Taran Nathan
23 July 2026

Independent Audio Transcription PoC

1: Executive Product Brief

1.a: Problem Statement

AmplifyMD has built AI features that use transcription of the video encounter with the patient as context for helping physicians with documentation. Currently AmplifyMD has access to these transcripts if its native video platform is used. However, many customers use 3rd party video hardware/software solutions which may not allow AmplifyMD access to this data.

1.b: Value Propositions

Allows customers to use any video solution while still having access to AmplifyMD's AI documentation features.

1.c: Proposed Solution

The Proof of Concept is an embeddable script, that records audio from both the microphone and the system audio source(pipewire, coreaudio, wasapi) via browser APIs. The system audio stream is obtained through the screen sharing API, so even though it will show a popup asking for screen share, it discards the video stream and only uses the audio stream. Once the audio streams are combined into one, it can be sent to a server that provides a speech-to-text translation model, preferably hosted by AmplifyMD. Currently, for demonstration purposes, we are using WhisperAI's API. The simple websocket backend I wrote was made to be discarded, as I don't think that the WhisperAI API would be a good final solution. It would be best to switch to a locally hosted instance of whisper or an alternative for control, cost, and privacy reasons. Currently the transcription and audio are only stored in the browser and is ephemeral(goes away on page reload), and is not sent anywhere yet.

1.d: Scope Limits

The PoC will probably only work on Chromium based browsers(Chrome, Edge, Opera, Vivaldi), as audio capture is only supported on those. Though if the recording is done in a Chromium based browser, as long as the audio is being output through their device's speakers, it can be recorded(ie. Zoom or other video app can be run in the desktop app or in another browser, and it will still be able to record it).

2: User Flow & Functional Specs

2.a: Interaction Flow Diagram

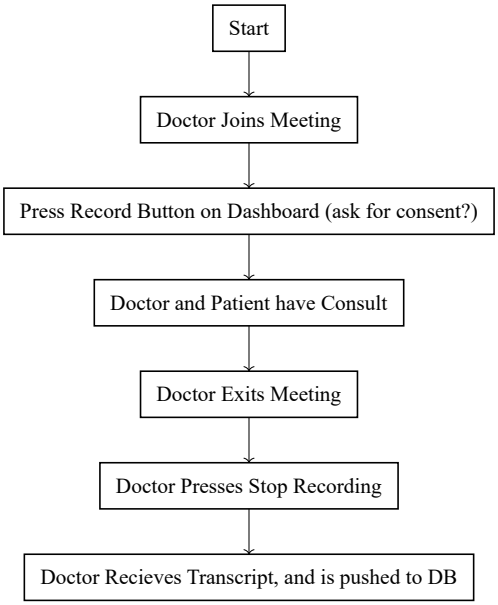

Figure 1: Clinician's Journey

2.b: Functional Requirements

Doctor presses record at start of consult, and the system starts recording and transcribing. Once doctor presses stop, the recording and transcribing stops, and full transcription is uploaded to the database.

3: Technical Design Outline

3.a: Architecture Diagram

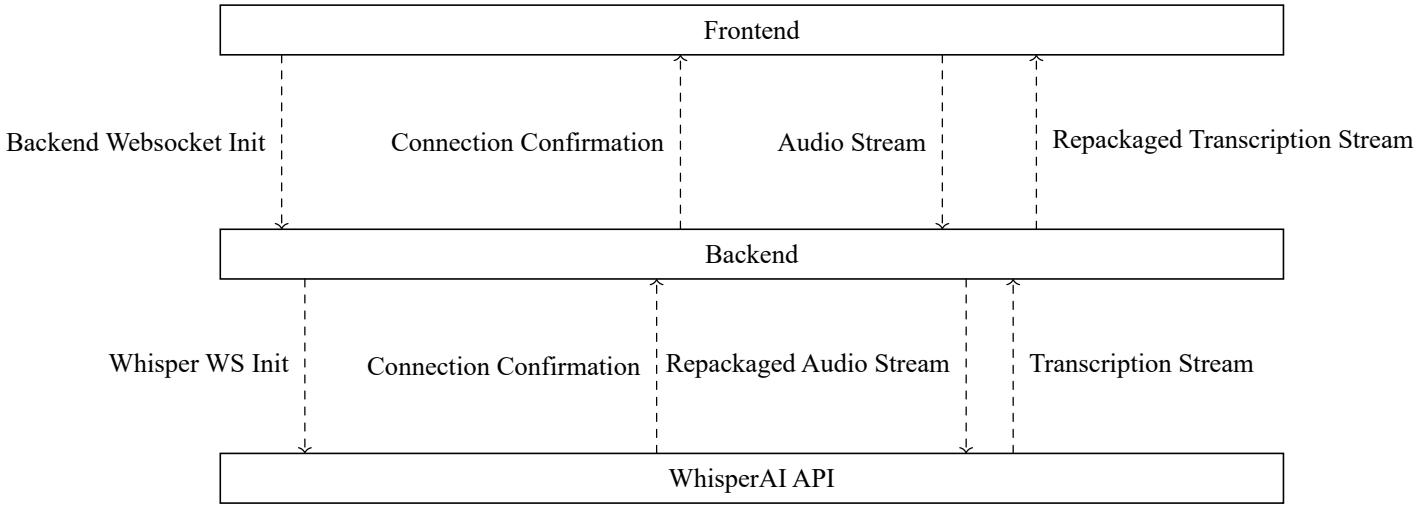

Figure 2: Current Data's Journey

4: Code Snippets

4.a: Frontend Script

Audio Stream Combining:

```JavaScript
microphoneStream = await navigator.mediaDevices.getUserMedia({
  audio: true,
});
systemStream = await navigator.mediaDevices.getDisplayMedia({
  video: true,
  audio: true,
});

if (systemStream.getAudioTracks().length === 0) {
  console.error("No screen share audio track. Microphone only.");
  systemStream.getTracks().forEach((track) => track.stop());
  systemStream = null;
}

audioContext = new AudioContext();
mixedDestination = audioContext.createMediaStreamDestination();

const micSource = audioContext.createMediaStreamSource(microphoneStream);
micSource.connect(mixedDestination);
if (systemStream) {
  const systemSource = audioContext.createMediaStreamSource(systemStream);
  systemSource.connect(mixedDestination);
}

mixedAudioStream = mixedDestination.stream;
```

4.a.a: Push audio on new audio available:

```JavaScript
audioChunks = [];

mediaRecorder.addEventListener("dataavailable", (event) => {
  if (event.data.size > 0) {
    audioChunks.push(event.data);
  }
});
```

4.a.b: Handle WebSocket:

```javascript
const socket = new WebSocket(BACKEND_WS_URL);
socket.binaryType = "arraybuffer";

socket.addEventListener("open", () => {
  transcriptionSocket = socket;
  resolve(socket);
});

socket.addEventListener("message", (event) => {
  if (typeof event.data !== "string") {
    return;
  }

  let parsed;
  try {
    parsed = JSON.parse(event.data);
  } catch {
    return;
  }

  if (typeof parsed.error === "string") {
    console.error("transcription error:", parsed.error);
    return;
  }

  if (parsed.type === "Turn") {
    handleTurnEvent(parsed);
  } else if (parsed.type === "SpeakerRevision") {
    handleSpeakerRevisionEvent(parsed);
  }
});
```

4.a.c: Sending Audio across WebSocket:

```javascript
streamProcessorNode.onaudioprocess = (audioProcessEvent) => {
  if (
    !transcriptionSocket ||
    transcriptionSocket.readyState !== WebSocket.OPEN
  ) {
    return;
  }

  const channelSamples = audioProcessEvent.inputBuffer.getChannelData(0);
  const downsampled = downsampleToTargetSampleRate(
    channelSamples,
    audioContext.sampleRate,
    TARGET_SAMPLE_RATE,
  );
  const buffer = float32ToPcm16Buffer(downsampled);
  transcriptionSocket.send(buffer);
};
```

4.b: Backend WebSocket Pipe

4.b.a: Router:

```rust
let state = AppState {
    whisper_api_key: Arc::new(whisper_api_key),
    whisper_api_base: Arc::new(whisper_api_base),
};

let cors = CorsLayer::new()
    .allow_origin(Any)
    .allow_headers(Any)
    .allow_methods(Any);

let app = Router::new()
    .route("/ws/transcribe", get(ws_transcribe))
    .with_state(state)
    .layer(cors);
```

4.b.b: Websocket Handling (upstream to client is omitted because it is basically the same code, just flipped):

```rust
async fn ws_transcribe(ws: WebSocketUpgrade, State(state): State<AppState>) -> impl IntoResponse {
    ws.on_upgrade(move |socket| proxy_ws_to_whisper(socket, state))
}


async fn proxy_ws_to_whisper(client_socket: WebSocket, state: AppState) {
…                                                                       …
    let client_to_upstream = async {
        while let Some(message_result) = client_rx.next().await {
            let message = match message_result {
                Ok(message) => message,
                Err(error) => {
                    return Err(format!("Client socket read error: {error}"));
                }
            };

            let upstream_message = match message {
                Message::Binary(bytes) => UpstreamMessage::Binary(bytes),
                Message::Text(text) => UpstreamMessage::Text(text.to_string().into()),
                Message::Ping(bytes) => UpstreamMessage::Ping(bytes),
                Message::Pong(bytes) => UpstreamMessage::Pong(bytes),
                Message::Close(frame) => {
                    let close_frame =
                        frame.map(|f| tokio_tungstenite::tungstenite::protocol::CloseFrame {
                            code: f.code.into(),
                            reason: f.reason.to_string().into(),
                        });
                    let _ = upstream_tx.send(UpstreamMessage::Close(close_frame)).await;
                    return Ok(());
                }
            };

```

```
88              if let Err(error) = upstream_tx.send(upstream_message).await {
89                  return Err(format!("Upstream socket write error: {error}"));
90              }
91          }
92
93          let _ = upstream_tx
94              .send(UpstreamMessage::Text(
95                  "{\"type\":\"Terminate\"}".to_string().into(),
96              ))
97              .await;
98          let _ = upstream_tx.send(UpstreamMessage::Close(None)).await;
99          Ok(())
100     };
…                                                                   …
187 }
```