Mirror, Mirror: Prototype Development Journey

1. Introduction

"Mirror, Mirror" was conceived as an interactive prototype for Dutch Design Week. The central idea was to craft an experience where an AI character, an Unreal Engine 5 MetaHuman with dialogue driven by Gemini, not only converses with a visitor but also subtly, progressively clones their voice (using F5-TTS) and face (via live deepfake technology). The journey culminates in the AI confronting the visitor with their own digital doppelgänger.

This document charts the project's development, with a keen focus on the technical hurdles I faced, the problem-solving strategies I employed, and the thinking behind the pivotal architectural and implementation choices made along the way.

2. System Architecture Overview

At its core, "Mirror, Mirror" is a symphony of several interconnected frontend and backend components:

- **Frontend Components (The User-Facing Side):**
    - **User Interface (`Gemini-Live`):** This is the React-based application where the visitor chats with the Gemini-powered AI.
    - **Visual Representation (The AI's Avatar):** The AI character is brought to life as an Unreal Engine 5 MetaHuman.
    - **Input Capture (Listening and Seeing):** The system captures the user's voice through the browser's Web Audio API and their video feed via a standard webcam.
- **Backend Components (The Brains and Mechanics):**
    - **Conversational AI (The AI's Mind):** Google's Gemini API provides the intelligence for the dialogue.
    - **Voice Cloning Subsystem (Mimicking Speech):**
        - **F5-TTS:** The core engine that turns text into synthesized speech.
        - **Custom FastAPI Server (`tts_server.py`):** My own server built to wrap the F5-TTS Python API, specifically designed for quick TTS responses and the ability to switch voice models on the fly.
    - **Face Cloning Subsystem (Visual Mimicry):**
        - **InsightFace:** A neural network model used for detecting and analyzing faces.
        - **Live Deepfake Technology:** The specific library isn't detailed in these logs, but the system is set up to feed it the necessary data.
        - **OpenCV:** A workhorse library used extensively for managing cameras, processing images, and handling video streams.
    - **Primary Application Server (`web_app.py`):** A Flask-based server that juggles camera feeds, handles image uploads, kicks off face detection, and manages the "Pro Stream" monitoring camera.
- **Key Enabling Technologies (The Glue and Special Tools):**
    - **Web Audio API:** Absolutely crucial for getting microphone input in the browser and working with raw audio data.
    - **Trigger Phrases:** These are specific phrases the AI says, which act as cues to automatically start backend and frontend processes like voice recording, switching voice models, or capturing an image.
    - **NVIDIA Audio2Face (A2F):** This tool is integrated to take the F5-TTS audio output and use it to drive real-time facial animation of the MetaHuman.
    - **OBS Virtual Camera:** Plays a vital role as a bridge, taking the visuals of the Unreal Engine MetaHuman and piping them into our face-swapping pipeline.

3. Frontend Development: Creating User Interaction and Experience

Developing the `Gemini-Live` interface and its logic was a journey of refining usability, boosting performance, and weaving in the complex cloning features. My general approach was to identify a problem, dig into its root cause, implement a focused solution, and then see how it worked, ready to iterate further.

3.1. Tackling UI Clutter for a Better Chat Experience

- **The Problem:** My starting point was the official Gemini-Live console, a git repo published by Google. The first look at the chat interface was a bit jarring. It was filled with technical debug info like timestamps and system labels, which made it tough to actually follow the conversation.
- **My Solution Approach:** I started by pinpointing all the visual elements that weren't essential for the user. The fix involved systematically stripping these from the main message display. Then, I streamlined related UI parts, like changing "Console" to the friendlier "Chat" and swapping out complicated filter options for a simple connection status. A bit of restyling for better text layout and spacing finished it off.
- **The Rationale:** The aim was straightforward: make it cleaner and more intuitive so users could focus on the dialogue. A standard, uncluttered chat interface was key to a good interactive experience.

3.2. Nailing AI Turn Completion for Smooth Handoffs

- **The Problem:** For the backend Text-to-Speech (TTS) system to work correctly, I needed to know *exactly* when the Gemini AI had finished its part of the conversation. No guesswork allowed.
- **My Solution Approach:** To get this right, I built a small UI component just for development. This little tool would pop up a "DONE" message and also log the complete AI response to the browser's console. This let me directly watch and debug how the system was detecting the end of an AI turn.
- **The Rationale:** This temporary debugging aid was invaluable. It allowed me to confirm that the turn completion logic was solid, which was a non-negotiable first step before the TTS pipeline could function reliably.

3.3. Wrangling Streaming API Data into Coherent Text

- **The Problem:** The Gemini Live API sends back responses in chunks because it's a streaming API. This initially led to sentences appearing broken or cut off in the chat and in the console logs, which was a problem for readability and for getting complete text for the TTS.
- **My Iterative Solution Process:**
    1. **First Try (Message Grouping):** My initial thought was to try and bundle related message pieces together. This tidied up the UI a bit, but I still had issues getting a perfectly assembled message for other system parts.
    2. **Second Try (Combining at Display Time):** Next, I tried to stitch the text chunks together right before they were shown on screen. This looked better visually, but didn't quite solve the core issue of needing a clean, complete message for things like TTS.
    3. **The Breakthrough (Centralized Message Collector):** The approach that finally worked involved setting up a data structure (a simple `currentModelTurnParts` array) in the main message handling logic. As new text chunks arrived, they were added to this collection. Only when the AI signaled it was done speaking did we combine all these parts into a single, complete message for display and any other processing.
- **The Rationale:** This final method tackled the fragmentation problem right at the source: when the data was received. It ensured that a complete and coherent message was always available *before* it was used by any other part of the system, which proved much more robust than trying to fix it later at the display stage.

3.4. Microphone Management: Banishing Delays and Automating Control

- **The Two-Pronged Problem:** First, the standard way of muting/unmuting the mic (`setMicEffectivelyMuted`) was causing an annoying 2-3 second lag when unmuting. This lag, due to audio capture re-initialization, often meant the beginning of a user's speech got cut off. Second, we needed to make sure the user's mic was silent when the AI was speaking via TTS to avoid feedback.
- **My Solution Approach:**
    - **Killing the Delay:** I changed tactics. Instead of stopping and starting, the microphone stream (via the Web Audio API) now stays physically active all the time. I introduced a "logical" mute state (`isMicMuted`). When the system decided the mic should be "muted" (like when the AI was talking), the recording process simply ignored any incoming audio data. For really precise control, especially in the `AudioRecorder` component, I dived into the Web Audio API's `GainNode`. I created a `setMute` function that could instantly tweak `GainNode.gain.setValueAtTime()` to either zero (mute) or one (unmute).

    - **Automating Mute for AI Speech:** To handle muting during TTS playback, I developed a little helper function called `simulateMicClick`. This function programmatically "clicked" the existing microphone toggle button in our UI. I'd call this to mute the mic just before sending text off for TTS, and then call it again to unmute once the `TTSDebugPlayer` signaled that playback was over.
- **The Rationale:** The "logical muting" with direct `GainNode` control got rid of the re-initialization delay, making mic transitions instant and the conversation flow much smoother. Using the existing UI toggle for automation was a clean way to ensure consistency and avoid writing new muting code just for when the AI spoke.

3.5. Speeding Things Up: Client-Side Camera Verification

- **The Problem:** Even after some backend optimizations for finding cameras, using those backend-streamed MJPEG previews to check if a camera was working on the frontend was still painfully slow (3-5 seconds per camera) and often just plain unreliable (CORS errors, broken streams).
- **My Solution Approach:** I decided to shift the task of camera enumeration and preview for *verification* purposes completely to the client-side. This meant using browser native APIs like `navigator.mediaDevices.enumerateDevices()` and `navigator.mediaDevices.getUserMedia()`. This gave me direct, super-fast (sub-100ms) access to camera streams, which I then rendered in a local `<video>` element. Matching device labels to pick the right camera was then done in JavaScript.
- **The Rationale:** For this specific task of quick verification, using direct browser APIs was a dramatic improvement in both speed and reliability. It cut out all the network lag, MJPEG complexities, and CORS headaches that came with routing it through the backend.

3.6. Creating a Seamless Start: Automated System Initialization

- **The Problem:** The way the application started up initially was not providing a great UX. A user might have to manually select cameras or start streams, and there wasn't clear feedback if things were still loading. It wasn't the polished, professional experience I was after, especially for an art installation.
- **My Solution Approach:** I designed and built a proper loading screen component in React. This screen doesn't just look good; it automates and visualizes the entire startup sequence:
    1. It first makes sure it can connect to the backend server.
    2. Then, it waits for the camera list (using the improved naming and OBS detection I'd developed for the backend).
    3. If an OBS Virtual Camera is found, it's automatically selected.
    4. It then automatically fetches a default source image (`image.jpeg`), sends it to the backend, and kicks off the face detection process.
    5. Finally, it auto-starts the main video stream.
    6. Only when all these steps are successfully completed does it transition to the main user interface.
       The loading screen itself features project branding, clear progress indicators, and an expandable log panel for any troubleshooting.
- **The Rationale:** The goal here was to create an "art installation ready" experience: something that just works when you turn it on. This automated sequence eliminates any need for manual configuration by the visitor, provides clear visual feedback on what's happening, and ensures the entire system (including having a source face ready for swapping) is good to go before the main interface appears. It uses the existing `DeepLiveCamContext` for managing state, which keeps everything consistent.

4. Backend Development: Building Robust Services and Core Logic

On the backend, the focus was on creating services that were not only reliable but also performed well enough for the real-time demands of the installation. My problem-solving here revolved around optimizing the core AI functionalities and ensuring all the different pieces could talk to each other without a hitch.

4.1. The F5-TTS Server: Prioritizing Low Latency with Smart API Design

- **The Problem:** For the conversation to feel natural, the AI's speech (TTS) needed to be generated very quickly. My initial thought was that just using the F5-TTS command-line tool for every sentence the AI needed to say would be far too slow, mainly because it would have to load the big TTS models every single time.
- **My Investigation and Decision:** I dug into how F5-TTS worked. Sure enough, its CLI (`f5-tts_infer-cli`) loads the models from scratch on each run: a huge overhead. However, its Gradio web application (and the underlying Python API, `src/f5_tts/api.py`) was smarter: it loaded the models once at startup. So, the clear path forward was to use that Python API within a server process that would stay running.
- **Implementation:** I built `tts_server.py` using FastAPI. When this server starts, it loads up the F5-TTS model, its vocoder, and a default voice. Its main job is to listen for requests on a `/generate_tts` POST endpoint.
- **The Rationale:** This server setup means the costly model loading happens only once. Every subsequent request for speech is then much faster. I picked FastAPI because it's known for being quick and making it relatively easy to build these kinds of asynchronous services.

4.2. The Voice Switcheroo: Implementing Dynamic Voice Cloning

- **The Objective:** A key part of the experience was for the AI to start with a generic voice and then, at a specific moment, switch to using the visitor's own cloned voice.
- **My Implementation Strategy:**
    - **Managing Voice Profiles:** The server keeps tabs on two voice profiles: a "default" one (`reference_default.wav/.txt`) and a "user" one (`reference_user.wav/.txt`).
    - **State and Control:** I used server-side global variables to track which voice was currently active. Then, I added a new POST endpoint (`/set_active_reference`). The frontend can call this endpoint with a simple message like `{"reference_name": "user"}` or `{"reference_name": "default"}` to tell the server which voice to use.
    - **Frontend Coordination:** When the chat interface loads, the frontend tells the server to use the "default" voice. Later, when the AI says the trigger phrase **"Test subject identification complete."**, the frontend tells the server to switch to the "user" voice. This call is timed to happen *just before* the TTS request for that specific AI utterance, ensuring the voice change is immediate.
- **The Rationale:** This feature really helps sell the narrative of the AI adapting to the user. Using an API to control the switch keeps the frontend's job (managing the conversation) separate from the backend's job (making the sounds), which is a cleaner way to design it.

4.3. Bringing the MetaHuman to Life: NVIDIA Audio2Face Integration

- **The Objective:** I wanted the speech synthesized by F5-TTS to drive the facial animation of the MetaHuman character in real-time.
- **My Implementation Strategy (`tts_server.py`):**
    - For communication with NVIDIA Audio2Face, I used gRPC. This involved generating some Python client code from the official `audio2face.proto` file (which is like a blueprint for how they talk to each other).
    - I created an `async` function, `send_audio_to_a2f`, to handle the gRPC connection and stream the processed F5-TTS audio (converted to Float32 format and broken into small chunks) over to Audio2Face. This stream also included necessary control messages.
    - The really important part here was using FastAPI's `BackgroundTasks` feature. After the main TTS audio is generated for the `Gemini-Live` frontend, the job of sending that same audio to Audio2Face is scheduled to run as a background task.
- **The Rationale for the Background Task:** This was a crucial design choice. Streaming audio to Audio2Face can sometimes take a moment, and I didn't want that to delay sending the primary TTS audio back to the user in the chat. Running it in the background keeps the main chat interaction snappy.

4.4. Backend Performance Boosts: Camera Enumeration & Model Pre-loading

- **The Problems:** Early on, just figuring out which cameras were connected was taking ages (30-90 seconds!) because of how Python/OpenCV handled it. Separately, the very first time a user uploaded an image for the face swap, there was a long pause (10+ seconds) because the large InsightFace model was loading on demand.
- **My Solution Approach (Backend):**
    - **Speeding Up Camera Discovery:** In `web_app.py`, I first tried reducing the number of camera indexes it checked. Then, I added a 60-second cache for the list of found cameras. I also made it start looking for cameras as a background task when the server first boots up and added a quick check to make sure each found camera could actually provide a frame.
    - **Pre-loading the Face Model:** I changed the Flask application (`web_app.py`) to load that big InsightFace model right when it starts, instead of waiting for the first upload.
- **The Rationale:** These backend fixes dramatically cut down on initial startup times and made those first user interactions feel much more responsive. Pre-loading heavy models and optimizing how we search for devices are pretty standard ways to improve performance.

4.5. Smarter Cameras: Better Naming & Critical OBS Integration

- **The Problem:** The camera list initially just showed generic names like "Camera 0", which wasn't very helpful. More importantly, the whole deepfake pipeline depended on reliably finding and using "OBS Virtual Camera" (which captures the Unreal Engine output).
- **My Solution Approach (Backend - `web_app.py`):** I built a more sophisticated camera detection system. It tries several methods to get useful names (looking at OpenCV properties, querying WMI on Windows, using `pygrabber` for DirectShow names, and having some sensible fallbacks). I also wrote a specific function, `find_obs_camera_index()`, just to locate OBS cameras. The API response from `web_app.py` now clearly flags any OBS cameras, and the frontend (as detailed in 3.5) uses this information to auto-select it.
- **The Rationale:** Getting OBS Virtual Camera selected automatically is non-negotiable for the core rendering-to-deepfake workflow. Better camera names are a nice UX win for everyone. The multi-layered detection approach means we get the best possible names, whatever the system setup.

4.6. Adding a Watchful Eye: The Independent "Pro Stream" Camera

- **The Problem:** I realized I needed a way to have a secondary, lightweight camera stream for general monitoring (like keeping an eye on the user or the overall setup) without interfering with the main camera feed being used for face-swapping.
- **My Solution Approach (Backend - `web_app.py`):** I added new API endpoints specifically for this "Pro Stream" camera: one to detect if such a camera is present, others to start and stop its stream, and one for the video feed itself. The camera enumeration logic was updated to also look for cameras named "PRO STREAM." This stream uses its own separate OpenCV capture instance and is deliberately set to use lower-demand settings (like 320x240 resolution at 15 FPS), all managed by `web_app.py`.
- **The Rationale:** This gave us the monitoring capability we needed without bogging down the main, resource-intensive face-swapping pipeline. Keeping it separate and lightweight was key.

4.7. Quick Checks: A Utility for Rapid TTS Testing (`play_tts.py`)

- **The Objective:** I needed a simple way to quickly test the output of the `tts_server.py` without having to fire up the entire frontend application.
- **My Solution:** I created a small command-line script called `play_tts.py`. It just sends some text to the `/generate_tts` endpoint and then uses the `sounddevice` and `soundfile` libraries to play the audio it gets back.
- **The Rationale:** This little tool helped speed up development of the TTS component. It's good practice to have ways to test parts of your system in isolation.

5. Integrated Automated Workflows: The Mechanics of Voice and Face Cloning

The real magic happens when the frontend and backend systems work together to automate the voice and face cloning processes.

5.1. From User's Voice to AI's Voice: The Automated Recording Pipeline

- **The Problem:** The original way of cloning a voice was a pain: manual downloads, file transfers, just a lot of manual steps.
- **My Solution Approach (An Integrated Frontend/Backend Effort):**
    1. **Trigger-Happy Recording (Frontend):** As discussed before (in `Altair.tsx`), voice recording (controlled by the `isRecordingStarted` flag) only starts when the AI says a specific trigger phrase. Raw audio gets captured using the Web Audio API.
    2. **Clean-Up Crew (Frontend):** The captured audio then gets a bit of processing on the client-side: silence is removed, a little padding is added, and it's encoded into a proper WAV format.
    3. **Off to the Server (Frontend/Backend):** When a "stop" trigger phrase is heard, the processed audio `Blob` is sent from `Altair.tsx` straight to the `/upload_reference_user` endpoint on `tts_server.py`. The server saves this as `reference_user.wav`. The frontend then immediately pings another endpoint, `/transcribe_reference`; `tts_server.py` gets to work transcribing that audio and saves the text as `reference_user.txt`. If anything goes wrong with the server communication, the frontend has a fallback to just download the audio locally.
    4. **Activation Time (Frontend/Backend):** Once the transcription is done, `tts_server.py` is all set to use this new voice. This is typically activated by another call from the frontend to `/set_active_reference`.
- **The Rationale:** This automated pipeline was a huge step up for the user experience, making the voice cloning process practically invisible. The trigger phrases ensure we only record what we need, the client-side processing makes the audio better, and the server automation handles all the backend grunt work.

5.2. "Say Your Name": Face Capture Triggered by Conversation

- **The Objective:** I wanted to automate capturing the user's face for the deepfake process, making it feel like a natural part of the conversation rather than a separate "upload your photo" step.
- **My Solution Approach (Another Frontend/Backend Team-Up):**
    - **The Cue:** The AI says, **"State your name."**
    - **The Process:**
        1. `Altair.tsx` on the frontend spots this phrase.
        2. The frontend then calls an API endpoint, `/api/capture_pro_stream_screenshot`, on the `web_app.py` backend server.
        3. `web_app.py` takes a high-resolution photo using the Pro Stream camera (which should be pointed at the user), saves it, and then tells the face-swapping system to use this new image as its source.
- **Smart Pro Stream Camera:** As mentioned, the Pro Stream camera is set up to grab high-quality images for these important screenshots (good for face detection accuracy).
- **The Rationale:** This makes a crucial part of the deepfake pipeline completely seamless. The user's face is captured naturally during the chat, which really boosts the sense of immersion.

6. Recap

The overall interactive narrative was carefully designed as a sequence of automated events, all kicked off by what the AI says. This was to achieve the AI's transformation: adopting the user's voice and face in a way that felt gradual and impactful, without the visitor needing to do anything more than just talk.

- **My Design Rationale for the Sequenced Automation:**
    1. **Kicking off Voice Recording:** The AI says something like **"...describe what you see outside..."** → This tells the frontend to start recording the user's voice.
    2. **Processing and Adopting the Voice:** The AI says, **"Test subject identification complete."** → Recording stops. The audio is processed, sent to `tts_server.py`, and transcribed. Then, the frontend tells `tts_server.py` to switch to this new "user" voice profile. From this point, the AI starts speaking with the user's cloned voice.
    3. **Capturing the Face:** The AI asks, **"State your name."** → The Pro Stream camera snaps the user's photo, which is then processed and set as the source image for the deepfake.
    4. The Full Reveal: The AI says, "...take a moment to look at yourself..." → At this point, the video output (the MetaHuman) shows the AI character now with the user's face deepfaked onto it, and speaking with their cloned voice.

       This carefully orchestrated flow allows the AI's adoption of the user's characteristics to unfold step-by-step. This gradual reveal, rather than an abrupt switch, was designed to maximize the intended psychological and artistic impact of the "digital doppelgänger" effect.