AURA: Adaptive Unison-Response Audio Engine

Supplementary file for the Samsung GenAI Hackathon

Team AURA

September 28, 2025

Contents

1 Executive Summary

The Adaptive Unison-Response Audio (AURA) Engine is a comprehensive framework for creating real-time, emotionally-aware generative and adaptive audio experiences. AURA demonstrates how multi-modal inputs—ranging from in-game events to the player's real-world facial expressions and speech—can transform a static piece of music into a living, breathing score that perfectly matches the on-screen action.

The project is realized through two distinct pillars:

1. **A Standalone Pygame Engine** featuring advanced on-device adaptive and generative music algorithms, most notably a state-based generative system using Markov Chains to create infinite, non-repetitive musical scores.
2. **A Web-based SDK Framework** that utilizes a suite of sensor modules (face, voice, game state) and an intelligent orchestrator to modulate music for web clients, illustrating a broader, extensible use case for the technology.

The core vision of AURA is to serve as a proof-of-concept for a technology that could be integrated into Samsung devices, enabling applications to generate personalized, royalty-free soundtracks for user-generated content by analyzing scene context and emotional cues.

2 System Architecture

The AURA ecosystem is designed as a modular, multi-layered system that separates concerns between data input, processing, and audio output. The architecture, depicted in Figure 1, consists of several interconnected components that work in concert to deliver a real-time adaptive audio experience.

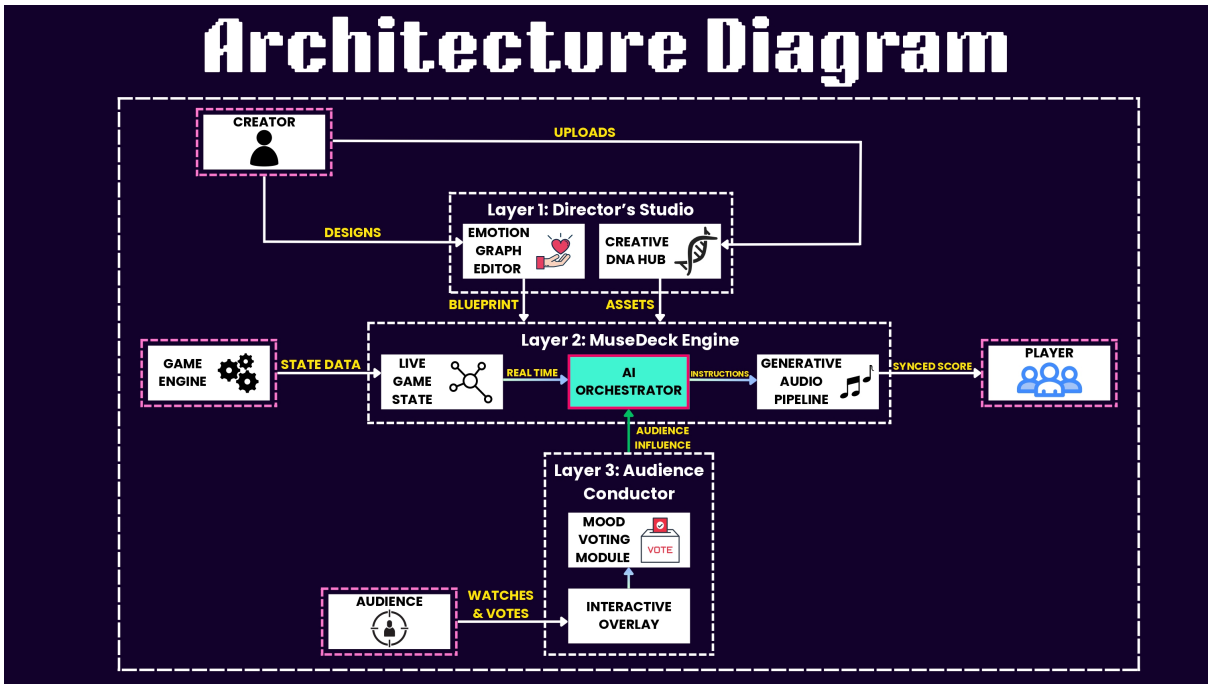

Figure 1: The High-Level Architecture of the AURA Engine.

The primary components of the web-based framework are:

- **Sensor Modules & Game Client:** These are the data sources. Python scripts using

OpenCV, DeepFace, and Librosa capture facial and vocal emotion, while a Three.js game client reports its internal state (e.g., player health, enemy proximity).

- **Backend (FastAPI):** A central hub that serves the web clients and manages real-time communication via WebSockets. It receives data from all sensor modules.
- **Emotion Orchestrator:** The intelligent core of the system. It aggregates and weighs inputs from all sources to compute a final, unified "Emotion Vector."
- **Audio Modulator:** This module takes the Emotion Vector and translates it into concrete musical parameters (e.g., tempo, filter cutoff, effects) that are sent to the client.
- **Frontend (React Director's Studio):** A comprehensive dashboard for visualizing all data streams, uploading "Music DNA" (audio files), and controlling the system.

3 Innovation and Contribution

AURA's novelty lies not in a single algorithm, but in the sophisticated synthesis of generative techniques, multi-modal sensing, and a complete end-to-end system architecture. This project makes several key innovative contributions that meet and exceed the scope of a GenAI-focused hackathon.

- **Real-time, State-Driven Generative Music:** The core GenAI innovation is the Markov Chain system within the Pygame engine. While adaptive audio switches between pre-made tracks, our system *procedurally generates* a unique, non-repetitive musical score in real-time. By dynamically switching between "calm" and "tension" transition matrices based on gameplay events, the engine creates a soundtrack that is both infinitely varied and perfectly synchronized to the player's emotional journey.
- **Multi-Modal Emotion Fusion:** AURA moves beyond simple, single-source triggers. The `Orchestrator` acts as an intelligent fusion engine, programmatically weighing and combining disparate, noisy, real-time inputs (in-game metrics, facial expressions, vocal tonality) into a single, coherent emotional vector. This is a robust approach to affective computing that creates a more nuanced and accurate representation of the user's state.
- **Dual-Pillar Demonstration:** The project's structure—a self-contained Pygame application *and* a scalable Web SDK framework—is a key strategic strength. It demonstrates the technology's versatility for two distinct commercial paths: as a high-performance, on-device library for native applications (games, creative tools) and as a distributed, service-oriented architecture for web-based and live experiences.
- **Complete End-to-End Workflow:** We have delivered a fully integrated pipeline. This includes data capture (Python sensor modules), real-time transport (WebSockets), intelligent processing (FastAPI backend with Orchestrator), and a comprehensive command center (the React Director's Studio). The inclusion of an in-browser AI music creation tool (Magenta.js) allows for a full "concept-to-creation-to-modulation" workflow within a single project.

4 Core Technologies & Implementation

AURA is built on two primary pillars, each demonstrating a different application of the core concept.

4.1 Pillar 1: The Standalone Pygame Engine

A complete 2D RPG was developed in Pygame to serve as a powerful, self-contained showcase for on-device music generation and adaptation. The player navigates a dungeon, fights enemies, and collects items. The innovation lies in its three selectable music modes.

4.1.1 Linear and Adaptive Music

The first two modes represent traditional and modern approaches to game audio.

- **Linear Music:** Plays a single, static track on a loop, regardless of gameplay context.
- **Adaptive Music:** Dynamically crossfades between pre-composed tracks based on the game state. The ‘world_manager.py‘ component tracks player proximity to enemies to transition between ”Exploration,” ”Close to Enemy,” and ”Battle” themes.

4.1.2 Generative Music via Markov Chains

This mode is the primary GenAI implementation within the Pygame engine. Instead of playing pre-recorded audio, the engine generates a continuous, non-repetitive musical score in real-time.

Algorithm: The system models chord progressions as a first-order Markov Chain. The state of the chain is the currently playing chord, and the system uses a transition matrix to probabilistically determine the next chord to play. We defined two distinct transition matrices to reflect different emotional states: a consonant, stable matrix for ‘calm‘ (exploration) and a more dramatic, dissonant matrix for ‘tension‘ (combat).

State-Based Transition Matrices: The ‘generative_music‘ method in ‘world_manager.py‘ initializes two sets of chords and their corresponding transition matrices. For example, the high-tension matrix is defined as follows:

$$\mathbf{T}_{\text{tension}} = \begin{pmatrix} 0.1 & 0.3 & 0.2 & 0.1 & 0.1 & 0.0 & 0.0 & 0.0 & 0.2 \\ 0.2 & 0.1 & 0.2 & 0.3 & 0.0 & 0.0 & 0.0 & 0.1 & 0.1 \\ 0.2 & 0.2 & 0.0 & 0.3 & 0.0 & 0.1 & 0.0 & 0.0 & 0.2 \\ 0.0 & 0.3 & 0.3 & 0.1 & 0.0 & 0.2 & 0.0 & 0.1 & 0.0 \\ 0.5 & 0.0 & 0.0 & 0.0 & 0.5 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.5 & 0.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.5 & 0.5 & 0.0 \\ 0.5 & 0.0 & 0.0 & 0.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.5 & 0.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

Real-time Generation Logic: The ‘music_generator‘ function is called every frame. It checks the current game state (‘exploration‘ vs. ‘battle‘) and selects the appropriate transition matrix. It then uses ‘numpy.random.choice‘ to select the next chord based on the probabilities in the current chord’s row in the matrix. This creates a seamless, ever-evolving soundtrack perfectly synchronized with gameplay.

```
def get_next_chord(self, current_index, transition_matrix):
    """
    Retrieves the next chord index based on the current index and
    transition matrix probabilities.
    """
    probabilities = transition_matrix[current_index]
    next_index = np.random.choice(len(probabilities), p=probabilities)
    return next_index
```

```

def music_generator(self):
    """
    Generates and plays music based on the current game state.
    """
    current_time = pygame.time.get_ticks()
    if current_time >= self.next_time:
        if self.game_state == 'battle':
            # ... (logic to switch to tension matrix)
            next_chord_index = self.get_next_chord(self.current_chord_index,
                                                   self.TURBO_tension_transition_matrix)
            self.play_chord(self.current_channel,
                   self.TURBO_high_tension_chords_mp3[self.TURBO_tension_chords
    [next_chord_index]])
        else:
            # ... (logic to use calm matrix)
            next_chord_index = self.get_next_chord(self.current_chord_index,
                                                   self.calm_transition_matrix)
            self.play_chord(self.current_channel,
                   self.calm_chords_mp3[self.calm_chords[next_chord_index]])

        self.current_chord_index = next_chord_index
        # ... (logic to schedule next chord)
```

Listing 1: Core logic for selecting the next chord in ‘world_manager.py‘

4.2 Pillar 2: The AURA Web SDK Framework

This pillar demonstrates the broader applicability of AURA as a portable, service-based technology.

4.2.1 Multi-Modal Emotion Sensing

AURA aggregates data from multiple sources to build a comprehensive emotional profile.

- **Face Emotion Sensor:** A Python script (‘face_emotion_sender.py‘) uses OpenCV to capture webcam frames and the DeepFace library to analyze facial expressions. It sends the dominant emotion and confidence score via WebSocket.
- **Speech Emotion Sensor:** A Python script (‘speech_emotion_sender.py‘) uses PyAudio to capture microphone input and a pre-trained scikit-learn model (with features extracted by Librosa) to classify the emotion in the user’s voice.
- **Web Game Client:** A simple Three.js game (‘game_client/static/js/game.js‘) sends its state (player health, enemy count, threat proximity, etc.) to the backend.

4.2.2 The Emotion Orchestrator

The ‘orchestrator.py‘ module is the brain of the system. It receives raw data from all sources and intelligently combines them into a single, coherent ‘final_emotion_vector‘.

Weighted Aggregation: The orchestrator uses a dictionary of weights to determine the influence of each input source. This allows for fine-tuning the system’s sensitivity. It also dynamically re-normalizes weights if a source becomes stale (e.g., the game client disconnects), ensuring the audio remains responsive.

```
class Orchestrator:
    def __init__(self):
        self.weights = {
```

```
            "game_state": 0.8,
            "face": 0.1,
            "speech": 0.1,
            "audience": 0.0
        }
        # ... state initialization
```

Listing 2: Weighting system in ‘orchestrator.py‘

Vector Mapping: The orchestrator translates disparate inputs into a standardized five-dimensional emotion vector: (Tension, Excitement, Fear, Joy, Calm). For example, data from the game state is mapped algorithmically: high threat proximity increases ”Fear” and ”Tension,” while high player speed increases ”Excitement.”

4.2.3 The Audio Modulator

Once the final emotion vector is computed, the ‘audio_modulator.py‘ module translates it into concrete musical parameters. It calculates a target tempo, primary emotion, and a rich descriptor of FX intents (e.g., filter cutoff, reverb mix, distortion drive). This descriptor is sent to the client, which can use the Web Audio API to apply these modulations in real-time.

4.2.4 Frontend: The Director’s Studio

The frontend, built with React and Vite, serves as the central control and visualization hub.

- **Live Data Visualization:** Uses Recharts to display the final emotion vector, and provides panels showing raw data from the game, sensors, and audience.
- **Music DNA Hub:** Allows a user to upload any audio file, which becomes the ”Music DNA” that AURA modulates.
- **AI Music Generator:** Integrates Magenta.js, allowing the user to generate new musical melodies using a MusicVAE model directly in the browser and upload them as DNA.
- **Audience Polling:** Provides a link to a separate audience view (‘audience.html‘) where viewers can vote on the desired mood, adding an interactive layer for live-streamed scenarios.

5 Challenges and Solutions

During development, our team navigated several technical hurdles inherent to building a real-time, multi-modal system. Overcoming these challenges was critical to the project’s success.

- **Challenge: Real-time Generative Audio Without Stuttering**
  Generating and playing audio seamlessly in a single-threaded environment like Pygame can easily lead to lag or audio dropouts, as chord generation could block the main game loop.

  Solution: We implemented a double-buffered audio playback system using two distinct Pygame mixer channels (`channel1` and `channel2`). While one channel is playing the current chord, the next chord is generated and queued on the other channel. The system uses `pygame.time.get_ticks()` for precise, non-blocking scheduling, allowing us to trigger the playback of the next chord at the exact moment the previous one ends, resulting in a continuous and seamless musical score.

- **Challenge: High Latency and CPU Load from Sensor Streams**
  Processing high-resolution webcam video and continuous audio for emotion analysis is computationally expensive and can introduce significant latency, making the audio modulation feel disconnected from the user's actions.

  Solution: We adopted a multi-faceted optimization strategy. For the face sensor, video frames were downscaled to a manageable width (`STREAM_WIDTH = 320`) before processing. More importantly, the computationally intensive DeepFace analysis was performed only once every N frames, while a simpler face detection ran on every frame to maintain a responsive bounding box. For all sensor data, WebSockets were used for their persistent, low-overhead connection, minimizing network latency compared to repeated HTTP requests.

- **Challenge: Adhering to Browser Autoplay Policies**
  Modern web browsers block audio from playing automatically to improve user experience. This prevented our "Music DNA" track from starting as soon as the page loaded, creating a disjointed experience for the user.

  Solution: We designed the React `AudioPlayer.jsx` component to be user-interaction-driven. The Web Audio API's `AudioContext` is only created and "unlocked" after the user explicitly clicks a "Start Music" button. Once this initial user gesture is registered, the application gains the privilege to make subsequent audio changes—such as pausing, playing, or modifying playback parameters—programmatically, enabling the real-time modulation AURA requires.

6 Setup and Usage Guide

To run the full AURA project locally, follow these steps. A webcam and microphone are required.

6.1 Prerequisites

- Python 3.8+ and 'pip'
- Node.js and 'npm'
- Git

6.2 Installation

```
# 1. Clone the repository
git clone https://github.com/Samsung-Team-AURA/AURA
cd AURA

# 2. Set up a Python virtual environment
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate

# 3. Install all Python packages
pip install -r requirements.txt

# 4. Install all Node.js packages
cd aura_frontend
npm install
```

```
cd ..
```

6.3 Running the Services

The system requires five separate terminal windows.

```
# Terminal 1: AURA Backend
cd aura_backend
uvicorn app.main:app --host 0.0.0.0 --port 8000 --reload

# Terminal 2: AURA Frontend
cd aura_frontend
npm run dev
# Director's Studio available at http://localhost:5173

# Terminal 3: Face Emotion Sensor
python sensor_modules/face_emotion_sender.py

# Terminal 4: Speech Emotion Sensor
python sensor_modules/speech_emotion_sender.py

# Terminal 5: Pygame Client
cd pygame_and_sdks_extensible
python game_runner.py
```

7 Value Proposition and Future Work

7.1 Value for Stakeholders

AURA provides a flexible framework with significant value for both game developers and the broader content creation industry.

- **For Game Studios:** It offers a path to creating deeply immersive and replayable audio experiences that elevate player engagement. The generative engine, in particular, ensures a unique soundtrack for every playthrough.
- **For Content Creators:** The technology serves as a proof-of-concept for a "Video Soundtrack Generator." AURA could be integrated into mobile editing apps on Samsung devices to analyze user videos and automatically generate a custom, royalty-free soundtrack that matches the mood and pacing of their personal memories.

7.2 Future Work

- **Advanced Generative Models:** Replace the Markov Chain model with more sophisticated deep learning models like LSTMs or Transformers for more complex and long-term musical structures.
- **Stem-Based Modulation:** Instead of modulating a single audio track, work with multitrack stems (drums, bass, melody, pads) to allow for more granular control, such as muting the drums during calm moments or adding an intense melodic layer during combat.
- **Engine Integration:** Develop lightweight SDKs for popular game engines like Unity and Unreal to allow developers to easily integrate AURA's orchestration and modulation logic with their native audio systems.

8 Conclusion

The AURA Engine successfully demonstrates a powerful, multi-faceted approach to dynamic audio. By combining traditional adaptive techniques with state-of-the-art generative models and multi-modal sensing, it provides a blueprint for the next generation of interactive and emotionally resonant audio experiences. Its potential extends far beyond gaming, positioning it as a foundational technology for personalized content creation on a global scale.

A Project Dependencies

A.1 Python Dependencies (‘requirements.txt‘)

```
fastapi
uvicorn
python-socketio
websockets
python-dotenv
opencv-python
deepface
librosa==0.9.2
soundfile
scikit-learn
pyaudio
numpy
pydub
tensorflow
tf_keras
python-multipart
pygame
```

A.2 Node.js Dependencies (‘package.json‘)

```
"dependencies": {
    "@magenta/music": "^1.23.1",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-dropzone": "^14.2.3",
    "recharts": "^2.12.7",
    "socket.io-client": "^4.7.5"
}
```