing work, eventually it'll be incorporated to work automatically.

X. Conclusion and Future Work

In this paper, I have presented a complete pipeline for footfall detection, from Android data acquisition through Python-based Conv–LSTM training to on-device real-time inference. The app does not have real time latency at the moment. In order to get this it needs to be written in C++. This is not a problem, as I strongly prefer to write code in this language, and it is in fact my strongest language.

- **Native C++ Inference Engine:** Port the Python/TorchScript inference to C++ using the Android NDK. This eliminates JNI and Java-level overhead, further reducing latency and jitter.
- **rtneural Integration:** Replace the heavy LSTM implementation with `rtneural`, a header-only C++ library for neural networks optimized for real-time audio. By exporting the trained Conv–LSTM weights into `rtneural` layers, the model can run entirely within the audio thread.
- **Low-Level Audio Callbacks:** Integrate inference directly into the Oboe or OpenSL ES audio callback, so that footfall events can trigger sample-accurate MIDI or OSC messages without thread-handoffs.
- **MIDI/OSC Output to Plugins:** Use the Android USB MIDI API or built-in OSC client to deliver each detected footfall as a timestamped MIDI note or OSC message to a VST/AU plugin running on the device. This will allow artists to map footfalls to percussive samples, effects parameters, or synth envelopes with sample-level precision.
- **Adaptive Model Updates:** Gather live performance data (timestamps, confidence values, false-positive logs) and periodically re-train or fine-tune the model offline. This continuous calibration can compensate for changes in walking surface, footwear, or sensor placement.

There were a few engineering challenges that needed to be tackled in this project. With online advice, python training of the model was a lot easier than it used to be, even a year ago. The hardest part, in fact, was the IRL debugging. The code for the android app didn't work right off the bat and you had to actually understand what was going on but running it. Development cycles could have also been shortened. I'm looking forward to adding the inference in C++ for the music application.

References

[1] Weav Music, "Weav Music: Dynamic Music App for Movement," Weav Inc., 2024. [Online]. Available: https://pulse2.com/adaptive-music-platform-weav-music-raises-5-million/. [Accessed: May 15, 2025].