



Transcribe Client Audio Without Sending It Anywhere

How the Private Whisper Transcriber runs entirely in your browser, and why that matters for privilege

You have a recording. A client phone call you were allowed to record, a voice memo you dictated in the car, a hearing you captured for your own notes, a recorded statement from a witness. You want it as text. The easy path is a cloud transcription service: upload the file, wait a minute, download the transcript. The problem with the easy path is the word "upload." You just handed a recording of your client to a company you have never vetted, whose servers you cannot see, under terms of service you did not read.

There is a better way, and it is closer to free than you would guess. A modern browser can now run a real speech-recognition model itself, on your machine, with the audio never leaving the device. I built exactly this, a single HTML file I call the Private Whisper Transcriber. You double-click it; it downloads a speech model once, then transcribes everything locally, forever, offline. This tip walks through the actual code so you can understand what you are running, build your own, or know the right questions to ask when you commission one.

What "on-device" actually means

Most software you think of as "an app" is really a thin window onto someone else's computer. When you transcribe in the cloud, your file travels to a server farm, gets processed there, and the text comes back. On-device flips that. The model, the computation, and the storage all live on the machine in front of you. Nothing is transmitted. If you pulled your network cable, the tool would keep working.

For a lawyer that distinction is not a nicety; it is the whole ballgame. Audio that never leaves your laptop was never disclosed to a third party. There is no vendor data-retention question, no "who else can subpoena this," no terms-of-service clause quietly claiming a license to your upload. I will come back to the ethics; first, how it is possible at all.

The three pieces that make it work

Three things had to become true at once, and around 2024 they did.

A speech model you are allowed to run. OpenAI released Whisper, a speech-to-text model, with open weights. "Open weights" means the trained model itself is downloadable and runnable by

anyone, not locked behind an API. It comes in sizes; the small ones are a few tens of megabytes and good enough for clean speech, the larger ones are more accurate and heavier to download.

A way to run it in a browser. A library called Transformers.js loads models like Whisper and runs them using WebAssembly and, where available, WebGPU. Plain English: WebAssembly lets a browser run heavy computation at near-native speed, and WebGPU lets it borrow your graphics card the way a real desktop program would. So the same model that used to need a Python setup now runs from a web page.

A place to keep the results locally. Browsers ship with a built-in database called IndexedDB. It stores data on your disk, in your browser, tied to that page. The Private Whisper Transcriber uses it to hold every transcript so your work survives closing the tab, with nothing synced anywhere.

Put those together and you get a transcription tool with no server, no account, no subscription, and no upload.

The code up close

You do not need to be a programmer to follow these. Each block below is real code from the Private Whisper Transcriber, and I explain what every part is doing and why.

One: run the model in a background worker, and let it handle long audio. Transcribing a long file is heavy work, and if you do it on the page's main thread the whole tab freezes. So the app spins up a Web Worker, a second thread of JavaScript, and runs the model there. The worker is handed the audio and calls the model like this:

```
const result = await transcriber(audio, {
  chunk_length_s: options.chunkLength,      // process in ~30s windows
  stride_length_s: options.strideLength,    // that OVERLAP by a few seconds
  return_timestamps: options.returnTimestamps,
  language: options.language || null,
  task: options.task,
  force_full_sequences: false,
  chunk_callback: (chunk) => { self.postMessage({ type: "chunk", chunk }); },
});
```

The two lines that matter most are `chunk_length_s` and `stride_length_s`. Rather than chopping the audio into hard thirty-second blocks myself, I hand the whole thing to the library and let it slide a window across the audio with a few seconds of overlap between windows. That overlap is what keeps a word spoken across a seam from getting dropped or doubled. This is the single most common mistake in home-built transcribers, and the fix is to not cut the audio yourself. `chunk_callback` streams each finished window back to the page so you watch the transcript appear live.

The audio is handed to the worker without copying it, using a "transfer" so a two-hour recording does not briefly exist twice in memory:

```
worker.postMessage({ type: "transcribe", audio, metadata, options },
[audio.buffer]);
```

That trailing `[audio.buffer]` is the transfer list; it moves the audio to the worker instead of duplicating it.

Two: turn the audio file into numbers the model expects. Whisper does not want an MP3; it wants raw sound as a long list of numbers, sampled 16,000 times a second, in a single channel. A stereo voice memo or a video has to be decoded, mixed to mono, and resampled first.

```
async function decodeToMono16k(file) {
  const arrayBuffer = await file.arrayBuffer();
  const context = new AudioContext();
  try {
    const audioBuffer = await context.decodeAudioData(arrayBuffer.slice(0));
    const mono = mixToMono(audioBuffer);
    const resampled = resampleLinear(mono, audioBuffer.sampleRate,
TARGET_SAMPLE_RATE);
    return { samples: resampled, duration: audioBuffer.duration,
originalSampleRate: audioBuffer.sampleRate };
  } finally {
    try { await context.close(); } catch (_) {}    // release the audio engine
every time
  }
}
```

`arrayBuffer()` hands back the file's raw bytes. `decodeAudioData` is the browser's own audio engine unpacking those bytes into actual sound samples; it handles MP3, MP4, WAV, and the rest for free. Then two small steps that every speech tool needs and no demo mentions: fold the channels down to one, and step the rate to 16,000, which is the rate Whisper was trained on. Notice the `finally` block closes the audio engine no matter what; browsers cap how many you can open, so leaking them is a real bug I made sure to avoid.

The mono fold averages every channel, not just the first two, so a center-channel voice in a five-channel file is not lost:

```
function mixToMono(audioBuffer) {
  const channels = audioBuffer.numberOfChannels;
  const length = audioBuffer.length;
  const mono = new Float32Array(length);
  for (let channel = 0; channel < channels; channel += 1) {
    const data = audioBuffer.getChannelData(channel);
    for (let i = 0; i < length; i += 1) mono[i] += data[i] / channels; //
  }
  return mono;
}
```

Three: sanitize the settings before they reach the model. Every run reads the dropdowns and number boxes, and this is where sloppy input handling bites. This function is the single place both the worker and the fallback path read from, so it is the right place to enforce sane values.

```
function getRunOptions() {
  const chunkLength = clamp(numOr(els.chunkInput.value, 30), 5, 30);
  let strideLength = clamp(numOr(els.strideInput.value, 5), 0, 10);
  strideLength = Math.min(strideLength, Math.max(0, chunkLength - 1));
  const model = els.modelSelect.value;
  const device = els.deviceSelect.value;
  return {
    model,
    device,
    dtype: effectiveDtype(device, els.dtypeSelect.value), // gate precision
    task: effectiveTask(model, els.taskSelect.value), // gate translate
    language: els.languageInput.value.trim(),
    chunkLength,
    strideLength,
    returnTimestamps: els.timestampToggle.checked,
  };
}
```

`numOr` reads a number field but falls back to a default for a blank or garbage entry, and it does not mistake a legitimate zero for "empty," a trap I walked into and fixed. `clamp` keeps every value inside a safe range, and the stride is forced to stay smaller than the chunk so the overlap can never exceed the window. `effectiveDtype` and `effectiveTask` are two guards I added after testing: the first refuses precision settings a given device cannot run reliably, and the second refuses to ask an English-only model to translate. Guarding at the one chokepoint means every code path downstream is safe by construction.

Four: keep everything on disk, locally. After transcription, each transcript record goes into IndexedDB, the browser's built-in database. This is what makes it feel like an app instead of a toy: close the tab, come back next week, your transcripts are still there. And still only there.

```
const store = db.createObjectStore(STORE_NAME, { keyPath: "id", autoIncrement: true });
store.createIndex("createdAt", "createdAt", { unique: false });
store.createIndex("fileName", "fileName", { unique: false });
```

```
async function addRecord(record) {
  const db = await openDB();
  return new Promise((resolve, reject) => {
    const tx = db.transaction(STORE_NAME, "readwrite");
    const request = tx.objectStore(STORE_NAME).add(record);
    request.onsuccess = () => resolve(request.result); // returns the new id
    request.onerror = () => reject(request.error);
  });
}
```

The store keeps each transcript under an auto-numbered id, with indexes on date and filename so the history list can sort and search. `addRecord` writes one record and hands back its id. Nothing here touches a network; the data lives in this browser, on this machine, until you delete it, which is precisely the property that matters for confidential material.

Five: export, without a round trip. Every export is built from the transcript text right in the browser and saved with a fake link click. Here is the plain-text export and the little helper that saves any file:

```
async function exportText(record) {
  const text = els.transcriptArea.value || record.formattedText || record.text
  || "";
  const header = makeExportHeader(record);
  const blob = new Blob(`[${header}\n\n${text}\n`], { type: "text/
plain;charset=utf-8" });
  downloadBlob(blob, `${baseExportName(record)}.txt`);
}

function downloadBlob(blob, filename) {
  const url = URL.createObjectURL(blob);
  const a = document.createElement("a");
  a.href = url;
  a.download = filename;
  document.body.appendChild(a);
  a.click(); // the browser saves it, no
server involved
  a.remove();
  setTimeout(() => URL.revokeObjectURL(url), 2000); // then release the
temporary handle
}
```

`exportText` assembles a header plus the transcript into a `Blob`, which is just a file held in memory. `downloadBlob` makes a temporary link pointing at that blob, clicks it so the browser saves the file, then releases the handle. The Word export works the same way; it uses a document library to build the `.docx` and saves it with the same helper. Every byte is assembled and delivered on your computer.

Those pieces are ninety percent of the tool. The rest is buttons, a progress bar, and the history list.

What I found and fixed by stress-testing my own build

Anyone who tells you a transcription tool is a weekend project is skipping the parts that bite. I pulled the logic apart and tested it, and I fixed six real problems before I would trust it with a file. I am listing them because they are exactly the things to check in any build like this, yours or a vendor's.

You could not turn overlap off. The stride field read its value with a common shortcut that treats zero as "empty" and quietly substitutes the default. So a user asking for no overlap silently got five seconds. The fix was a small helper that only falls back for a truly blank or non-numeric entry.

The plain-text copy ate real annotations. When you edit and save, the app derives a clean version by stripping the leading timestamps. My first version stripped anything in brackets at the start of a line, which also deleted a real `[Inaudible]` or `[CROSSTALK]` the model had emitted. I tightened it to remove only brackets that actually look like a timestamp.

The GPU could be handed a setting it chokes on. Whisper is unreliable at the smallest compressed precisions on a graphics card. I now gate the precision setting to the device: float precision on the GPU, the compact quantized options on the CPU, so an incompatible combination cannot be selected.

You could ask an English-only model to translate. That is an incoherent request. I disabled the translate option whenever an English-only model is chosen.

The progress bar could look frozen. During a long stretch with no finished window to report, the bar sat still and the tool looked hung. I added a slow heartbeat that nudges it so you can tell work is happening.

It might not run from a double-clicked file. The background worker imports code from the internet, and some browsers block that when the page is opened directly from disk rather than served. I added a fallback that runs the model on the main thread in that case, and a note suggesting you serve the file with a one-line local web server for the smoothest experience.

None of these are reasons not to build the tool. They are the reasons to test it before you trust it, and to keep a human reviewing the output, which you were going to do anyway.

Why this belongs in a law practice

Your duty of confidentiality does not have a "but the app was convenient" exception. Under Missouri Rule 4-1.6, and its equivalent in every state, information relating to the representation stays protected, and handing it to an outside vendor is a disclosure you have to be able to justify. Upload a client's recorded call to a cloud transcription service and you have shared it with that company, its subprocessors, and whoever its retention policy and its next security incident expose it to. You may need the client's informed consent, you may need to vet the vendor's data terms, and you have created one more copy of privileged material in a place you do not control.

An on-device tool removes the question instead of answering it. The audio is decoded, transcribed, and stored on the same machine, and the transcript is deleted when you delete it. There is no third party because there is no transmission. That is the same instinct behind running a privilege check before you paste anything into a public AI chatbot: the safest disclosure is the one that never happens. It also means you can transcribe on a plane, in a courthouse hallway with no signal, or on a matter sensitive enough that "it went to a cloud vendor" is an answer you never want to give.

The trade-offs are real and worth stating plainly. On-device transcription is only as fast as your own hardware, the smaller models are less accurate than the best cloud services, and you own the maintenance. For a solo or small firm handling confidential audio, that trade is usually easy.

Using AI to help with this. You do not have to write this from scratch. Describe what you want and let the model scaffold it: "Build me a single HTML file that uses Transformers.js and Whisper to transcribe an audio file entirely in the browser, in a Web Worker, store results in IndexedDB, and export to text and Word. No server, no upload." Then hand it any function above and ask it to explain the setup line by line, or ask it to harden the parts I named: overlapping chunks instead of hard cuts, gating precision to the device, and preserving bracketed annotations when you strip timestamps. Paste it your ugliest transcript and ask where it would break.

The playbook

The idea in five steps:

1. Use an open-weights speech model (Whisper) so you are allowed to run it yourself.
2. Run it in the browser with Transformers.js, in a Web Worker, on your GPU when you have one.
3. Convert audio to 16k mono, and let the library transcribe in OVERLAPPING chunks so no words fall in the seams.
4. Store everything locally in IndexedDB; export to text or Word on your own machine.
5. Because nothing uploads, the client audio was never disclosed. That is the point.

Understand those five and you can build it, buy it wisely, or explain to a colleague why the convenient cloud button is the expensive choice.