ChatGPT

Local ASR with Speaker Diarization in the Browser

Running offline speech recognition **with speaker diarization** in a Chrome browser is now feasible using efficient open-source models and WebAssembly/WebGPU technology. Below we outline a complete solution – from choosing models to implementation – that works on **Android tablets** and **macOS/Linux laptops** without cloud services. The approach balances accuracy and performance to achieve near real-time transcription with speaker labels while avoiding device overheating or freezing.

1. Recommended ASR Model and Diarization System

OpenAI Whisper is a top choice for local ASR due to its high accuracy across many languages and robustness to noise [1] [2]. Whisper comes in multiple sizes (Tiny, Base, Small, Medium, Large) to trade accuracy for speed. For browser use on typical tablets/laptops, the **Tiny or Base** models (39–74 MB) are ideal – they are significantly lighter than Whisper Large but still achieve good accuracy [3] [4]. In practice, Whisper-Tiny (approximately 39 million parameters) or Whisper-Base (~74 million parameters) can run on CPU in near real-time with acceptable latency, while larger models may overwhelm a tablet's resources.

For **speaker diarization**, a proven solution is to use **PyAnnote** pre-trained models. In particular, the **Pyannote Segmentation** model (e.g. `pyannote/segmentation`) can detect speaker change points and assign speaker identities to segments. An ONNX-converted version of this model (`pyannote-segmentation-3.0`) is only ~6 MB and works well in tandem with Whisper [5] [6]. Pyannote's diarization pipeline uses neural **speaker embedding and clustering** to determine "who spoke when" with labels like *Speaker_1*, *Speaker_2*, etc. This yields consistent speaker labels throughout the transcript (not just per segment). The combination of **Whisper for transcription** and **Pyannote for diarization** has been demonstrated to produce accurate word-level transcripts with speaker tags [7] [8].

Alternatively, **Vosk** (a Kaldi-based ASR) is an option for very low-resource scenarios. Vosk models (50 MB to 1.5 GB) run fully on CPU and are optimized for efficiency [3]. However, Vosk's accuracy lags behind Whisper on challenging audio, and it has *"limited advanced features like speaker diarization"* built-in [9]. If you use Vosk for transcription, you would need a separate diarization step (e.g. using a speaker embedding model and clustering) since Vosk itself does not assign speaker labels. In contrast, Whisper + Pyannote provides an integrated, high-accuracy solution at the cost of higher compute. For most cases, we recommend a **Whisper (Tiny/Base) + Pyannote diarization** pipeline as the best balance of accuracy and local performance.

2. Running Models in the Browser (WebAssembly & WebGPU)

Modern browsers can execute neural networks efficiently thanks to WebAssembly and emerging WebGPU/WebGL support. Both Whisper and Pyannote models can be converted to run in-browser. The preferred approach is to use **ONNX models** and run them with a WebAssembly backend: - **ONNX Runtime Web**: ONNX Runtime has a web edition that executes ONNX models using WebAssembly on the CPU, and can leverage WebAssembly SIMD and multi-threading for speed. It also offers a WebGL backend for GPU

acceleration of some operations, and experimental **WebGPU** support is on the horizon [10] [11]. This means a heavy model like Whisper can run faster by utilizing the device's GPU via WebGL/WebGPU when available, with a fallback to pure WASM if not. In fact, enabling WebGPU in Chrome can significantly accelerate Whisper – one demo recommends using a *"modern browser with WebGPU enabled (fallback to WASM if unavailable)"* for best performance [12]. - **Transformers.js (Hugging Face)**: An easy way to deploy Whisper in-browser is via the Hugging Face **Transformers.js** library. This library loads models (including Whisper and Pyannote) that have been converted to ONNX, and automatically picks the best available backend (WebGPU, WebGL, or pure WASM). For example, the Transformers.js pipeline can run Whisper entirely in the browser using WebAssembly and even take advantage of WebGPU when available [13] [14]. In an in-browser diarization demo, the pipeline ran Whisper's encoder on WebGPU (FP32) and decoder in 4-bit quantized mode, while running the Pyannote model on WASM (since WebGPU wasn't supported for that model) [13] [15]. This hybrid approach maximizes performance by using GPU for the heavy parts of Whisper and keeping smaller models on CPU. - **TensorFlow.js**: Another option is converting models to TensorFlow.js format. TF.js can run models with WebGL, and as of 2023 it introduced WebGPU support in experimental builds. TF.js might be used to run a smaller speech model or a custom model, but Whisper's official conversions are more mature with ONNX/Transformers.js. Still, TF.js could be viable if you convert a smaller ASR model or use a pre-trained TF Lite model for ASR. It provides APIs for both **WebGL (GPUs)** and **WASM** backends.

In summary, **WebAssembly** enables cross-platform CPU inference, and **WebGPU/WebGL** can accelerate math operations on devices with GPUs. The recommended setup is to use ONNX models with a library like Transformers.js or ONNX Runtime Web, which will transparently use available optimizations. This way, the transcription can run *completely locally in the browser* – no server needed – as demonstrated by Hugging Face's Whisper diarization demo *"running 100% locally in your browser using Transformers.js and ONNX Runtime Web"* [16].

3. Libraries and Tools for On-Device ASR in the Browser

To implement the above, you can leverage the following open-source tools:

- **Hugging Face Transformers.js**: High-level library that mirrors the Python Transformers API. It supports pipelines for automatic-speech-recognition and can load models hosted on Hugging Face Hub that have been made web-friendly [17] [18]. Using Transformers.js dramatically simplifies setup – for example, you can load a Whisper model with one line of code: `const transcriber = await pipeline('automatic-speech-recognition', 'Xenova/whisper-tiny.en');` [18] [19]. This will behind-the-scenes download the ONNX model and run inference in browser. Transformers.js also has convenience methods for post-processing (like returning timestamps or generating chunks) and it can load the Pyannote segmentation model similarly. The Hugging Face diarization demo used this library to run both ASR and diarization fully in-browser [5] [20].
- **ONNX Runtime Web**: A lower-level alternative is to use ONNX Runtime Web directly. You would manually load the ONNX model files (which could be served from your web app or a CDN) and then use `onnxruntime-web` APIs to run inference. This gives you more control over the execution provider (WASM vs WebGL) and threading. ONNX Runtime Web supports running models with multiple threads and SIMD for speed (if the site is served with proper cross-origin isolation for threading) and has a WebGL backend for certain operations. It's slightly more involved than Transformers.js but can be useful if you need to optimize or customize the runtime environment.

- **Whisper.cpp (WASM port)**: Whisper.cpp is a C++ implementation of Whisper that is highly optimized (it uses 4-bit quantization and efficient CPU kernels via the GGML library). There are community projects that compile whisper.cpp to WebAssembly for browser use [21] [22]. For example, the `@remotion/whisper-web` NPM package provides helpers to run whisper.cpp in the browser using WASM threads and SIMD. This approach can yield faster CPU inference and a smaller memory footprint (since you can use quantized `.bin` models, e.g. a 32 MB Whisper Tiny model). The downside is you'd need to implement the diarization integration yourself (whisper.cpp doesn't include diarization). However, if you want a pure C++/WASM pipeline, you could run whisper.cpp for transcription and then a smaller WASM module for diarization (e.g. a simple VAD + speaker embedding model).
- **Web Audio / Media APIs**: While not a ML library, the Web Audio API will be essential for capturing audio and possibly doing pre-processing. Libraries like **Recorder.js** or **MediaRecorder API** can help record audio streams if you prefer not to handle raw audio streaming yourself.

In practice, the **Hugging Face solution (Transformers.js + ONNX)** is the most straightforward: it's already configured to use the optimal web backends and provides both the Whisper model and the Pyannote diarization model. Everything is open-source (Whisper's Apache-2.0 license [23], Pyannote's pretrained models, and the Transformers.js MIT license), so you can freely integrate it in your project.

4. Obtaining Pre-Trained Models (Hugging Face, GitHub, etc.)

All required models can be obtained from open repositories:

- **Hugging Face Hub**: Hugging Face hosts converted ONNX models that are ready for browser use. For instance, the ONNX versions of Whisper and Pyannote we discussed are available as:
- `onnx-community/whisper-base_timestamped` – a timestamp-enabled Whisper Base model in ONNX [5].
- `onnx-community/pyannote-segmentation-3.0` – the Pyannote speaker segmentation model in ONNX format [5]. These were used in the reference demo and are optimized for Transformers.js. Hugging Face also provides smaller English-only Whisper models (e.g. `Xenova/whisper-tiny.en`) and quantized variants. For example, `Xenova/whisper-tiny.en` is the Whisper Tiny model converted to ONNX; it can be loaded via Transformers.js easily [24] [18]. The model hub has many such models tagged for **Transformers.js** or **ONNX** [23] [25]. You can search by task (automatic-speech-recognition) and filter for ONNX or Transformers.js compatibility.
- **GitHub**: If you prefer using whisper.cpp, the models (ggml format) can be downloaded from the whisper.cpp GitHub or its linked repository. For example, running the whisper.cpp downloader will fetch models like `ggml-base.en.bin` (English Whisper base) or `ggml-tiny.bin` [26]. These can then be used with the WASM build of whisper.cpp. Additionally, the Pyannote team provides pretrained models in their GitHub/pyannote-audio (though those are PyTorch; for ONNX you'll rely on the HF conversions or convert them yourself using **Optimum** or **pyannote-onnx** toolkit [27]).
- **Other sources**: The Vosk models can be downloaded from the Alphacephei website or via their GitHub. For example, `vosk-model-small-en-us-0.22` (40MB) is an English model. If using Sherpa-ONNX, that project provides pre-trained models for streaming ASR (like next-gen Kaldi models) and even a diarization model (often an X-vector or ECAPA TDNN based model). Those are available on their GitHub or as part of their releases [28] [29], but integrating Sherpa's models in a browser would likely still use ONNX Runtime Web as the backend.

Caching the models in the browser is important for performance and offline use – we discuss that next. But to summarize, the easiest path is to fetch models from Hugging Face Hub on first load (via Transformers.js which does this for you), or self-host the model files and fetch from your server if you need to avoid external calls entirely. All models suggested are open-source and free to use in development (Whisper and Pyannote have acceptable licenses for integration).

5. Offline Model Caching vs. Bundling

Once the model files are downloaded, you'll want to avoid re-downloading them every time. There are two main strategies:

- **Caching after first download**: Browsers can store large assets using IndexedDB, Cache Storage (via Service Workers), or simply in-memory across sessions if the library supports it. The Hugging Face Transformers.js pipeline will automatically cache models in IndexedDB. In the diarization demo, after the ~80 MB Whisper model and 6 MB diarization model are loaded the first time, *"the models will be cached and reused when you revisit the page."* [6] This means even if the user goes offline after the initial load, the models stay available for reuse from local storage. If you implement your own loading, you can use the Cache API or store the response ArrayBuffer of the ONNX model in IndexedDB for subsequent loads.
- **Bundling for offline-first**: You might package the model files with your web app so they're available from the start. For example, you could include the ONNX files in the app and have them load via a local URL or embed them as base64. This ensures completely offline capability on first use. The trade-off is **initial download size** – bundling a ~100 MB model will make your app heavy. You also need to ensure the files are served with proper headers so they can be used by WebAssembly (if using threads, you need correct MIME types and cross-origin isolation). Bundling is useful if users may not have internet at first use, but often a better approach is to let the app fetch the model once and cache it. This way, only users who need the feature pay the download cost, and it can even be done on-demand (lazy-load when user activates transcription).

In either case, after the model is cached or bundled, the system can run **fully offline**. The Hugging Face demo explicitly notes you can *"disconnect from the internet after the model has loaded!"* [30] with no loss of functionality. If using Service Workers, you can also implement an offline fallback page or an installable PWA so that the whole app (including the model) is available offline.

Trade-offs: - Caching (download on first use) gives smaller initial app size and ensures the latest model version is fetched, but introduces a one-time wait when the user first uses the feature. - Pre-bundling ensures immediate availability and no external dependence, but increases app load size for all users (even those who might not use the feature every time) and could complicate deployment (especially if models update – you'd have to update the app).

For development and flexibility, many choose the caching approach (download from a CDN or the Hub, then store). If you need an entirely self-contained offline app (say on an intranet or a device with no internet ever), bundling is the way to go.

6. Integration Guide: Audio Capture to Speaker-Labeled Transcripts

This section outlines how to wire everything together in the browser – from capturing live audio to outputting transcriptions with speaker labels. The pipeline involves: **microphone capture → audio processing → ASR inference → diarization processing → result merging** → display.

A. Capturing microphone audio: Use the Web APIs to get a live audio stream. The primary method is `navigator.mediaDevices.getUserMedia({ audio: true })`, which prompts the user for mic access and returns a MediaStream containing an audio track [31] [32]. For example:

```
const stream = await navigator.mediaDevices.getUserMedia({ audio: true });
// stream now contains a live audio track from the mic
```

This stream can be fed into either the **MediaRecorder API** or the **Web Audio API**: - Using **MediaRecorder**: This API can record the stream in chunks. You can start the recorder and on each `dataavailable` event get a Blob of audio data (e.g., every few seconds). This is simpler for batch processing (you get whole chunks of audio to send to the model), but adds a bit of latency (you wait for each chunk to finish). - Using **Web Audio**: For lower latency streaming, you can pipe the MediaStream into an AudioContext. For instance, create a `MediaStreamAudioSourceNode` from the stream, then use an `AudioWorklet` or ScriptProcessorNode to read raw PCM samples in real-time. This lets you feed the model continuously (frame by frame or in small batches, e.g. 0.5 sec of audio at a time).

For simplicity, many implementations record short segments (e.g. 5 seconds) and run ASR on them sequentially, which achieves near-real-time transcription with manageable load. You may choose a segment length that balances latency and performance (common is 5–10 seconds per chunk with a slight overlap to avoid losing words at boundaries [33]).

B. Feeding audio into the model: Whether you get a continuous PCM stream or a chunk Blob, you need audio in the correct format for the model: - **Resampling**: Whisper and most ASR models expect 16 kHz mono audio. Browser audio is often 48 kHz. If using Transformers.js, the AutoProcessor will handle resampling and conversion when you pass in the audio data. For example, you can pass a Blob or ArrayBuffer of WAV audio to the `transcriber` pipeline and it will internally decode and resample it. If you use your own pipeline, you might use an offline audio context or a library to resample to 16 kHz PCM. - **Audio format**: Ensure you provide the model with raw audio samples (PCM Float32Array) or a WAV Blob that the library can decode. The HF pipeline supports passing a Blob/URL directly [34] [35]. If you have raw float samples from an AudioWorklet, you can pass that Float32Array to the transcriber as well (Transformers.js accepts Float32Array input) [35].

C. Running ASR and diarization in parallel: Once audio data is ready, you will run two inferences: 1. **Transcription model (ASR)** – e.g. Whisper model to get text and word timestamps. 2. **Diarization model** – e.g. Pyannote segmentation model to get speaker turn timings.

Using the Transformers.js pipeline, you can run these asynchronously. In the reference implementation, they invoked the Whisper pipeline and the Pyannote model *simultaneously* on the full audio input [36] [37], since diarization doesn't depend on the transcript. The code looked roughly like:

```
// Pseudocode: run ASR and diarization together
const [transcript, speakerSegments] = await Promise.all([
    transcriber(audioData, { return_timestamps: 'word', language: 'en',
chunk_length_s: 30 }),
    segmenter(audioData)
]);
```

Here, `transcriber` is the Whisper pipeline and `segmenter` is a function that uses the Pyannote model to produce segments. The `chunk_length_s: 30` option tells Whisper to process the audio in chunks of 30s internally (for long inputs), which helps memory usage [38]. The diarization `segmenter` would use the `AutoModelForAudioFrameClassification` for the segmentation model and then a post-processing step to generate speaker clusters. In the HF example, the segmentation processor's `post_process_speaker_diarization` method handled clustering and returned an array of speaker-labeled segments [39] [40]. Each segment includes a start time, end time, speaker ID, and (if available) confidence.

If you implement manually, a basic diarization algorithm is: - Run a **Voice Activity Detection (VAD)** or segmentation model to break the audio into homogeneous segments where one speaker is talking. - For each segment, compute a **speaker embedding** (a vector representing the speaker's voice). There are pre-trained models for this (e.g. ECAPA-TDNN in pyannote, or even simpler MFCC+PCA). - Cluster the embeddings (e.g. K-means or spectral clustering) to group segments by speaker. Assign a label (Speaker 1, Speaker 2, etc.) to each cluster. - You can estimate the number of speakers via clustering algorithms or assume a reasonable maximum. In many cases, the number of distinct speakers is small (2-4) and can be determined by the clustering algorithm automatically.

Pyannote's `speaker-diarization` pipeline essentially automates the above steps with a learned model for segmentation and an internal clustering for the final labels [41] [42]. By using their model, you offload the heavy logic to a well-tested solution. The output of Pyannote's processing will be segments like: e.g. `[ { start: 0.0, end: 5.2, speaker: "SPEAKER_0" }, { start: 5.2, end: 10.1, speaker: "SPEAKER_1" }, ... ]` covering the whole audio.

D. Merging transcript with speaker labels: Now, combine Whisper's output with the diarization segments. Whisper (when `return_timestamps:'word'`) gives timing for each word (or each chunk of text) [43] [44]. To label each transcript segment: - Iterate through the words (or sentence segments) with their timestamps. - For each word timestamp, find which speaker segment encompasses that time. - Assign that speaker label to the word. If using full sentence segments, you might assign the speaker of the majority of words in that segment. - The result can be aggregated into an output transcript such as: - **Speaker 1:** Hello, how are you? - **Speaker 2:** I'm fine, thanks. And you?

This was done in the demo's front-end by taking the `result.transcript` (text + word timings) and `result.segments` (speaker segments) and aligning them [42] [45]. Since both have accurate timestamps, a simple overlap check works. An important detail is to handle speaker changes that occur mid-sentence – you may need to split a Whisper segment if a speaker change occurs in the middle (this can happen if two people spoke over each other or interrupted). Using word-level timestamps makes this manageable: you can change the speaker label whenever the diarization segment id changes while iterating through words.

E. Displaying transcriptions with speaker labels: Finally, present the labeled transcript in the UI. You can prepend the speaker label to each line or use color-coding for each speaker. For example, render each segment as `<p><strong>Speaker 1:</strong> [transcript text]</p>`. If you want live updating (streaming text display), you can push interim results to the UI as each chunk is processed. For instance, if you transcribe every 5 seconds, you can append the newest text with the speaker label once it's ready. In a real-time scenario, update the UI in smaller increments (word by word or every second). Just ensure you keep track of speaker turns – when a new speaker segment starts, break the line and label the new speaker.

Code example – using Transformers.js pipelines: To illustrate, here's a simplified snippet using Transformers.js for a one-shot transcription of an `audioBlob` with diarization:

```
import { pipeline, AutoModelForAudioFrameClassification, AutoProcessor } from
'@xenova/transformers';

// Load models (this will download ONNX models on first run and cache them)
const transcriber = await pipeline('automatic-speech-recognition', 'onnx-
community/whisper-base_timestamped');
const diarizerModel = await
AutoModelForAudioFrameClassification.from_pretrained('onnx-community/pyannote-
segmentation-3.0');
const diarizerProcessor = await AutoProcessor.from_pretrained('onnx-community/
pyannote-segmentation-3.0');

// Prepare audio (assuming audioBlob is a Blob from MediaRecorder or input file)
const audioData = await audioBlob.arrayBuffer();  // get raw data
// Run ASR and diarization
const [asrResult, diarization] = await Promise.all([
    transcriber(new Float32Array(audioData), { return_timestamps: 'word' }),
    (async () => {
        // Preprocess and run diarization model
        const inputs = await diarizerProcessor(audioData);
        const { logits } = await diarizerModel(inputs);
        return diarizerProcessor.post_process_speaker_diarization(logits,
inputs.tensor.shape[1])[0];
    })()
]);
// `asrResult.text` contains the transcription and asrResult.chunks the words
with timestamps.
// `diarization` is an array of segments with start, end, and speaker IDs.

// Merge results (simple approach):
let finalTranscript = "";
for (let chunk of asrResult.chunks) {
    const wordStart = chunk.timestamp[0];
    // find speaker for this word's timestamp
    let speaker = diarization.find(seg => wordStart >= seg.start && wordStart <
```

```
seg.end)?.label || "Speaker ?";
    finalTranscript += `${speaker}: ${chunk.text}\n`;
}
console.log(finalTranscript);
```

This is a rough outline – in practice you'd need to refine the merging (group words by continuous speaker, etc.), but it shows the general flow. The key point is that using high-level libraries greatly simplifies processing: e.g., `transcriber()` handles audio decoding, mel spectrogram creation, and the Whisper model inference internally; `diarizerProcessor.post_process_speaker_diarization` handles the logic of turning frame-level outputs into speaker segments [39] [40].

Note on real-time vs batch: If you need real-time streaming transcription (continuous update while speaking), you will have to feed the model audio in smaller increments (since Whisper isn't inherently a streaming model, you might implement a sliding window approach with overlapping context). Libraries like **Sherpa-ONNX** provide streaming ASR models (e.g. RNNT/Transducer-based) that can output text in real-time with partial results [28] [46]. However, integrating diarization in streaming is tricky – usually diarization is done after the fact or on larger windows, because clustering speakers requires some history. A practical approach is **semi-real-time diarization**: e.g., decide speaker labels on 5-second windows. For most applications, a slight delay in assigning the correct speaker label is acceptable. You might show interim transcripts without labels, then update with speaker labels once the segment is long enough to determine the speaker. This keeps the UX fluid.

Web Workers: It's highly recommended to run the heavy inference in a Web Worker thread (or Worklet for audio) so that the main UI thread remains responsive. The example above from the HF demo used a Web Worker (`worker.js`) to do the pipeline loading and inference, posting messages back to the UI with results [47] [48]. This way, even if the transcription takes a couple of seconds, the UI won't freeze and you can show progress. Chrome on Android supports web workers and they are essential for offloading WASM execution from the main thread.

By following these steps, you'll capture audio, transcribe it, diarize it, and display speaker-labeled text – all within the browser and fully offline.

7. Performance Tips for Android Tablets and Chrome

Running on an Android tablet (or any mobile device) introduces performance constraints like lower CPU power, possible lack of active cooling (thermal throttling), and memory limits. Here are **performance tips and considerations** to ensure the system runs smoothly on Chrome for Android (as well as low-end laptops):

- **Choose the right model size**: As mentioned, prefer Whisper Tiny or Base for tablets. The *AssemblyAI* benchmark specifically notes that for mobile/edge deployment, **Vosk or Whisper Tiny** are recommended due to their minimal resource usage and offline capability [49]. Whisper Tiny will use far less computation than Whisper Small/Medium. If multilingual support isn't needed, the `.en` models (English-only) are 2x smaller than the multilingual ones (e.g. `tiny.en` is ~75 MB memory vs ~145 MB for multilingual tiny). For diarization, the 6 MB Pyannote segmentation is fine; you could also downsample audio more for it or reduce its frame rate to lighten load if needed.

- **Quantization**: Use quantized models where possible. The ONNX models by Xenova often come pre-quantized (8-bit). For example, when running Whisper in WASM mode, the demo used 8-bit weights (`dtype: 'q8'`) to speed up inference [50] [51]. Quantization can dramatically reduce CPU usage at a small cost to accuracy. Similarly, if you compile whisper.cpp for WASM, use 4-bit or 8-bit quantized GGML models. This can reduce model size (memory and download) and inference time. Just ensure the library you use supports those quantized formats.
- **Leverage WebAssembly SIMD and multithreading**: Chrome supports WASM SIMD and threading (with SharedArrayBuffer and cross-origin isolation enabled). Ensure your app is served in a way that allows threads (for example, using `Cross-Origin-Opener-Policy: same-origin` and `Cross-Origin-Embedder-Policy: require-corp` headers). ONNX Runtime Web can then use multiple threads to compute in parallel, and whisper.cpp WASM can spawn worker threads for different layers. This yields significant speed-ups on multi-core devices. However, more threads = more CPU heat, so consider a balance (e.g., use 2-3 threads on a 4-core mobile chip to leave some headroom).
- **Use WebGPU/WebGL if available**: Many Android devices have decent GPUs. If the user's Chrome supports WebGPU (currently as of 2025 it might be available behind a flag or in beta on Android), using it can accelerate matrix multiplications on the GPU which is more power-efficient for large models. The Transformers.js library will try WebGPU first by default for Whisper (as configured in the demo) [52] [15]. WebGL can also be used but it's less flexible for ML and might be slower than optimized WASM for some workloads. Still, it's worth testing – ONNX Runtime's WebGL backend could offload some operations from the CPU.
- **Manage audio chunk size and overlap**: On mobile, you don't want to buffer extremely long audio before transcribing (memory and compute cost grow with length). It's better to process in rolling chunks (e.g. 30 seconds at a time). Whisper transcribing a full 5-minute audio on a tablet in one go could cause a long freeze. Instead, process incrementally so that each inference call stays under e.g. ~5-10 seconds of audio. Use the `chunk_length_s` parameter (if using HF pipeline) to have the model internally segment long audio, which avoids extremely large memory usage [53].
- **Thermal considerations**: Continuous real-time transcription is CPU/GPU intensive and might heat up a tablet. To mitigate this:
- Insert small delays or yielding in between chunks to give the device a breather (e.g., transcribe 5 seconds, then wait 1 second before next chunk if not urgent).
- Monitor the processing time and if it starts increasing (a sign of throttling), consider lengthening the chunk interval or reducing model complexity.
- Ensure you're not doing unnecessary work – e.g., if silence is detected via a VAD, you can skip running Whisper on that segment to save computation.
- **Memory management**: On Android Chrome, memory is limited. Free model resources if not needed. Transformers.js caches the model, but you can call `model.dispose()` if you know you won't use it again for a while (just be prepared to load again). Avoid holding onto huge audio buffers – process and then drop them. Using streaming approaches helps here.
- **Optimize JavaScript**: Use Web Workers for heavy work, as mentioned, to keep the UI smooth. Also, avoid heavy DOM updates for every word (batch updates if possible, e.g., append transcript per sentence rather than per character). This prevents the UI thread from lagging.
- **Test on target devices**: Profile the transcription on an actual Android tablet using Chrome dev tools remote debugging. Check the timing of inference and the CPU usage. This will guide you to tune thread counts or chunk sizes. Each device may behave differently (for instance, an M1 MacBook can easily handle Whisper Small in real-time, but an entry-level Android tablet might struggle with anything above Whisper Tiny).

In summary, the system **can run in near-real-time on Android Chrome** with careful optimization. Many have successfully run offline Whisper on phones – one in-browser demo processed ~1 minute of audio on a mid-range device with good accuracy [54]. By using smaller models, quantization, and the aforementioned techniques, you'll avoid freezes and excessive heat. The entire pipeline (ASR + diarization) will be local, so no network latency – the only latency is computation time, which we keep within a few seconds for practical real-time use.

Sources:

- AssemblyAI – *"Top 8 open source STT... in 2025"* (accuracy, model sizes, and use-case guidance) [3] [9] [49]
- Hugging Face – Whisper diarization demo code (local Transformers.js pipeline with Whisper Base & Pyannote) [5] [16]
- Hugging Face – *Xenova/whisper-tiny.en* model card (usage of Transformers.js pipeline) [18] [19]
- Hugging Face – Transformers.js diarization worker (parallel ASR+diarization, model config) [38] [37]
- Picovoice Blog – *"Add Speaker Diarization to Whisper in C++"* (Whisper.cpp with Falcon, concept of merging speaker labels) [55] [56]
- MDN Web Docs – MediaDevices.getUserMedia (microphone capture API) [31] [32]
- Reddit – *Whisper Diarization Web* discussion (confirmation of local browser inference with Transformers.js) [57].

---

[1] [7] [8] [33] [41] [42] [45] Whisper and Pyannote: The Ultimate Solution for Speech Transcription
https://scalastic.io/en/whisper-pyannote-ultimate-speech-transcription/

[2] [3] [4] [9] [49] [53] Top 8 open source STT options for voice applications in 2025
https://www.assemblyai.com/blog/top-open-source-stt-options-for-voice-applications

[5] [6] [16] [20] [30] whisper-speaker-diarization/src/App.jsx · Xenova/whisper-speaker-diarization at main
https://huggingface.co/spaces/Xenova/whisper-speaker-diarization/blob/main/whisper-speaker-diarization/src/App.jsx

[10] [12] Whisper OpenAI In-Browser for Offline Audio Transcription | Data Science Collective
https://medium.com/data-science-collective/implementing-whisper-openai-in-browser-for-offline-audio-transcription-adab61be7af7

[11] Whisperlivekit for real time speech to text translation - Facebook
https://www.facebook.com/groups/mojolang/posts/1783750242274960/

[13] [14] [15] [36] [37] [38] [39] [40] [47] [48] [50] [51] [52] whisper-speaker-diarization/src/worker.js · Xenova/whisper-speaker-diarization at main
https://huggingface.co/spaces/Xenova/whisper-speaker-diarization/blob/main/whisper-speaker-diarization/src/worker.js

[17] GitHub - Programming-from-A-to-Z/transformers-js-examples
https://github.com/Programming-from-A-to-Z/transformers-js-examples

[18] [19] [23] [24] [25] [43] [44] Xenova/whisper-tiny.en · Hugging Face
https://huggingface.co/Xenova/whisper-tiny.en

[21] whisper.cpp : WASM example - ggml.ai
https://ggml.ai/whisper.cpp/

22 @remotion/whisper-web - npm
https://www.npmjs.com/package/@remotion/whisper-web

26 55 56 Step-by-Step Guide: Add Speaker Diarization to OpenAI Whisper in C++
https://picovoice.ai/blog/whisper-cpp-speaker-diarization/

27 pengzhendong/pyannote-onnx - GitHub
https://github.com/pengzhendong/pyannote-onnx

28 29 46 GitHub - k2-fsa/sherpa-onnx: Speech-to-text, text-to-speech, speaker diarization, speech enhancement, source separation, and VAD using next-gen Kaldi with onnxruntime without Internet connection. Support embedded systems, Android, iOS, HarmonyOS, Raspberry Pi, RISC-V, RK NPU, Ascend NPU, x86_64 servers, websocket server/client, support 12 programming languages
https://github.com/k2-fsa/sherpa-onnx

31 32 MediaDevices: getUserMedia() method - Web APIs | MDN
https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia

34 35 javascript - How to transcribe local audio File/Blob with Transformers.js pipeline? (JSON.parse error) - Stack Overflow
https://stackoverflow.com/questions/79696758/how-to-transcribe-local-audio-file-blob-with-transformers-js-pipeline-json-par

54 57 Whisper Diarization Web: In-browser multilingual speech recognition with word-level timestamps and speaker segmentation : r/LocalLLaMA
https://www.reddit.com/r/LocalLLaMA/comments/1e9nux8/whisper_diarization_web_inbrowser_multilingual/