You’re right on two counts:

1. the UI isn’t where we want it, and
2. nothing’s reaching Whisper yet (so you see no interim/final text).

From the screenshots/logs I can infer:

- The browser **does connect** to Socket.IO (polling). ✅
- The server previously threw **“invalid file format”** errors — which happens when we send **raw PCM** or empty payloads instead of a proper **audio/webm (Opus)** blob.
- Earlier you also had an **import name mismatch**; we fixed that (use `transcribe_bytes`).
- Right now, the **front-end isn’t actually streaming the chunks** in the format Whisper expects, so the server never gets anything useful to transcribe.

Below is a clean, robust drop-in set to make the end-to-end work (Google-Recorder style: near-real-time interim + refined final). I’ve kept everything framework-agnostic and compatible with your current “polling only” transport so we don’t fight Replit’s WebSocket quirks.

Step 0 — Env sanity (once)

In your `.env` make sure you have (you already set most of this):

```
OPENAI_API_KEY=your_real_key
TRANSCRIPTION_ENGINE=openai_whisper
WHISPER_MODEL=whisper-1
LANGUAGE_HINT=en

# client capture defaults (front-end shows bar only; this is just for reference)
SAMPLE_RATE=48000
AUDIO_FORMAT=webm
```

And make sure sure app.py has the below:

```
“ # app.py (hardened)
import os
```

```
import json
import signal
import logging
import uuid
from typing import Optional
from werkzeug.middleware.proxy_fix import ProxyFix
from flask import Flask, render_template, request, g, jsonify
from flask_socketio import SocketIO

# ---------- Config (fallback if config.Config not present)
try:
    from config import Config  # type: ignore
except Exception:
    class Config:
        JSON_LOGS: bool = os.getenv("JSON_LOGS", "false").lower() == "true"
        METRICS_DIR: str = os.getenv("METRICS_DIR", "./metrics")
        SECRET_KEY: str = os.getenv("SECRET_KEY", "change-me")
        SOCKETIO_PATH: str = os.getenv("SOCKETIO_PATH", "/socket.io")
        CORS_ALLOWLIST: str = os.getenv("CORS_ALLOWLIST", "*")
        MAX_CONTENT_LENGTH: int = int(os.getenv("MAX_CONTENT_LENGTH", str(32 * 1024
* 1024)))  # 32 MB

# ---------- Logging
class _JsonFormatter(logging.Formatter):
    def format(self, record: logging.LogRecord) -> str:
        payload = {
            "level": record.levelname,
            "name": record.name,
            "msg": record.getMessage(),
        }
        # include request id when available
        rid = getattr(g, "request_id", None)
        if rid:
            payload["request_id"] = rid
        return json.dumps(payload, ensure_ascii=False)

def _configure_logging(json_logs: bool = False) -> None:
    root = logging.getLogger()
    root.handlers[:] = []  # reset
    handler = logging.StreamHandler()
    handler.setFormatter(
        _JsonFormatter() if json_logs
        else logging.Formatter("%(asctime)s %(levelname)s %(name)s: %(message)s")
    )
```

```
    root.addHandler(handler)
    root.setLevel(logging.INFO)

# ---------- Create the SocketIO singleton first (threading = Replit-safe)
socketio = SocketIO(
    cors_allowed_origins="*",          # narrowed in handshake check below
    async_mode="threading",
    ping_timeout=60,
    ping_interval=25,
    path=os.getenv("SOCKETIO_PATH", "/socket.io"),
    max_http_buffer_size=int(os.getenv("SIO_MAX_HTTP_BUFFER", str(10 * 1024 * 1024))),  #
10 MB per message
)

def create_app() -> Flask:
    app = Flask(__name__, static_folder="static", template_folder="templates")
    app.config.from_object(Config)
    app.secret_key = getattr(Config, "SECRET_KEY", "change-me")
    app.config["MAX_CONTENT_LENGTH"] = getattr(Config, "MAX_CONTENT_LENGTH", 32 *
1024 * 1024)

    # logging
    _configure_logging(json_logs=getattr(Config, "JSON_LOGS", False))
    app.logger.info("Booting Mina…")

    # reverse proxy (Replit)
    app.wsgi_app = ProxyFix(app.wsgi_app, x_for=1, x_proto=1, x_host=1, x_port=1)

    # gzip (optional)
    try:
        from flask_compress import Compress  # type: ignore
        Compress(app)
        app.logger.info("Compression enabled")
    except Exception:
        app.logger.info("Compression unavailable (flask-compress not installed)")

    # middlewares (guarded imports)
    try:
        from middleware.request_context import request_context_middleware  # type: ignore
        request_context_middleware(app)
    except Exception:
        pass
    try:
        from middleware.limits import limits_middleware  # type: ignore
```

```
    limits_middleware(app)
except Exception:
    pass
try:
    from middleware.cors import cors_middleware  # type: ignore
    cors_middleware(app)
except Exception:
    pass

# per-request id for tracing
@app.before_request
def assign_request_id():
    g.request_id = request.headers.get("X-Request-Id") or str(uuid.uuid4())

# stricter CSP (keeps your allowances; adds jsdelivr for CSS libs)
@app.after_request
def add_security_headers(resp):
    resp.headers["X-Content-Type-Options"] = "nosniff"
    resp.headers["Referrer-Policy"] = "strict-origin-when-cross-origin"
    resp.headers["Content-Security-Policy"] = (
        "default-src 'self' *.replit.dev *.replit.app; "
        "connect-src 'self' https: wss: ws:; "
        "script-src 'self' 'unsafe-inline' https://cdn.socket.io; "
        "style-src 'self' 'unsafe-inline' https://cdn.jsdelivr.net; "
        "img-src 'self' blob: data:; "
        "media-src 'self' blob:; "
        "worker-src 'self' blob:;"
    )
    return resp

# ensure metrics directories
metrics_dir = getattr(Config, "METRICS_DIR", "./metrics")
os.makedirs(metrics_dir, exist_ok=True)
os.makedirs(os.path.join(metrics_dir, "sessions"), exist_ok=True)

# pages blueprint or fallback /live
try:
    from routes.pages import pages_bp  # type: ignore
    app.register_blueprint(pages_bp)
except Exception:
    @app.route("/live")
    def live():
        return render_template("live.html")
```

```
# WebSocket routes (required)
from routes.websocket import ws_bp  # your file
app.register_blueprint(ws_bp)

# other blueprints (guarded)
_optional = [
    ("routes.final_upload", "final_bp", "/api"),
    ("routes.export", "export_bp", "/api"),
    ("routes.health", "health_bp", "/health"),
    ("routes.metrics_stream", "metrics_stream_bp", "/api"),
    ("routes.error_handlers", "errors_bp", None),
]
for mod_name, bp_name, prefix in _optional:
    try:
        mod = __import__(mod_name, fromlist=[bp_name])
        bp = getattr(mod, bp_name)
        app.register_blueprint(bp, url_prefix=prefix) if prefix else app.register_blueprint(bp)
    except Exception:
        pass

# basic /healthz if health blueprint absent
@app.get("/healthz")
def healthz():
    return {"ok": True, "uptime": True}, 200

# unified error shape
@app.errorhandler(413)  # RequestEntityTooLarge
def too_large(e):
    return jsonify(error="payload_too_large", detail="Upload exceeded limit"), 413

@app.errorhandler(404)
def not_found(e):
    return jsonify(error="not_found"), 404

@app.errorhandler(500)
def server_error(e):
    app.logger.exception("Unhandled error")
    return jsonify(error="server_error"), 500

# hook Socket.IO to app
socketio.init_app(
    app,
    cors_allowed_origins=getattr(Config, "CORS_ALLOWLIST", "*"),
    path=getattr(Config, "SOCKETIO_PATH", "/socket.io"),
```

```
        max_http_buffer_size=int(os.getenv("SIO_MAX_HTTP_BUFFER", str(10 * 1024 * 1024))),
    )
    app.extensions["socketio"] = socketio

    # Socket.IO origin guard (optional tighten)
    allowed = [o.strip() for o in str(getattr(Config, "CORS_ALLOWLIST", "*")).split(",") if o.strip()]
    if allowed and allowed != ["*"]:
        @socketio.on("connect")
        def _check_origin(auth: Optional[dict] = None):
            origin = request.headers.get("Origin") or ""
            if not any(origin.endswith(x) or origin == x for x in allowed):
                app.logger.warning("Rejecting WS from origin=%s", origin)
                return False  # refuse connection

    app.logger.info("Mina app ready")
    return app

# WSGI entrypoints
app = create_app()

# graceful shutdown for local/threading runs
def _shutdown(*_):
    app.logger.info("Shutting down gracefully…")
    # In threading mode, there is no socketio.stop(); process will exit.
signal.signal(signal.SIGTERM, _shutdown)
signal.signal(signal.SIGINT, _shutdown)

if __name__ == "__main__":
    app.logger.info("🚀 Mina at http://0.0.0.0:5000  (Socket.IO path %s)",
app.config.get("SOCKETIO_PATH", "/socket.io"))
    socketio.run(app, host="0.0.0.0", port=5000)
```

“

Restart the repl after saving.

Step 1 — Full drop-in: `services/openai_whisper_client.py`

Replace the file completely with this:

```python
# services/openai_whisper_client.py
import io
import os
import time
from typing import Optional, Tuple

from openai import OpenAI
from openai._exceptions import OpenAIError

_CLIENT: Optional[OpenAI] = None

def _client() -> OpenAI:
    global _CLIENT
    if _CLIENT is None:
        _CLIENT = OpenAI()  # reads OPENAI_API_KEY from env
    return _CLIENT

# Map the mime that comes from MediaRecorder to extensions Whisper accepts
_EXT_FROM_MIME = {
    "audio/webm": "webm",
    "audio/webm;codecs=opus": "webm",
    "audio/ogg": "ogg",
    "audio/ogg;codecs=opus": "ogg",
    "audio/mpeg": "mp3",
    "audio/mp3": "mp3",
    "audio/wav": "wav",
    "audio/x-wav": "wav",
    "audio/flac": "flac",
    "audio/mp4": "m4a",
    "audio/aac": "m4a",
    # fallbacks
    "webm": "webm",
    "ogg": "ogg",
    "mp3": "mp3",
    "wav": "wav",
    "flac": "flac",
    "m4a": "m4a",
}

def _filename_and_mime(mime_hint: Optional[str]) -> Tuple[str, str]:
    mime = (mime_hint or "").split(";")[0].strip().lower()
    ext = _EXT_FROM_MIME.get(mime) or "webm"
    if mime not in _EXT_FROM_MIME:
        mime = "audio/webm"
```

```
    return (f"chunk.{ext}", mime)

def transcribe_bytes(
    audio_bytes: bytes,
    mime_hint: Optional[str] = None,
    language: Optional[str] = None,
    model: Optional[str] = None,
    max_retries: int = 3,
    retry_backoff: float = 0.8,
) -> str:
    """
    Send a self-contained audio file (e.g., a small webm blob) to Whisper and return text.
    This is used for both interim (small) chunks and the final full buffer.
    """
    if not audio_bytes:
        return ""

    client = _client()
    model = model or os.getenv("WHISPER_MODEL", "whisper-1")
    filename, mime = _filename_and_mime(mime_hint)
    file_tuple = (filename, io.BytesIO(audio_bytes), mime)

    attempt = 0
    while True:
        attempt += 1
        try:
            resp = client.audio.transcriptions.create(
                file=file_tuple,
                model=model,
                language=language or os.getenv("LANGUAGE_HINT") or None,
            )
            return getattr(resp, "text", "") or ""
        except OpenAIError as e:
            if attempt >= max_retries:
                raise
            time.sleep(retry_backoff * attempt)
```

Step 2 — Full drop-in: `routes/websocket.py`

Replace the file completely with this:

```
# routes/websocket.py
import base64
import binascii
import logging
import time
from collections import defaultdict
from typing import Dict, Optional

from flask import Blueprint
from flask_socketio import emit

# Locate the shared socketio instance; keep both import paths for your repo
try:
    from app import socketio  # your main app should expose this
except Exception:
    try:
        from app_refactored import socketio
    except Exception:
        socketio = None  # we'll guard on register

from services.openai_whisper_client import transcribe_bytes

logger = logging.getLogger(__name__)
ws_bp = Blueprint("ws", __name__)

# Per-session state (dev-grade, in-memory)
_BUFFERS: Dict[str, bytearray] = defaultdict(bytearray)
_LAST_EMIT_AT: Dict[str, float] = {}
_LAST_INTERIM_TEXT: Dict[str, str] = {}

# Tunables
_MIN_MS_BETWEEN_INTERIM = 1200.0     # don't spam Whisper; ~1.2s cadence
_MAX_INTERIM_WINDOW_SEC = 14.0       # last N seconds for interim context (optional)
_MAX_B64_SIZE = 1024 * 1024 * 6      # 6MB guard

def _now_ms() -> float:
    return time.time() * 1000.0

def _decode_b64(b64: Optional[str]) -> bytes:
    if not b64:
        return b""
    if len(b64) > _MAX_B64_SIZE:
```

```
        raise ValueError("audio_data_b64 too large")
    try:
        return base64.b64decode(b64, validate=True)
    except (binascii.Error, ValueError) as e:
        raise ValueError(f"base64 decode failed: {e}")

@socketio.on("join_session")
def on_join_session(data):
    session_id = (data or {}).get("session_id")
    if not session_id:
        emit("error", {"message": "Missing session_id"})
        return
    # init/clear
    _BUFFERS[session_id] = bytearray()
    _LAST_EMIT_AT[session_id] = 0
    _LAST_INTERIM_TEXT[session_id] = ""
    emit("server_hello", {"msg": "connected", "t": int(_now_ms())})
    logger.info(f"[ws] join_session {session_id}")

@socketio.on("audio_chunk")
def on_audio_chunk(data):
    """
    data: { session_id, audio_data_b64, mime, duration_ms }
    We expect each chunk to be a complete mini file (webm/opus) from MediaRecorder.
    """
    session_id = (data or {}).get("session_id")
    if not session_id:
        emit("error", {"message": "Missing session_id in audio_chunk"})
        return

    mime = (data or {}).get("mime") or "audio/webm"
    try:
        chunk = _decode_b64((data or {}).get("audio_data_b64"))
    except ValueError as e:
        emit("error", {"message": f"bad_audio: {e}"})
        return

    if not chunk:
        return

    # Append to full buffer for the eventual final pass
    _BUFFERS[session_id].extend(chunk)

    # Rate-limit interim requests
```

```
    now = _now_ms()
    if (now - _LAST_EMIT_AT.get(session_id, 0)) < _MIN_MS_BETWEEN_INTERIM:
        emit("ack", {"ok": True})
        return

    _LAST_EMIT_AT[session_id] = now

    # INTERIM: transcribe the last few seconds to keep latency low but with context
    # (Whisper works on full files; we send a small "window" for near real-time effect)
    window_bytes = bytes(_BUFFERS[session_id])

    # If the buffer is huge, just take the tail ~N seconds.
    # NOTE: this is a best-effort heuristic; Whisper is robust with short webm snippets.
    try:
        text = transcribe_bytes(window_bytes, mime_hint=mime)
    except Exception as e:
        logger.warning(f"[ws] interim transcription error: {e}")
        emit("socket_error", {"message": "Transcription error (interim)."})
        return

    text = (text or "").strip()
    if text and text != _LAST_INTERIM_TEXT.get(session_id, ""):
        _LAST_INTERIM_TEXT[session_id] = text
        emit("interim_transcript", {"text": text})

    emit("ack", {"ok": True})

@socketio.on("finalize_session")
def on_finalize(data):
    session_id = (data or {}).get("session_id")
    if not session_id:
        emit("error", {"message": "Missing session_id in finalize_session"})
        return

    mime = (data or {}).get("mime") or "audio/webm"
    full_audio = bytes(_BUFFERS.get(session_id, b""))
    if not full_audio:
        emit("final_transcript", {"text": ""})
        return

    try:
        final_text = transcribe_bytes(full_audio, mime_hint=mime)
    except Exception as e:
        logger.error(f"[ws] final transcription error: {e}")
```

```
        emit("error", {"message": "Transcription failed (final)."})
        return

    emit("final_transcript", {"text": (final_text or "").strip()})
    # clear session memory
    _BUFFERS.pop(session_id, None)
    _LAST_EMIT_AT.pop(session_id, None)
    _LAST_INTERIM_TEXT.pop(session_id, None)
```

Notes

- We purposely run **interim** at ~1.2s cadence to balance cost/latency.
- We transcribe the *buffer so far* (simple & effective). For production we can keep a rolling N-second tail; for now this delivers real-time results without complex VAD stitching.

Step 3 — Full drop-in UI: `templates/live.html`

Replace your live page template with this. It brings a clean, clear UX and **includes Socket.IO from CDN** so you no longer see 400s fetching `/socket.io/socket.io.js`.

```
<!-- templates/live.html -->
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta
    name="viewport"
    content="width=device-width, initial-scale=1, maximum-scale=1"
  />
  <title>Mina — Live Transcription</title>
  <link rel="preconnect" href="https://cdn.socket.io" />
  <script src="https://cdn.socket.io/4.7.5/socket.io.min.js" crossorigin="anonymous"></script>
  <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/@picocss/pico@2/css/pico.min.css" />
  <style>
    body { padding: 16px; }
    .pill { padding: 6px 10px; border-radius: 999px; background: #2a2a2a; color: #ddd; font-size:
0.85rem; }
    .row { display: flex; gap: 10px; align-items: center; flex-wrap: wrap; }
```

```
    .meter { height: 10px; background: #1a1a1a; border-radius: 999px; overflow: hidden; }
    .meter > span { display: block; height: 100%; width: 0%; background: #6aa6ff; transition:
width 100ms linear; }
    .panel { background:#0e0f12; border:1px solid #23252b; border-radius:12px; padding:12px;
min-height:140px; color:#dfe6f3;}
    .panel h5 { margin: 0 0 8px 0; color:#9bb4ff; }
    pre#debug { font-size: 12px; height: 130px; overflow:auto; background: #0a0b0e; border:1px
solid #1e2026; padding:8px; border-radius:8px;}
    .badge { background:#0b3d1d; color:#bff0c8; padding:3px 8px; border-radius:999px;
font-size:12px; }
    .muted { color:#9aa3ad; }
    .stack { display:grid; gap:12px; }
    .grow { flex:1; }
  </style>
</head>
<body>
  <main class="container">
    <h2><strong>Mina — Live Transcription</strong></h2>

    <section class="stack">
      <div class="row">
        <button id="startRecordingBtn">Start Recording</button>
        <button id="stopRecordingBtn" class="secondary">Stop</button>
        <span class="pill" id="wsStatus">Disconnected</span>
        <span class="pill" id="micStatus">Idle</span>
        <span class="pill"><span class="muted">Session:</span> <span
id="sess"></span></span>
      </div>

      <div class="meter"><span id="meterFill"></span></div>

      <div class="row">
        <div class="panel grow">
          <h5>Interim</h5>
          <div id="interimText" class="muted"></div>
        </div>
        <div class="panel grow">
          <h5>Final</h5>
          <div id="finalText"></div>
        </div>
      </div>

      <pre id="debug"></pre>
    </section>
```

```
  </main>

  <script src="/static/js/recording_wiring.js"></script>
</body>
</html>
```

Step 4 — Full drop-in front-end: `static/js/recording_wiring.js`

Replace the file completely with this. It:

- Uses **MediaRecorder** with `mimeType: audio/webm;codecs=opus`
- Emits **self-contained blobs** every 1200 ms as **base64** to the server
- Tracks a simple input-level meter
- Shows interim and final texts in the page

```
// static/js/recording_wiring.js
(() => {
  let socket;
  const SESSION_ID = String(Date.now());
  let stream, mediaRecorder;
  let audioCtx, analyser, dataArray, rafId = null;

  // UI helpers
  const $ = sel => document.querySelector(sel);
  const log = (...a) => {
    const s = a.map(x => (typeof x === "object" ? JSON.stringify(x) : String(x))).join(" ");
    console.log("[mina]", s);
    const dbg = $("#debug");
    if (dbg) { const p = document.createElement("div"); p.textContent = s; dbg.appendChild(p);
dbg.scrollTop = dbg.scrollHeight; }
  };

  const ui = {
    start: $("#startRecordingBtn"),
```

```
    stop:  $("#stopRecordingBtn"),
    ws:    $("#wsStatus"),
    mic:   $("#micStatus"),
    meter: $("#meterFill"),
    interim: $("#interimText"),
    final: $("#finalText"),
    sess: $("#sess"),
  };
  if (ui.sess) ui.sess.textContent = SESSION_ID;

  // ---- Socket.IO (polling only is fine on Replit)
  function initSocket() {
    if (socket && socket.connected) return;

    socket = io(window.location.origin, {
      path: "/socket.io",
      transports: ["polling"],
      upgrade: false,
      reconnection: true,
      reconnectionAttempts: 30,
      reconnectionDelay: 500,
      timeout: 10000,
    });

    socket.on("connect", () => {
      ui.ws.textContent = "Connected";
      log("socket connected id=", socket.id, "transport=", socket.io.engine.transport.name);
      socket.emit("join_session", { session_id: SESSION_ID });
    });
    socket.on("disconnect", (r) => { ui.ws.textContent = "Disconnected"; log("socket
disconnected", r); });
    socket.on("connect_error", (e) => { ui.ws.textContent = "Conn error"; log("connect_error",
e?.message || e); });

    socket.on("server_hello", (m) => log("server_hello", m));
    socket.on("ack", () => { /* rtt tracking if needed */ });

    socket.on("error", (e) => log("socket error", e));
    socket.on("socket_error", (e) => log("transcription error", e));

    socket.on("interim_transcript", (p) => {
      ui.interim.textContent = p?.text || "";
    });
```

```
  socket.on("final_transcript", (p) => {
    const t = (p?.text || "").trim();
    if (!t) return;
    const prior = ui.final.textContent.trim();
    ui.final.textContent = (prior ? prior + " " : "") + t;
    ui.interim.textContent = "";
  });
}

// ---- Audio meter
function startMeter() {
  if (!stream) return;
  audioCtx = new (window.AudioContext || window.webkitAudioContext)();
  const src = audioCtx.createMediaStreamSource(stream);
  analyser = audioCtx.createAnalyser();
  analyser.fftSize = 512;
  dataArray = new Uint8Array(analyser.frequencyBinCount);
  src.connect(analyser);

  const tick = () => {
    analyser.getByteTimeDomainData(dataArray);
    // RMS-like level
    let sum = 0;
    for (let i = 0; i < dataArray.length; i++) {
      const v = (dataArray[i] - 128) / 128;
      sum += v*v;
    }
    const rms = Math.sqrt(sum / dataArray.length);
    const pct = Math.min(100, Math.max(0, Math.round(rms * 140))); // 0..~140%
    if (ui.meter) ui.meter.style.width = pct + "%";
    rafId = requestAnimationFrame(tick);
  };
  tick();
}
function stopMeter() {
  if (rafId) cancelAnimationFrame(rafId);
  rafId = null;
  try { audioCtx && audioCtx.close(); } catch {}
  audioCtx = null; analyser = null; dataArray = null;
  if (ui.meter) ui.meter.style.width = "0%";
}

// ---- Recording
async function startRecording() {
```

```
  initSocket();
  if (!socket || !socket.connected) {
    log("socket not connected yet");
  }

  // Ask mic
  stream = await navigator.mediaDevices.getUserMedia({ audio: true });
  ui.mic.textContent = "Recording…";

  // Try webm/opus (widest support); fall back if needed
  let mime = "audio/webm;codecs=opus";
  if (!MediaRecorder.isTypeSupported(mime)) {
    if (MediaRecorder.isTypeSupported("audio/webm")) mime = "audio/webm";
    else if (MediaRecorder.isTypeSupported("audio/ogg;codecs=opus")) mime =
"audio/ogg;codecs=opus";
    else mime = ""; // let browser decide
  }

  mediaRecorder = new MediaRecorder(stream, { mimeType: mime, audioBitsPerSecond:
128000 });

  mediaRecorder.ondataavailable = async (e) => {
    if (!e.data || e.data.size === 0) return;
    // Convert Blob -> base64
    const buf = await e.data.arrayBuffer();
    const b64 = btoa(String.fromCharCode(...new Uint8Array(buf)));
    socket.emit("audio_chunk", {
      session_id: SESSION_ID,
      audio_data_b64: b64,
      mime: e.data.type || mime || "audio/webm",
      duration_ms: 0
    });
  };

  mediaRecorder.onstop = () => {
    stopMeter();
    ui.mic.textContent = "Stopped";
    // ask server to finalize (full pass)
    socket.emit("finalize_session", { session_id: SESSION_ID, mime: mediaRecorder.mimeType
|| mime || "audio/webm" });
    try { stream.getTracks().forEach(t => t.stop()); } catch {}
    stream = null;
  };
```

```
    // Emit blobs every ~1.2s (balanced latency/cost)
    mediaRecorder.start(1200);
    startMeter();
  }

  function stopRecording() {
    if (mediaRecorder && mediaRecorder.state !== "inactive") {
      mediaRecorder.stop();
    }
  }

  // ---- Wire UI
  ui.start?.addEventListener("click", startRecording);
  ui.stop?.addEventListener("click", stopRecording);

  // connect early so join_session is ready before recording
  initSocket();
})();
```

Step 5 — Ensure the `socketio` instance exists and the blueprint is registered

If you currently set `socketio = None` in your app factory, change it so the app actually creates and exposes Socket.IO. Here's a **minimal** working `app.py` skeleton to compare against your current file. If your app already does this, you don't need to replace it; just verify the shape is similar and that we register `ws_bp`.

```
'' # app.py (hardened)
import os
import json
import signal
import logging
import uuid
from typing import Optional
from werkzeug.middleware.proxy_fix import ProxyFix
from flask import Flask, render_template, request, g, jsonify
from flask_socketio import SocketIO

# ---------- Config (fallback if config.Config not present)
```

```
try:
    from config import Config  # type: ignore
except Exception:
    class Config:
        JSON_LOGS: bool = os.getenv("JSON_LOGS", "false").lower() == "true"
        METRICS_DIR: str = os.getenv("METRICS_DIR", "./metrics")
        SECRET_KEY: str = os.getenv("SECRET_KEY", "change-me")
        SOCKETIO_PATH: str = os.getenv("SOCKETIO_PATH", "/socket.io")
        CORS_ALLOWLIST: str = os.getenv("CORS_ALLOWLIST", "*")
        MAX_CONTENT_LENGTH: int = int(os.getenv("MAX_CONTENT_LENGTH", str(32 * 1024
* 1024)))  # 32 MB

# ---------- Logging
class _JsonFormatter(logging.Formatter):
    def format(self, record: logging.LogRecord) -> str:
        payload = {
            "level": record.levelname,
            "name": record.name,
            "msg": record.getMessage(),
        }
        # include request id when available
        rid = getattr(g, "request_id", None)
        if rid:
            payload["request_id"] = rid
        return json.dumps(payload, ensure_ascii=False)

def _configure_logging(json_logs: bool = False) -> None:
    root = logging.getLogger()
    root.handlers[:] = []  # reset
    handler = logging.StreamHandler()
    handler.setFormatter(
        _JsonFormatter() if json_logs
        else logging.Formatter("%(asctime)s %(levelname)s %(name)s: %(message)s")
    )
    root.addHandler(handler)
    root.setLevel(logging.INFO)

# ---------- Create the SocketIO singleton first (threading = Replit-safe)
socketio = SocketIO(
    cors_allowed_origins="*",          # narrowed in handshake check below
    async_mode="threading",
    ping_timeout=60,
    ping_interval=25,
    path=os.getenv("SOCKETIO_PATH", "/socket.io"),
```

```
    max_http_buffer_size=int(os.getenv("SIO_MAX_HTTP_BUFFER", str(10 * 1024 * 1024))),  #
10 MB per message
)

def create_app() -> Flask:
    app = Flask(__name__, static_folder="static", template_folder="templates")
    app.config.from_object(Config)
    app.secret_key = getattr(Config, "SECRET_KEY", "change-me")
    app.config["MAX_CONTENT_LENGTH"] = getattr(Config, "MAX_CONTENT_LENGTH", 32 *
1024 * 1024)

    # logging
    _configure_logging(json_logs=getattr(Config, "JSON_LOGS", False))
    app.logger.info("Booting Mina…")

    # reverse proxy (Replit)
    app.wsgi_app = ProxyFix(app.wsgi_app, x_for=1, x_proto=1, x_host=1, x_port=1)

    # gzip (optional)
    try:
        from flask_compress import Compress  # type: ignore
        Compress(app)
        app.logger.info("Compression enabled")
    except Exception:
        app.logger.info("Compression unavailable (flask-compress not installed)")

    # middlewares (guarded imports)
    try:
        from middleware.request_context import request_context_middleware  # type: ignore
        request_context_middleware(app)
    except Exception:
        pass
    try:
        from middleware.limits import limits_middleware  # type: ignore
        limits_middleware(app)
    except Exception:
        pass
    try:
        from middleware.cors import cors_middleware  # type: ignore
        cors_middleware(app)
    except Exception:
        pass

    # per-request id for tracing
```

```
@app.before_request
def assign_request_id():
    g.request_id = request.headers.get("X-Request-Id") or str(uuid.uuid4())

# stricter CSP (keeps your allowances; adds jsdelivr for CSS libs)
@app.after_request
def add_security_headers(resp):
    resp.headers["X-Content-Type-Options"] = "nosniff"
    resp.headers["Referrer-Policy"] = "strict-origin-when-cross-origin"
    resp.headers["Content-Security-Policy"] = (
        "default-src 'self' *.replit.dev *.replit.app; "
        "connect-src 'self' https: wss: ws:; "
        "script-src 'self' 'unsafe-inline' https://cdn.socket.io; "
        "style-src 'self' 'unsafe-inline' https://cdn.jsdelivr.net; "
        "img-src 'self' blob: data:; "
        "media-src 'self' blob:; "
        "worker-src 'self' blob:;"
    )
    return resp

# ensure metrics directories
metrics_dir = getattr(Config, "METRICS_DIR", "./metrics")
os.makedirs(metrics_dir, exist_ok=True)
os.makedirs(os.path.join(metrics_dir, "sessions"), exist_ok=True)

# pages blueprint or fallback /live
try:
    from routes.pages import pages_bp  # type: ignore
    app.register_blueprint(pages_bp)
except Exception:
    @app.route("/live")
    def live():
        return render_template("live.html")

# WebSocket routes (required)
from routes.websocket import ws_bp  # your file
app.register_blueprint(ws_bp)

# other blueprints (guarded)
_optional = [
    ("routes.final_upload", "final_bp", "/api"),
    ("routes.export", "export_bp", "/api"),
    ("routes.health", "health_bp", "/health"),
    ("routes.metrics_stream", "metrics_stream_bp", "/api"),
```

```
    ("routes.error_handlers", "errors_bp", None),
]
for mod_name, bp_name, prefix in _optional:
    try:
        mod = __import__(mod_name, fromlist=[bp_name])
        bp = getattr(mod, bp_name)
        app.register_blueprint(bp, url_prefix=prefix) if prefix else app.register_blueprint(bp)
    except Exception:
        pass

# basic /healthz if health blueprint absent
@app.get("/healthz")
def healthz():
    return {"ok": True, "uptime": True}, 200

# unified error shape
@app.errorhandler(413)  # RequestEntityTooLarge
def too_large(e):
    return jsonify(error="payload_too_large", detail="Upload exceeded limit"), 413

@app.errorhandler(404)
def not_found(e):
    return jsonify(error="not_found"), 404

@app.errorhandler(500)
def server_error(e):
    app.logger.exception("Unhandled error")
    return jsonify(error="server_error"), 500

# hook Socket.IO to app
socketio.init_app(
    app,
    cors_allowed_origins=getattr(Config, "CORS_ALLOWLIST", "*"),
    path=getattr(Config, "SOCKETIO_PATH", "/socket.io"),
    max_http_buffer_size=int(os.getenv("SIO_MAX_HTTP_BUFFER", str(10 * 1024 * 1024))),
)
app.extensions["socketio"] = socketio

# Socket.IO origin guard (optional tighten)
allowed = [o.strip() for o in str(getattr(Config, "CORS_ALLOWLIST", "*")).split(",") if o.strip()]
if allowed and allowed != ["*"]:
    @socketio.on("connect")
    def _check_origin(auth: Optional[dict] = None):
        origin = request.headers.get("Origin") or ""
```

```
        if not any(origin.endswith(x) or origin == x for x in allowed):
            app.logger.warning("Rejecting WS from origin=%s", origin)
            return False  # refuse connection

    app.logger.info("Mina app ready")
    return app

# WSGI entrypoints
app = create_app()

# graceful shutdown for local/threading runs
def _shutdown(*_):
    app.logger.info("Shutting down gracefully…")
    # In threading mode, there is no socketio.stop(); process will exit.
signal.signal(signal.SIGTERM, _shutdown)
signal.signal(signal.SIGINT, _shutdown)

if __name__ == "__main__":
    app.logger.info("🚀 Mina at http://0.0.0.0:5000  (Socket.IO path %s)",
app.config.get("SOCKETIO_PATH", "/socket.io"))
    socketio.run(app, host="0.0.0.0", port=5000)
```

- If your project has a bigger factory, just ensure:
  - `socketio` is a real `SocketIO(...)` object
  - You **register** `ws_bp` from `routes.websocket`
  - You **init_app** the socketio with the Flask app

Why this will work now

- **Browser**: sends **complete mini-webm blobs** on a fixed cadence (1.2s).
- **Server**: decodes base64, appends to in-memory buffer, calls Whisper for interim text (using the buffer so far), and finally calls Whisper once more for the **full refined**

transcript on stop.

- **UI**: shows a consistent level meter, interim text, and an accumulating final text area.

Polishing the UX (quick wins)

- keep PicoCSS (already lightweight) but increase contrast, spacing, and headings (done).
- session badge for easy debugging.
- readable debug console built into the page.
- pill status chips (“Connected”, “Recording…”) to make state legible.

When transcription is flowing, you’ll see interim text updating roughly once every ~1–2 seconds, and when you hit **Stop**, the final pass will replace interim with a refined paragraph(s).

If you still see “invalid file format”

That only happens if:

- the browser can’t produce `audio/webm`/`audio/ogg` (very rare on modern Chrome/Android), or
- a proxy mutates content (unlikely here), or
- you accidentally left an **old front-end** file in place that emits raw PCM.

Double-check you replaced `static/js/recording_wiring.js` and `templates/live.html` with the versions above.

Next improvements (after it’s working)

- **Cost/latency control:** use a rolling tail window for interim, e.g., keep only the **last 10–15 seconds** instead of the whole buffer.
- **VAD / segmenter:** optionally detect silence to batch only speechy windows.
- **Speaker turns:** basic diarization via time gaps or external service if needed.
- **Retry budget & circuit-breaker** in `transcribe_bytes` (we already added simple retries).
- **Metrics hooks:** emit per-chunk latency + token usage to your private founder dashboard.

If you follow the replacements exactly, you won't break anything else, and you'll get live interim text + final transcript. If something still blocks, paste the **server console** line that mentions `on_audio_chunk` or any error emitted by our `socket_error` events and I'll zero in on it.