NTNU
Kunnskap for en bedre verden

Avatar Pipeline Analysis

Analysis Reports

Pipeline Analysis Report

Report:
Time, spending and bottleneck analysis of the party avatar pipeline.

13th February 2026

Table of Contents

List of Figures

List of Tables

1 Introduction

This report analyses the party avatar pipeline: the backend and frontend flow that turns a user question (text or speech) into a spoken, lip-synced avatar response. The goal is to identify weaknesses in terms of *time* (latency and throughput), *spending* (API and infrastructure cost), and *bottlenecks* (sequential steps, rate limits, and resource contention).

The pipeline is used in a hackathon avatar application where a political party representative answers questions. Users can send text (`/tts`) or voice (`/sts`); the backend calls OpenAI for reply generation, ElevenLabs for text-to-speech, and Rhubarb Lip-Sync for phoneme data, then returns messages with base64 audio and lip-sync cues to the frontend.

The analysis is based on the codebase in `src/avatar/backend` and `src/avatar/frontend`, including the Express server, OpenAI (LangChain) and Whisper modules, ElevenLabs TTS, and the lip-sync orchestration that uses ffmpeg and Rhubarb.

2 Pipeline Overview

The pipeline has two entry points: text (`POST /tts`) and speech (`POST /sts`). Both converge to the same processing chain after obtaining a user message.

2.1 Text Path (`/tts`)

The server receives `message` in the body. It first checks for default responses (empty message or missing API keys) by reading pre-generated audio and JSON from disk. If no default applies, it loads a structured-output parser and the OpenAI chain (including party program from `content/party_program.md`), invokes the chain to get up to three messages (text, facial expression, animation), then runs the lip-sync stage and returns the result.

2.2 Speech Path (`/sts`)

The server receives base64-encoded audio. It decodes to a buffer, converts WebM to MP3 via ffmpeg in `utils/audios.mjs`, writes the MP3 to a temp file, and calls the OpenAI Whisper API for transcription. The resulting text is then processed exactly like the text path (default check, OpenAI chain, lip-sync).

2.3 Lip-Sync Stage

For each message returned by the LLM:

1. **TTS**: ElevenLabs `textToSpeech` is called (with retries on HTTP 429). Audio is written to `audios/message_`i`.mp3`.
2. **Phonemes**: The MP3 is converted to WAV with ffmpeg, then Rhubarb Lip-Sync is run to produce `message_`i`.json` with mouth cues.
3. **Envelope**: The MP3 is read and base64-encoded; the JSON transcript is read and attached to the message.

TTS is done in parallel across messages (`Promise.all`); phoneme generation and file reads are also done in parallel across messages, but only *after* all TTS calls complete.

Figure 1 in the appendix summarises the flow.

3 Time Analysis

3.1 End-to-End Latency

The user experiences the sum of all sequential steps until the first byte of the response. There is no streaming: the client waits for the full list of messages (each with audio and lip-sync data) before playback.

- **Default-message branch**: Fast (disk reads only) when no API keys or empty input.
- **OpenAI chain**: Loading the party program and building the chain add one file read and chain construction per request. The dominant cost is the single LLM call (e.g. GPT-4o-mini), typically on the order of 1–5 seconds depending on input and model.
- **Lip-sync stage**: For n messages (up to 3), TTS runs in parallel but the stage is blocked until *all* TTS finish. Then ffmpeg (MP3→WAV) and Rhubarb run per message, again in parallel. So total time is roughly: $\max(\text{TTS}_1, \ldots, \text{TTS}_n) + \max(\text{ffmpeg+Rhubarb}_1, \ldots, \text{ffmpeg+Rhubarb}_n)$. Each ElevenLabs call can be 1–4 seconds; ffmpeg and Rhubarb add roughly 0.5–2 seconds per message depending on length. With three messages, the lip-sync phase alone can reach several seconds.

3.2 Weaknesses in Time

1. **Strictly sequential stages**: Default check → OpenAI → lip-sync. No overlap between LLM and TTS.
2. **No streaming**: The frontend cannot start playing the first message while the rest are still being generated or synthesised.
3. **Redundant work per request**: Party program and chain are re-built or re-loaded every time; only the chain reference is cached, not the result of `loadPartyProgram()` inside `getOpenAIChain()`.
4. **STS path extra cost**: Speech path adds WebM→MP3 conversion (file write, ffmpeg, read) and a Whisper API call before the same pipeline, increasing latency by 1–3 seconds.
5. **Temp files**: Whisper and audios use temp files; cleanup is done in the happy path, but not guaranteed on early errors.

4 Spending Analysis

4.1 Cost Drivers

- **OpenAI**: GPT-4o-mini (or configured model) is charged per token (input and output). Each user question and the system/party context consume input tokens; the JSON array of messages is output. Whisper is charged per minute of audio for `/sts` OpenAI (2025).
- **ElevenLabs**: Usage is typically metered by characters or by time of generated audio, depending on plan ElevenLabs (2025). Every non-default reply triggers one TTS call per message (up to three per request).
- **Local compute**: ffmpeg and Rhubarb run locally; cost is negligible compared to APIs but adds to hosting if scaled.

4.2 Weaknesses in Spending

1. **No caching**: Identical or repeated questions (or default-responses logic) still go through the full pipeline when they are not caught by the default-message check. No caching of LLM responses or TTS output for the same text.
2. **No tiering**: Every request uses the same model and TTS settings; there is no option to use a cheaper/faster model for simple queries or fallback when rate-limited.
3. **Redundant TTS**: If the same phrase appears in different conversations (e.g. error or fallback messages), it is re-synthesised every time. The default "API keys missing" and "I didn't catch that" paths use pre-generated files only when those exact flows are hit; any other path always calls ElevenLabs.
4. **Whisper on every STS**: Every speech input is sent to Whisper; there is no short-audio or low-confidence path to avoid transcription cost.
5. **Multiple messages**: Cap of 3 messages per turn multiplies TTS and phoneme cost by up to 3 compared to a single-sentence reply.

5 Bottlenecks

5.1 Structural Bottlenecks

1. **Single pipeline per request**: One request holds the full chain (OpenAI → lip-sync) until completion. Concurrency is limited by how many such pipelines the server runs (and by external rate limits).
2. **ElevenLabs as bottleneck**: The code retries on HTTP 429 (rate limit) with `MAX_RETRIES = 10` and `RETRY_DELAY = 0`. Zero delay gives little backoff; under load, many requests can hammer the API and worsen rate limiting. ElevenLabs becomes the main external bottleneck for TTS.
3. **Lip-sync stage ordering**: All TTS must finish before any phoneme step runs. So the slowest TTS message delays the whole phoneme phase. There is no pipeline where message 1 can already be in Rhubarb while message 2 is still in TTS.

5.2 Resource Bottlenecks

1. **Disk I/O**: Reads of party program, default audio/JSON, and writes of `audios/message_*.mp3`, `*.wav`, `*.json`. Under concurrency, multiple requests can contend on the same `audios` directory (and overwrite `message_0.mp3` etc. if not isolated per request).
2. **Process spawning**: Each request runs multiple ffmpeg and Rhubarb processes. No process pool or queue; under load this can stress the OS and CPU.
3. **Memory**: Base64-encoded audio and full JSON are held in memory per message and returned in one response; large replies increase memory and response size.

5.3 Correctness and Robustness

- **Shared filenames**: `audios/message_0.mp3`, `message_1.mp3`, etc., are shared across requests. Concurrent requests can overwrite each other's files and return wrong audio or phonemes. This is a critical correctness bottleneck.
- **Error handling**: If TTS or Rhubarb fails for one message, the error is logged and an empty `mouthCues` is used; the rest of the pipeline continues. Whisper or OpenAI failures fall back to default or "I didn't catch that" without retry or cost control.

6 Conclusion

The avatar pipeline is clear and modular but has several weaknesses in time, spending, and bottlenecks.

Time: Latency is dominated by the sequential chain (OpenAI then lip-sync) and by the lip-sync stage (TTS then phonemes) with no streaming. Caching the party program and reusing the chain more aggressively would reduce per-request work; streaming the first message while generating the rest would improve perceived latency.

Spending: There is no caching of LLM or TTS outputs, and no tiering by query complexity. Every `/sts` call uses Whisper; every non-default reply uses ElevenLabs for up to three messages. Caching and single-message or cheaper-model options would reduce cost.

Bottlenecks: ElevenLabs rate limits (with zero backoff) and the strict TTS-then-phonemes ordering make TTS the main bottleneck. Shared `audios` filenames across requests risk cross-request overwrites and incorrect responses; per-request or per-session directories would fix this. Process spawning for ffmpeg and Rhubarb could be capped or queued under high load.

Addressing the shared-audio filenames and adding minimal caching and backoff would improve correctness and efficiency with limited changes; streaming and cost controls would require broader pipeline changes but would yield the largest gains in user experience and cost.

Bibliography

ElevenLabs (2025). *Pricing - ElevenLabs*. URL: https://elevenlabs.io/pricing (visited on 13th Feb. 2025).

OpenAI (2025). *Pricing*. URL: https://openai.com/api/pricing/ (visited on 13th Feb. 2025).

Wolf, Daniel S. (2025). *Rhubarb Lip Sync*. URL: https://github.com/DanielSWolf/rhubarb-lip-sync (visited on 13th Feb. 2025).

Appendix

A Pipeline Flow Diagram

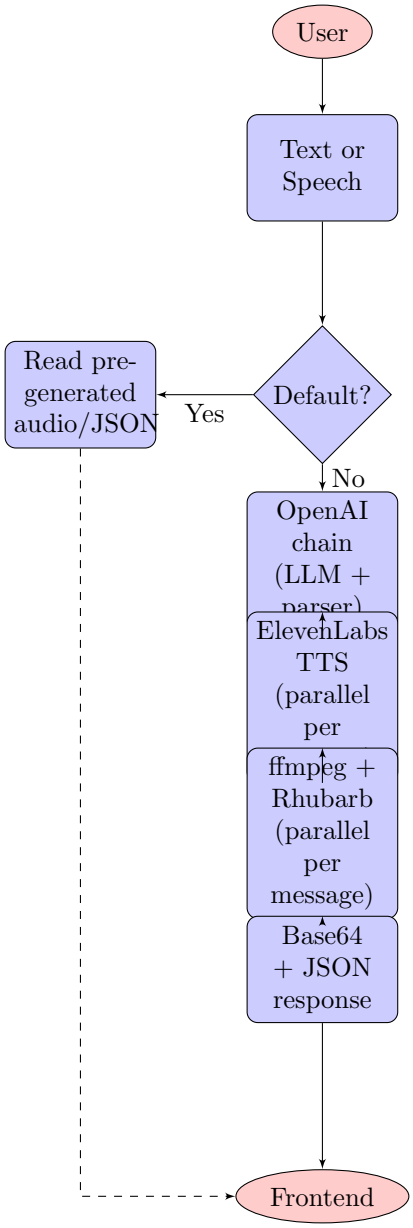

Figure 1: End-to-end pipeline: user input, default check, OpenAI chain, TTS, phonemes, response.

B Key Files

- `server.js`: Routes `/tts`, `/sts`, `/voices`; orchestrates default check, OpenAI, lip-sync.
- `modules/openAI.mjs`: LangChain chain, party program loading, structured output (text, facialExpression, animation).
- `modules/lip-sync.mjs`: Parallel TTS then parallel phonemes + base64/JSON attachment.
- `modules/elevenLabs.mjs`: ElevenLabs TTS.
- `modules/rhubarbLipSync.mjs`: ffmpeg MP3→WAV, Rhubarb phonemes Wolf (2025).

- `modules/whisper.mjs`: WebM→MP3, OpenAI Whisper transcription.