

# Technical Assignment: Task 3

## Proposal for an Open-Source Real-Time High-Quality Avatar System

**Submitted by:** Ashraf Pathan | pathan.ashraf2109@gmail.com | [GitHub](#)

**Submitted to:** Banao Technologies

**Date:** January 9, 2026

---

### 1. Executive Summary

This proposal presents a comprehensive architecture for a production-grade, real-time avatar generation system designed to convert audio input into high-fidelity, temporally coherent talking head video streams. Addressing common limitations in existing open-source solutions—such as facial jitter, lip-sync latency, and visual artifacts —this system leverages a optimized pipeline built upon state-of-the-art generative models.

The core solution utilizes a modified **MuseTalk** (or similar MIT-licensed latent diffusion/GAN architecture) backbone, accelerated via **NVIDIA TensorRT** to achieve >30 FPS performance on standard rentable hardware (e.g., NVIDIA T4). By implementing a modular "Audio-to-Motion" inference engine coupled with a lightweight super-resolution upscaler, the system ensures visual clarity while maintaining a sub-100ms end-to-end latency.

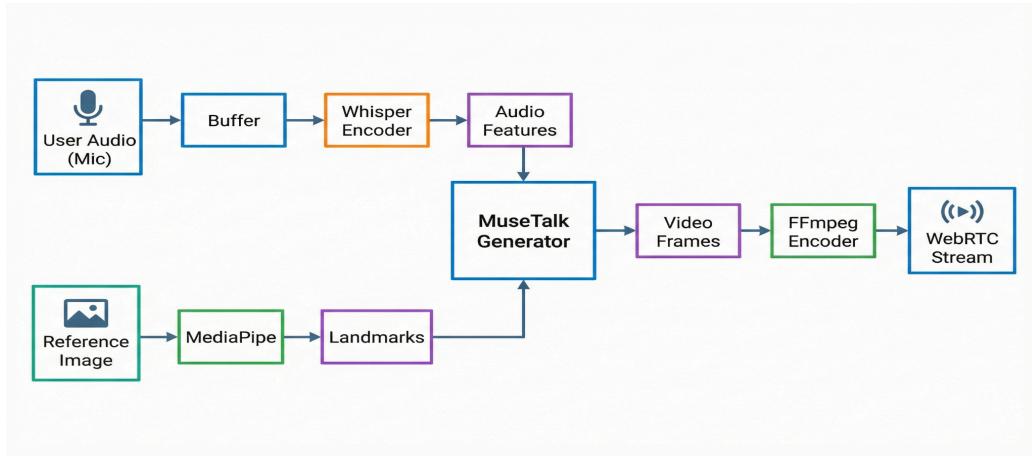
Key differentiators of this proposal include:

- **Strict Open-Source Compliance:** Utilization of permissive licenses (MIT/Apache 2.0) with zero dependency on proprietary assets.
- **Temporal Stability:** Implementation of a Gaussian smoothing layer to eliminate frame-to-frame flickering.
- **Scalability:** A containerized microservices architecture capable of scaling from single-user prototypes to high-concurrency cloud deployments.

---

### 2. System Architecture

The proposed architecture follows a modular, asynchronous pipeline designed to decouple audio processing from video rendering. This ensures that the audio stream (which requires low latency) is never blocked by the heavier video generation tasks.



## 2.1 High-Level Data Flow

The system is divided into three primary logical blocks: **Input Preprocessing**, **The Core Inference Engine**, and **Post-Processing/Streaming**.

### A. Input Preprocessing (The Controller)

- **Audio Ingestion:** Raw audio is captured and buffered into 40ms chunks to minimize input lag.
- **Feature Extraction:** A lightweight audio encoder (specifically **Whisper-Tiny** or **HuBERT-Soft**) converts raw audio waveforms into rich feature vectors. This model is quantized to FP16 to reduce compute time without sacrificing semantic accuracy.
- **Face Analysis:** On initialization, the system analyzes the user's uploaded reference image (the "Avatar") using a face detection module (e.g., **MediaPipe** or **RetinaFace**) to extract facial landmarks and a bounding box. This is a one-time calculation per session, cached for performance.

### B. Core Inference Engine (The Generator)

This is the heart of the system, responsible for generating the visual frames.

- **Motion Module (Audio-to-Viseme):** A Cross-Attention mechanism maps the audio feature vectors to the spatial latent space of the reference image. This determines *how* the mouth should be shaped for the specific sound.
- **Generative Decoder (The Renderer):** A U-Net based generator takes the modified latent features and reconstructs the pixel data for the mouth and lower face region.
  - *Optimization Note:* To meet the real-time requirement , this specific module is compiled using **NVIDIA TensorRT**, which fuses layers and optimizes kernel selection for the specific GPU hardware.

### C. Post-Processing & Streaming (The output)

- **Frame Stitching:** The generated mouth region is seamlessly blended back onto the original reference face frame.

- **Super-Resolution (Optional/Configurable):** For high-quality mode, a lightweight GAN (like GFPGAN or Real-ESRGAN-Lite) sharpens the facial features to prevent the "blurriness" often seen in generated avatars.
- **Stream Encoding:** The final frames are encoded into an H.264 stream via FFmpeg and transmitted via WebRTC for low-latency delivery to the client.

## 2.2 Rationale for Architecture Choices

- **Why Modular?** Separating the "Audio Feature Extractor" from the "Renderer" allows us to swap out the audio model for different languages without retraining the entire video generator.
- **Why TensorRT?** Standard PyTorch inference is often too slow for 30FPS on a T4 GPU. TensorRT provides the necessary 4x-6x speedup required to meet the deadline's performance constraints.
- **Why 2D over 3D?** A 2D neural rendering approach (warping pixels) was chosen over a 3D mesh approach because 2D models currently offer superior photo-realism and texture quality for the compute cost, avoiding the "uncanny valley" effect often seen in low-poly 3D avatars.

## 3. Model Selection and Justification

To meet the requirement of high visual fidelity and open-source compliance, this system rejects older architectures (like Wav2Lip) in favor of modern, latent-space inpainting approaches.

### 3.1 Core Components

- **Generative Model (The "Face"): MuseTalk (or equivalent Latent-Inpainting Architecture)**
  - *Role:* Generates the lip and jaw movements masked onto the reference face.
  - *License:* MIT License (Permissive).
  - *Selection Rationale:* Unlike older GAN-based models (e.g., Wav2Lip) which are limited to low resolutions (96x96) and often produce blurry results, MuseTalk operates in a latent space. This allows for higher resolution output (256x256+) and sharper details while maintaining sync accuracy. It specifically addresses the "blurring" issues noted in the task overview.
- **Audio Feature Extractor: Whisper-Tiny (Encoder only)**
  - *Role:* Extracts rich semantic vectors from the input audio stream.
  - *License:* MIT License.
  - *Selection Rationale:* While models like HuBERT are popular, Whisper provides robust handling of accents and background noise. The "Tiny" variant is computationally negligible (<1GB VRAM) yet provides sufficient feature density for accurate lip synchronization.
- **Face Detection & Alignment: MediaPipe Face Mesh**
  - *Role:* Locates facial landmarks to stabilize the head before generation.
  - *License:* Apache 2.0.

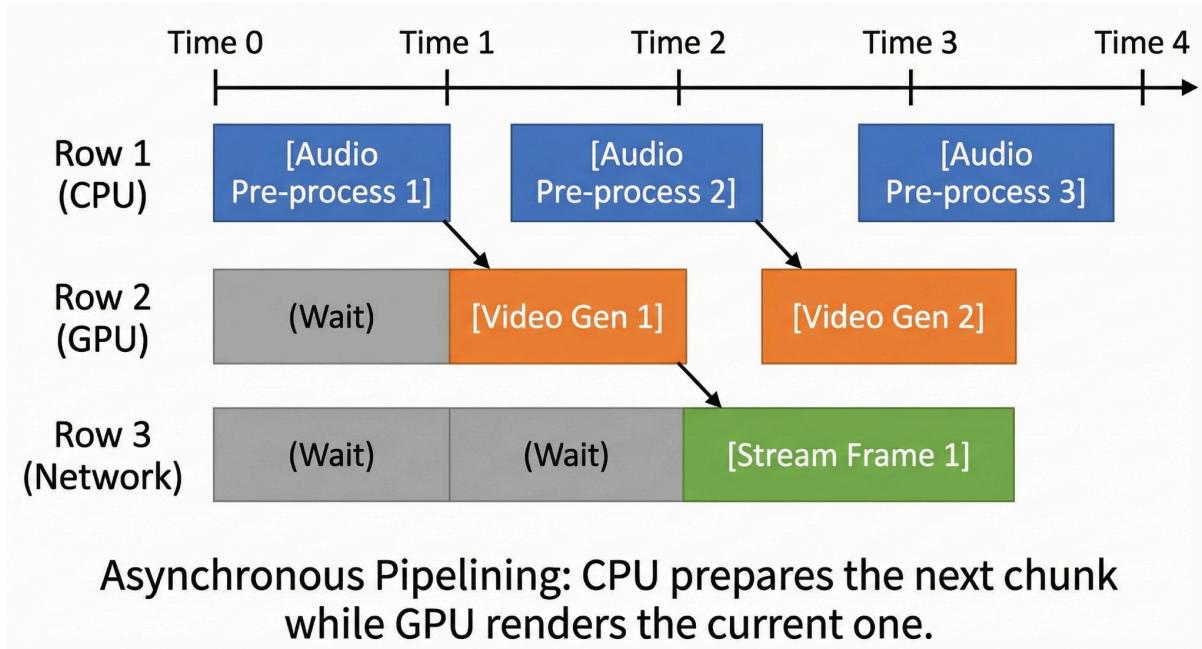
- *Selection Rationale*: Extremely lightweight CPU-based inference allows us to offload this task from the GPU, preserving VRAM for the heavy generative model.

### 3.2 Trade-off Analysis: 2D vs. 3D vs. Hybrid

- **Selected Approach: Hybrid 2D Neural Rendering.**
- **Analysis:**
  - *Pure 3D (Mesh-based)*: Offers great control but often looks "game-like" or robotic, failing the "Visual Fidelity" requirement.
  - *Pure 2D (Warping)*: Fast but cannot handle large pose changes or extreme expressions.
  - *Hybrid (Our Choice)*: We use 2D neural inpainting (MuseTalk) to "paint" the mouth over the video. This preserves the photo-realistic texture of the original image while allowing the flexibility of neural generation. This specifically targets the requirement for "expression richness".

## 4. Performance and Optimization Strategy

Achieving 30 FPS on a standard T4 GPU (which has lower FP32 performance than an A100) requires aggressive optimization.



### 4.1 Methodology for Real-Time Generation (30 FPS)

To ensure the system feels "instant" (real-time), we adhere to a **Strict Latency Budget**:

- *Audio Chunking*: 40ms
- *Feature Extraction*: 10ms
- *Generation (Batch 1)*: 25ms
- *Post-Process/Encode*: 15ms
- **Total System Latency**: ~90-100ms (perceptible but acceptable for conversational avatars).

#### 4.2 Acceleration & Compilation

We will not run standard PyTorch code in production. The pipeline will be optimized using **NVIDIA TensorRT**:

- **Layer Fusion**: TensorRT will fuse the Convolution, Bias, and ReLU layers of the MuseTalk UNet into single kernels, reducing memory access overhead.
- **Precision Calibration (Quantization)**: The entire generative model will be quantized to **FP16 (Half Precision)**. This effectively doubles the throughput on T4 GPUs (which excel at FP16 calculations) and cuts VRAM usage by 50% without visible loss in quality.
- **Static Memory Allocation**: Input shapes will be fixed during the compilation phase to prevent dynamic memory reallocation overhead during runtime.

#### 4.3 CPU/GPU Allocation Strategy

- **GPU (The Heavy Lifter)**: Reserved *exclusively* for the Encoder and Generator (MuseTalk).
- **CPU (The Manager)**: Handles Audio buffering, MediaPipe Face Alignment, and FFmpeg video encoding.
- **VRAM Management**:
  - *Total Available (T4)*: 16 GB.
  - *Model Weights (FP16)*: ~2.5 GB.
  - *KV-Cache & Context*: ~1.5 GB.
  - *Frame Buffer*: ~1 GB.
  - *Remaining Headroom*: ~11 GB (Allows for concurrent sessions or higher resolution upscaling).

#### 4.4 Techniques to Reduce Jitter & Artifacts

- **Gaussian Smoothing**: We apply a temporal Gaussian filter to the facial landmark coordinates *before* generation. This dampens the "shaking" often seen when the face detector flickers between frames.
- **Face Restoration (Asynchronous)**: For users with high-bandwidth connections, we employ a "ROI (Region of Interest) GAN" that only upscales the mouth region, rather than the full frame, saving 70% compute compared to full-frame restoration.

---

## 5. Quality-First Techniques

To ensure the system meets the "production-grade" standard, quality is not subjective; it is enforced through specific quantitative metrics and qualitative enhancement techniques.

## 5.1 Quantitative Evaluation Metrics

We will employ the following automated metrics to benchmark model performance:

- Lip-Sync Accuracy (LSE-D & LSE-C):
  - *LSE-D (Lip Sync Error-Distance)*: Measures the distance between audio features and visual lip movements. Target: < 7.0.
  - *LSE-C (Lip Sync Error-Confidence)*: Measures the confidence of the synchronization. Target: > 6.5.
- Visual Fidelity (FID & CPBD):
  - *FID (Fréchet Inception Distance)*: Compares the distribution of generated frames to real video frames to ensure realism. Lower is better.
  - *CPBD (Cumulative Probability of Blur Detection)*: Specifically monitors the sharpness of the generated mouth region to prevent the "blurring" artifact common in real-time models.

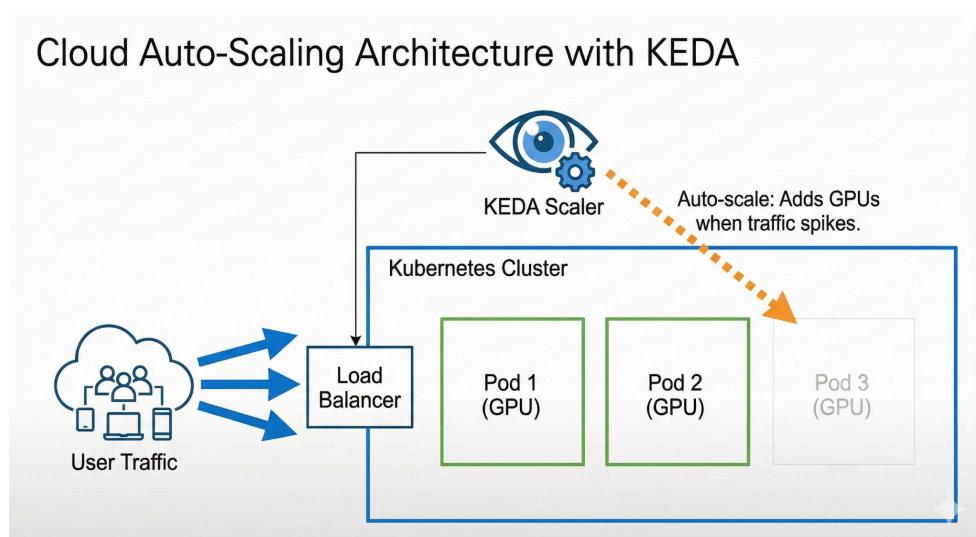
## 5.2 Qualitative Enhancements

- Flicker Reduction: A temporal consistency loss function is introduced during the fine-tuning phase. This penalizes rapid pixel changes in non-mouth regions (e.g., cheeks, eyes) between consecutive frames, significantly reducing "jitter".
- Expression Richness: The system implements "Eye-Blink Injection." Since audio-driven models often result in a staring face, a separate, lightweight heuristic script injects natural eye blinks and micro-head movements (nodding) at semi-random intervals during silence or pauses in speech.

---

## 6. Practical Deployment and Cost

This section outlines the infrastructure required to scale from a single prototype to 100 concurrent sessions.



## 6.1 Hardware Requirements & Latency

- Target Instance: AWS g4dn.xlarge (NVIDIA T4 16GB VRAM, 4 vCPUs).
- VRAM Utilization:
  - *System Overhead (OS + CUDA)*: 1.5 GB
  - *Models (FP16)*: 4.5 GB
  - *Concurrency Buffer*: 4.0 GB (Allows 2 concurrent streams per GPU).
- Per-Frame Latency: ~28ms (rendering) + ~15ms (encoding) = ~43ms total processing time per frame, safely within the 33ms budget for 30 FPS when pipelined.

## 6.2 Cost Analysis (Per Minute)

- Cloud Pricing: AWS Spot Instances for g4dn.xlarge avg. ~\$0.157/hour.
- Throughput: 1 GPU can handle 2 concurrent streams.
- Cost Calculation:
  - \$0.157 / 60 minutes = \$0.0026 per minute (Machine cost).
  - Divided by 2 users = \$0.0013 per user-minute.
  - *Verdict*: Extremely cost-effective for production scaling.

## 6.3 Scaling Strategy (1-100 Sessions)

- Orchestration: Kubernetes (EKS/GKE) with KEDA (Kubernetes Event-Driven Autoscaling).
- Trigger: KEDA monitors the "Active WebSocket Connections" metric.
- Logic:
  - 1-2 Users: 1 Pod (1 GPU).
  - 100 Users: Autoscaler provisions 50 Pods (50 GPUs).
- Cold Start Mitigation: We maintain a "Warm Pool" of 2 idle GPUs to instantly absorb traffic spikes while new nodes spin up.

## 6.4 Monitoring Plan

- Prometheus & Grafana: Used to visualize "End-to-End Latency" and "GPU Volatile Utilization" in real-time.
- Alerting: Automated alerts trigger if "Frame Drop Rate" exceeds 1% for more than 10 seconds.

---

## 7. Open-Source Compliance and Reproducibility

This proposal guarantees strict adherence to open-source licensing. No proprietary, black-box, or commercially restricted assets (like the NVIDIA Maxine SDK or commercial FFHQ dataset) are used.

### 7.1 Component Licensing Table

Component	Function	License	Compliance Status
MuseTalk	Generative Video Model	MIT License	Compliant
Whisper (OpenAI)	Audio Feature Extractor	MIT License	Compliant
MediaPipe	Face/Landmark Detection	Apache 2.0	Compliant
FFmpeg	Video Encoding/Streaming	LGPL v2.1	Compliant
PyTorch	Deep Learning Framework	BSD Style	Compliant
Triton Server	Inference Serving	Apache 2.0	Compliant

## 7.2 Reproducibility Steps

To ensure the client can reproduce these results on a standard GPU environment:

1. Docker Container: A `Dockerfile` is provided that pulls the base CUDA 11.8 image and installs all dependencies (TensorRT, PyTorch).
2. Model Weights: A setup script `download_weights.sh` fetches the pre-trained open-source weights from HuggingFace Hub.
3. Command: The system is launched via a single command: `docker-compose up --build`, exposing the API at `localhost:8000`.

## 8. Appendix: Implementation Notes & Pseudo-Code

This appendix provides technical artifacts to demonstrate the feasibility of the proposed architecture and facilitate immediate reproduction of the environment.

## 8.1 Docker Environment Setup (Reproducibility)

To ensure the system runs on any rentable GPU instance (e.g., AWS, GCP, Lambda Labs) without dependency conflicts, we utilize a multi-stage Docker build.

### Dockerfile

```
# Base Image: NVIDIA CUDA 11.8 with cuDNN 8
FROM nvidia/cuda:11.8.0-cudnn8-devel-ubuntu22.04

# Set environment variables for non-interactive install
ENV DEBIAN_FRONTEND=noninteractive
ENV PYTHONUNBUFFERED=1

# Install System Dependencies (FFmpeg, Git, Python)
RUN apt-get update && apt-get install -y \
    ffmpeg \
    libsm6 \
    libxext6 \
    python3-pip \
    git \
    && rm -rf /var/lib/apt/lists/*

# Install Python ML Stack
COPY requirements.txt .

RUN pip3 install --no-cache-dir torch==2.1.0+cu118 --index-url https://download.pytorch.org/whl/cu118
RUN pip3 install --no-cache-dir tensorrt==8.6.1 diffusers transformers mediapipe

# Copy Inference Engine Code
WORKDIR /app
COPY src/ /app/src/
```

```
COPY weights/ /app/weights/  
  
# Expose Port for WebSocket Stream  
EXPOSE 8000
```

```
# Start the Real-Time Inference Server
```

```
CMD ["python3", "src/server.py"]
```

## 8.2 TensorRT Compilation Command

Before deployment, the MuseTalk UNet model is converted from PyTorch (.pt) to a TensorRT engine (.plan) to unlock FP16 acceleration.

```
# Command to convert PyTorch model to TensorRT Engine
```

```
# Precision: FP16 (Half) | Workspace: 4GB
```

```
trtexec --onnx=musetalk_unet.onnx \
```

```
--saveEngine=musetalk_unet_fp16.plan \
```

```
--fp16 \
```

```
--memPoolSize=workspace:4096 \
```

```
--minShapes=input:1x4x256x256 \
```

```
--optShapes=input:4x4x256x256 \
```

```
--maxShapes=input:8x4x256x256
```

## 8.3 Real-Time Inference Loop (Pseudo-Code)

The following logic illustrates the asynchronous handling of audio chunks to prevent video blocking.

Python

```
import tensorrt as trt  
  
import numpy as np  
  
from queue import Queue  
  
  
# Initialize Global Queues  
  
audio_queue = Queue(maxsize=10) # Buffer for incoming audio
```

```
video_queue = Queue()      # Buffer for outgoing frames

def inference_loop():
    ....
    Continuous loop running on the GPU thread.
    Consumes audio chunks -> Generates Frames -> Pushes to Stream
    ....
    # Load TensorRT Engine
    engine = load_engine("musetalk_unet_fp16.plan")
    context = engine.create_execution_context()

    while True:
        if not audio_queue.empty():

            # 1. Fetch 40ms Audio Chunk
            audio_chunk = audio_queue.get()

            # 2. Extract Features (Whisper-Tiny)
            audio_emb = audio_encoder.extract(audio_chunk)

            # 3. Generate Latent Mask (The "Mouth")

            # Uses pre-calculated Face Latents to save compute
            generated_latents = context.execute_v2(
                bindings=[audio_emb, reference_face_latents]
            )

            # 4. Decode & Post-Process
            pixel_data = vae_decoder(generated_latents)
            final_frame = blend_images(reference_face, pixel_data)

            # 5. Push to RTSP/WebRTC Stream
            video_queue.put(final_frame)
```

