Community Radio App - Implementation Manual v2.0 (Final)

📋 Table of Contents

---

1. Project Setup

1.1 Create Your Vercel Project

```
# Create Next.js project
npx create-next-app@latest community-radio
cd community-radio

# Install required dependencies
npm install agora-rtc-react@^2.4.0 \
  @agoraio-extensions/agora-rtm-react@^2.2.0 \
  @supabase/supabase-js@^2.49.0 \
  agora-token@^2.0.3 \
  zod@^3.23.0

# Install dev dependencies
npm install -D @types/node@^20.11.0 \
  @types/react@^18.3.0 \
  typescript@^5.3.0
```

1.2 Get Your Agora Credentials

1. Go to Agora Console
2. Create a new project
3. Copy your **App ID** and **App Certificate**
4. Enable **Cloud Recording** in your project settings
5. Generate **Customer ID** and **Customer Certificate** for recording API access

6. Set up a **Webhook** endpoint URL for recording callbacks

1.3 Set Up Environment Variables

Create a `.env.local` file:

```
# ==========================================
# AGORA CONFIGURATION
# ==========================================
NEXT_PUBLIC_AGORA_APP_ID=your_app_id_here
AGORA_APP_CERTIFICATE=your_app_certificate_here
AGORA_CUSTOMER_ID=your_customer_id_here
AGORA_CUSTOMER_CERTIFICATE=your_customer_certificate_here
AGORA_WEBHOOK_SECRET=your_webhook_secret_here

# ==========================================
# SUPABASE CONFIGURATION
# ==========================================
NEXT_PUBLIC_SUPABASE_URL=your_supabase_url_here
NEXT_PUBLIC_SUPABASE_ANON_KEY=your_supabase_anon_key_here
SUPABASE_SERVICE_ROLE_KEY=your_service_role_key_here

# ==========================================
# CLOUD STORAGE (AWS S3)
# ==========================================
AWS_ACCESS_KEY_ID=your_access_key_here
AWS_SECRET_ACCESS_KEY=your_secret_key_here
AWS_BUCKET_NAME=your_bucket_name_here
AWS_REGION=your_region_here

# ==========================================
# AUTO-DJ BOT (External Service)
# ==========================================
AUTO_DJ_BOT_URL=https://your-bot.railway.app
AUTO_DJ_BOT_SECRET=shared_secret_between_vercel_and_bot
NEXT_PUBLIC_ICECAST_URL=https://your-icecast-server/stream.mp3

# ==========================================
# APPLICATION
# ==========================================
NEXT_PUBLIC_APP_URL=https://your-app.vercel.app
SCHEDULE_TIMEZONE=UTC
```

---

2. Live Audio Broadcasting

2.1 Create the Radio Component

Create `components/RadioBroadcast.tsx`:

```
import { useEffect, useState } from 'react';
import AgoraRTC, {
  AgoraRTCProvider,
  useJoin,
  useLocalMicrophoneTrack,
  usePublish,
  useRemoteUsers,
  useRemoteAudioTrack
} from 'agora-rtc-react';
```

```
// Create Agora client with live streaming mode
const client = AgoraRTC.createClient({
  mode: 'live',
  codec: 'vp8'
});

interface RadioBroadcastProps {
  appId: string;
  channelName: string;
  token: string;
  uid: number;
  role: 'host' | 'audience';
}

function RadioBroadcast({ appId, channelName, token, uid, role }:
RadioBroadcastProps) {
  const [isMuted, setIsMuted] = useState(false);
  const [micVolume, setMicVolume] = useState(100);
  const [playbackVolume, setPlaybackVolume] = useState(100);

  // Get local microphone track
  const { localMicrophoneTrack } = useLocalMicrophoneTrack(true);

  // Join the channel with role-based permissions
  const { isConnected } = useJoin({
    appId: appId,
    channel: channelName,
    token: token,
    uid: uid,
    options: {
      clientRole: role === 'host' ? 'host' : 'audience'
    }
  }, true);

  // Publish audio only if host
  usePublish(role === 'host' ? [localMicrophoneTrack] : []);

  // Get remote users (only hosts/co-hosts appear here, not audience)
  const remoteUsers = useRemoteUsers();

  // Track listener count via backend counter
  const [listenerCount, setListenerCount] = useState(0);

  useEffect(() => {
    // Subscribe to listener count updates from Supabase Realtime
    const subscription = supabase
      .channel('channels')
      .on('postgres_changes',
        { event: 'UPDATE', schema: 'public', table: 'channels', filter:
`slug=eq.${channelName}` },
        (payload) => {
          setListenerCount(payload.new.listener_count || 0);
        }
      )
      .subscribe();

    return () => subscription.unsubscribe();
  }, [channelName]);

  // Volume control functions using correct SDK methods
  const handleMicVolumeChange = (volume: number) => {
    setMicVolume(volume);
    // Volume range: 0 (mute) to 255 (max)
    const mappedVolume = Math.round(volume * 2.55);
```

```
    localMicrophoneTrack?.setVolume(mappedVolume);
  };

  const toggleMute = () => {
    setIsMuted(!isMuted);
    localMicrophoneTrack?.setEnabled(!isMuted);
  };

  const handlePlaybackChange = (volume: number) => {
    setPlaybackVolume(volume);
    // Adjust remote user audio volume
    const mappedVolume = Math.round(volume * 2.55);
    client.setPlaybackVolume(mappedVolume);
  };

  return (
    <div className="radio-broadcast">
      <div className="status">
        <span className={isConnected ? 'live' : 'offline'}>
          {isConnected ? '🔴 Live' : '⚫ Offline'}
        </span>
        <span className="listeners">👥 {listenerCount} listeners</span>
      </div>

      {role === 'host' && (
        <div className="host-controls">
          <div className="mic-controls">
            <label>Mic Volume</label>
            <input
              type="range"
              min="0"
              max="100"
              value={micVolume}
              onChange={(e) => handleMicVolumeChange(Number(e.target.value))}
            />
            <button onClick={toggleMute}>
              {isMuted ? '🔇 Unmute' : '🎤 Mute'}
            </button>
          </div>
        </div>
      )}

      <div className="listener-controls">
        <label>Speaker Volume</label>
        <input
          type="range"
          min="0"
          max="100"
          value={playbackVolume}
          onChange={(e) => handlePlaybackChange(Number(e.target.value))}
        />
      </div>

      {/* Remote users (hosts/co-hosts) */}
      <div className="remote-users">
        {remoteUsers.map((user) => (
          <RemoteUserTrack key={user.uid} user={user} />
        ))}
      </div>
    </div>
  );
}

// Component for rendering remote user audio
```

```
function RemoteUserTrack({ user }: { user: any }) {
  const audioTrack = useRemoteAudioTrack(user);

  useEffect(() => {
    if (audioTrack) {
      audioTrack.play();
    }
    return () => {
      audioTrack?.stop();
    };
  }, [audioTrack]);

  return <div>User: {user.uid}</div>;
}

export default RadioBroadcast;
```

2.2 Generate Secure Tokens (Server-Side)

Create `app/api/token/route.ts`:

```
import { NextRequest, NextResponse } from 'next/server';
import { RtcTokenBuilder, RtcRole } from 'agora-token';

export async function POST(req: NextRequest) {
  try {
    const { channelName, role = 'audience' } = await req.json();

    if (!channelName) {
      return NextResponse.json({ error: 'Missing channelName' }, { status:
400 });
    }

    const appId = process.env.NEXT_PUBLIC_AGORA_APP_ID!;
    const appCertificate = process.env.AGORA_APP_CERTIFICATE!;
    const uid = Math.floor(Math.random() * 1000000);

    const expirationTimeInSeconds = 3600;
    const currentTimestamp = Math.floor(Date.now() / 1000);
    const privilegeExpiredTs = currentTimestamp + expirationTimeInSeconds;

    // Set role based on request
    const rtcRole = role === 'host' ? RtcRole.PUBLISHER : RtcRole.SUBSCRIBER;

    const token = RtcTokenBuilder.buildTokenWithUid(
      appId,
      appCertificate,
      channelName,
      uid,
      rtcRole,
      privilegeExpiredTs
    );

    return NextResponse.json({
      token,
      uid,
      expiresAt: privilegeExpiredTs
    });

  } catch (error) {
    console.error('Token generation error:', error);
    return NextResponse.json({ error: 'Failed to generate token' }, { status:
500 });
```

```
  }
}
```

2.3 Use the Radio Component

Create `app/page.tsx`:

```
'use client';

import { useState, useEffect } from 'react';
import RadioBroadcast from '@/components/RadioBroadcast';

export default function Home() {
  const [channelName, setChannelName] = useState('main-radio');
  const [token, setToken] = useState('');
  const [uid, setUid] = useState(0);
  const [role, setRole] = useState<'host' | 'audience'>('audience');

  useEffect(() => {
    const fetchToken = async () => {
      const response = await fetch('/api/token', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ channelName, role })
      });
      const data = await response.json();
      setToken(data.token);
      setUid(data.uid);
    };

    fetchToken();
  }, [channelName, role]);

  return (
    <main>
      <h1>Community Radio</h1>
      <div className="controls">
        <select value={role} onChange={(e) => setRole(e.target.value as 'host' |
'audience')}>
          <option value="audience">Listener</option>
          <option value="host">DJ/Host</option>
        </select>
      </div>

      {token && (
        <RadioBroadcast
          appId={process.env.NEXT_PUBLIC_AGORA_APP_ID!}
          channelName={channelName}
          token={token}
          uid={uid}
          role={role}
        />
      )}
    </main>
  );
}
```

---

3. Volume Controls & Mute

3.1 DJ Volume Controls

```
function DJVolumeControls({ localMicrophoneTrack }: { localMicrophoneTrack:
any }) {
  const [micVolume, setMicVolume] = useState(100);
  const [isMuted, setIsMuted] = useState(false);

  const handleMicVolumeChange = (volume: number) => {
    setMicVolume(volume);
    // Correct: Use setVolume on track (0-255)
    const mappedVolume = Math.round(volume * 2.55);
    localMicrophoneTrack?.setVolume(mappedVolume);
  };

  const toggleMute = () => {
    setIsMuted(!isMuted);
    // Correct: Use setEnabled on track
    localMicrophoneTrack?.setEnabled(!isMuted);
  };

  return (
    <div className="dj-controls">
      <label>🎤 Mic Volume</label>
      <input
        type="range"
        min="0"
        max="100"
        value={micVolume}
        onChange={(e) => handleMicVolumeChange(Number(e.target.value))}
      />
      <button onClick={toggleMute}>
        {isMuted ? '🔇 Unmute' : '🎙️ Mute'}
      </button>
    </div>
  );
}
```

3.2 Listener Volume Controls

```
function ListenerVolumeControls({ client }: { client: any }) {
  const [playbackVolume, setPlaybackVolume] = useState(100);

  const handlePlaybackChange = (volume: number) => {
    setPlaybackVolume(volume);
    // Volume range: 0 (mute) to 255 (max)
    const mappedVolume = Math.round(volume * 2.55);
    client.setPlaybackVolume(mappedVolume);
  };

  return (
    <div className="listener-controls">
      <label>🔊 Speaker Volume</label>
      <input
        type="range"
        min="0"
        max="100"
        value={playbackVolume}
        onChange={(e) => handlePlaybackChange(Number(e.target.value))}
      />
    </div>
```

```
  );
}
```

3.3 Independent Volume Control for DJ Monitor

```
function DJMonitorControls({ remoteUsers }: { remoteUsers: any[] }) {
  const [userVolumes, setUserVolumes] = useState<Record<string, number>>({});

  const handleUserVolumeChange = (uid: string, volume: number) => {
    setUserVolumes(prev => ({ ...prev, [uid]: volume }));
    // Adjust volume for specific user's audio track
    const mappedVolume = Math.round(volume * 2.55);
    // Apply to specific remote user track
  };

  return (
    <div className="dj-monitor">
      <h4>Caller Audio</h4>
      {remoteUsers.map((user) => (
        <div key={user.uid} className="user-volume">
          <span>User {user.uid}</span>
          <input
            type="range"
            min="0"
            max="100"
            value={userVolumes[user.uid] || 100}
            onChange={(e) => handleUserVolumeChange(user.uid,
Number(e.target.value))}
          />
        </div>
      ))}
    </div>
  );
}
```

---

4. Recording Shows

4.1 Set Up Cloud Recording with Full 3-Step Flow

Create `app/api/recording/start/route.ts`:

```
import { NextRequest, NextResponse } from 'next/server';
import { createClient } from '@supabase/supabase-js';
import { RtcTokenBuilder, RtcRole } from 'agora-token';

const AGORA_API_BASE = 'https://api.agora.io/v1/apps';
const APP_ID = process.env.NEXT_PUBLIC_AGORA_APP_ID!;
const CUSTOMER_ID = process.env.AGORA_CUSTOMER_ID!;
const CUSTOMER_CERT = process.env.AGORA_CUSTOMER_CERTIFICATE!;
const BUCKET_NAME = process.env.AWS_BUCKET_NAME!;
const REGION = process.env.AWS_REGION!;

function getBasicAuth() {
  return 'Basic ' + Buffer.from(`${CUSTOMER_ID}:${CUSTOMER_CERT}
`).toString('base64');
}

function getAgoraRegionCode(region: string): number {
  const regionMap: Record<string, number> = {
```

```
    'us-east-1': 0,
    'us-west-1': 1,
    'eu-west-1': 2,
    'ap-southeast-1': 3,
    'ap-northeast-1': 4,
    'ap-south-1': 5,
    'sa-east-1': 6,
  };
  return regionMap[region] || 0;
}

function generateRecordingToken(channelName: string, uid: number): string {
  const appId = process.env.NEXT_PUBLIC_AGORA_APP_ID!;
  const appCertificate = process.env.AGORA_APP_CERTIFICATE!;
  const expirationTimeInSeconds = 3600;
  const currentTimestamp = Math.floor(Date.now() / 1000);
  const privilegeExpiredTs = currentTimestamp + expirationTimeInSeconds;

  return RtcTokenBuilder.buildTokenWithUid(
    appId,
    appCertificate,
    channelName,
    uid,
    RtcRole.SUBSCRIBER,
    privilegeExpiredTs
  );
}

export async function POST(req: NextRequest) {
  try {
    const { channelName, uid = 0, mode = 'mix', showId } = await req.json();

    if (!channelName) {
      return NextResponse.json({ error: 'Missing channelName' }, { status:
400 });
    }

    // Step 1: ACQUIRE resource ID
    const acquireRes = await fetch(
      `${AGORA_API_BASE}/${APP_ID}/cloud_recording/acquire`,
      {
        method: 'POST',
        headers: {
          'Authorization': getBasicAuth(),
          'Content-Type': 'application/json'
        },
        body: JSON.stringify({
          cname: channelName,
          uid: uid.toString(),
          clientRequest: {
            resourceExpiredHour: 24
          }
        })
      }
    );

    if (!acquireRes.ok) {
      const error = await acquireRes.text();
      console.error('Acquire failed:', error);
      return NextResponse.json({ error: 'Failed to acquire recording
resource' }, { status: 502 });
    }

    const acquireData = await acquireRes.json();
```

```
    const resourceId = acquireData.resourceId;

    // Generate token for recording
    const token = generateRecordingToken(channelName, uid);

    // Step 2: START recording
    const startRes = await fetch(
      `${AGORA_API_BASE}/${APP_ID}/cloud_recording/resourceid/${resourceId}/
mode/${mode}/start`,
      {
        method: 'POST',
        headers: {
          'Authorization': getBasicAuth(),
          'Content-Type': 'application/json'
        },
        body: JSON.stringify({
          cname: channelName,
          uid: uid.toString(),
          clientRequest: {
            token,
            recordingConfig: {
              maxIdleTime: 30,
              streamTypes: 0, // 0 = audio only
              channelType: 1, // 1 = live broadcast
              transcodingConfig: {
                height: 640,
                width: 360,
                bitrate: 500,
                fps: 15,
                mixedVideoLayout: 1,
                backgroundColor: '#000000'
              }
            },
            recordingFileConfig: {
              avFileType: ['hls', 'mp4']
            },
            storageConfig: {
              vendor: 1, // 1 = AWS S3
              region: getAgoraRegionCode(REGION),
              bucket: BUCKET_NAME,
              accessKey: process.env.AWS_ACCESS_KEY_ID!,
              secretKey: process.env.AWS_SECRET_ACCESS_KEY!,
              fileNamePrefix: ['recordings', channelName]
            }
          }
        })
      }
    );

    if (!startRes.ok) {
      const error = await startRes.text();
      console.error('Start recording failed:', error);
      return NextResponse.json({ error: 'Failed to start recording' }, { status:
502 });
    }

    const startData = await startRes.json();
    const { sid, serverResponse } = startData;

    // Step 3: Save to database
    const supabase = createClient(
      process.env.NEXT_PUBLIC_SUPABASE_URL!,
      process.env.SUPABASE_SERVICE_ROLE_KEY!
    );
```

```
    // Create episode record
    const { data: episode, error: episodeError } = await supabase
      .from('episodes')
      .insert({
        show_id: showId || null,
        title: `Live Recording ${new Date().toLocaleString()}`,
        mode: 'live',
        recording_resource_id: resourceId,
        recording_sid: sid,
        is_published: false
      })
      .select()
      .single();

    if (episodeError) {
      console.error('Failed to save episode:', episodeError);
    } else {
      // Save recording metadata
      await supabase.from('recordings').insert({
        episode_id: episode.id,
        channel_name: channelName,
        resource_id: resourceId,
        sid,
        status: 'recording'
      });

      // Update channel recording status
      await supabase
        .from('channels')
        .update({
          is_recording: true,
          recording_resource_id: resourceId,
          recording_sid: sid
        })
        .eq('slug', channelName);
    }

    return NextResponse.json({
      resourceId,
      sid,
      serverResponse,
      episodeId: episode?.id
    });

  } catch (error) {
    console.error('Recording start error:', error);
    return NextResponse.json({ error: 'Internal server error' }, { status:
500 });
  }
}
```

4.2 Stop Recording

Create `app/api/recording/stop/route.ts`:

```
import { NextRequest, NextResponse } from 'next/server';
import { createClient } from '@supabase/supabase-js';

const AGORA_API_BASE = 'https://api.agora.io/v1/apps';
const APP_ID = process.env.NEXT_PUBLIC_AGORA_APP_ID!;
const CUSTOMER_ID = process.env.AGORA_CUSTOMER_ID!;
const CUSTOMER_CERT = process.env.AGORA_CUSTOMER_CERTIFICATE!;
```

```
function getBasicAuth() {
  return 'Basic ' + Buffer.from(`${CUSTOMER_ID}:${CUSTOMER_CERT}
`).toString('base64');
}

export async function POST(req: NextRequest) {
  try {
    const { channelName, resourceId, sid, mode = 'mix', uid = 0 } = await
req.json();

    if (!channelName || !resourceId || !sid) {
      return NextResponse.json({ error: 'Missing required fields' }, { status:
400 });
    }

    // Step 1: STOP recording
    const stopRes = await fetch(
      `${AGORA_API_BASE}/${APP_ID}/cloud_recording/resourceid/${resourceId}/
sid/${sid}/mode/${mode}/stop`,
      {
        method: 'POST',
        headers: {
          'Authorization': getBasicAuth(),
          'Content-Type': 'application/json'
        },
        body: JSON.stringify({
          cname: channelName,
          uid: uid.toString(),
          clientRequest: {}
        })
      }
    );

    if (!stopRes.ok) {
      const error = await stopRes.text();
      console.error('Stop recording failed:', error);
      return NextResponse.json({ error: 'Failed to stop recording' }, { status:
502 });
    }

    const data = await stopRes.json();

    // Step 2: Update database
    const supabase = createClient(
      process.env.NEXT_PUBLIC_SUPABASE_URL!,
      process.env.SUPABASE_SERVICE_ROLE_KEY!
    );

    await supabase
      .from('recordings')
      .update({
        status: 'stopped',
        stopped_at: new Date().toISOString(),
        file_list: data.serverResponse?.fileList
      })
      .eq('sid', sid);

    // Step 3: Update channel recording status
    await supabase
      .from('channels')
      .update({
        is_recording: false,
        recording_resource_id: null,
```

```
        recording_sid: null
      })
      .eq('slug', channelName);

    return NextResponse.json(data);

  } catch (error) {
    console.error('Recording stop error:', error);
    return NextResponse.json({ error: 'Internal server error' }, { status:
500 });
  }
}
```

4.3 Webhook Receiver with Signature Verification

Create `app/api/webhooks/agora/route.ts`:

```
import { NextRequest, NextResponse } from 'next/server';
import { createHmac } from 'crypto';
import { createClient } from '@supabase/supabase-js';

const WEBHOOK_SECRET = process.env.AGORA_WEBHOOK_SECRET!;

function verifySignature(body: string, signature: string): boolean {
  if (!signature) return false;
  const expected = createHmac('sha256', WEBHOOK_SECRET)
    .update(body)
    .digest('hex');
  return signature === expected;
}

export async function POST(req: NextRequest) {
  try {
    // Verify webhook signature
    const signature = req.headers.get('x-agora-signature') || '';
    const body = await req.text();

    if (!verifySignature(body, signature)) {
      console.error('Invalid webhook signature');
      return NextResponse.json({ error: 'Invalid signature' }, { status: 401 });
    }

    const event = JSON.parse(body);
    const supabase = createClient(
      process.env.NEXT_PUBLIC_SUPABASE_URL!,
      process.env.SUPABASE_SERVICE_ROLE_KEY!
    );

    switch (event.eventType) {
      case 31: // Recording file upload done
        await handleRecordingComplete(event, supabase);
        break;
      case 40: // Recorder error
        await handleRecordingError(event, supabase);
        break;
      case 32: // Recording backup done
        console.log('Recording backup completed:', event.sid);
        break;
      default:
        console.log(`Unhandled event type: ${event.eventType}`);
    }

    return NextResponse.json({ received: true });
```

```
  } catch (error) {
    console.error('Webhook error:', error);
    return NextResponse.json({ error: 'Webhook processing failed' }, { status:
500 });
  }
}

async function handleRecordingComplete(event: any, supabase: any) {
  const { sid, cname, fileList } = event;

  // Update recording with file URLs
  await supabase
    .from('recordings')
    .update({
      status: 'completed',
      file_list: fileList
    })
    .eq('sid', sid);

  // Get recording to find episode_id
  const { data: recording } = await supabase
    .from('recordings')
    .select('episode_id')
    .eq('sid', sid)
    .single();

  if (recording?.episode_id) {
    // Update episode with audio URL
    const audioUrl = fileList?.[0]?.fileName || '';
    await supabase
      .from('episodes')
      .update({
        audio_url: audioUrl,
        audio_storage_path: audioUrl,
        is_published: true,
        published_at: new Date().toISOString()
      })
      .eq('id', recording.episode_id);
  }
}

async function handleRecordingError(event: any, supabase: any) {
  const { sid } = event;
  await supabase
    .from('recordings')
    .update({ status: 'failed' })
    .eq('sid', sid);
}
```

---

5. Live Sessions (Talk Shows & Interviews)

5.1 Role Management with Agora Signaling v2

Create `components/LiveSession.tsx`:

```
import { useState, useEffect } from 'react';
import { useAgoraRTM } from '@agoraio-extensions/agora-rtm-react';

interface LiveSessionProps {
```

```
  channelName: string;
  userId: string;
  userName: string;
  isHost: boolean;
}

export function LiveSession({ channelName, userId, userName, isHost }:
LiveSessionProps) {
  const [role, setRole] = useState<'host' | 'audience'>(isHost ? 'host' :
'audience');
  const [speakingRequests, setSpeakingRequests] = useState<any[]>([]);
  const [myRequestStatus, setMyRequestStatus] = useState<'none' | 'pending' |
'approved' | 'denied'>('none');

  // Initialize Agora Signaling v2
  const { client, login, sendMessage, onMessage } = useAgoraRTM({
    appId: process.env.NEXT_PUBLIC_AGORA_APP_ID!,
    channel: channelName
  });

  useEffect(() => {
    if (userId) {
      login(userId);
    }
  }, [userId, login]);

  // Listen for speaking requests
  useEffect(() => {
    if (isHost) {
      const unsubscribe = onMessage((message: any) => {
        if (message.type === 'speak_request') {
          setSpeakingRequests(prev => [...prev, {
            userId: message.userId,
            userName: message.userName,
            message: message.text,
            timestamp: new Date()
          }]);
        }
      });
      return unsubscribe;
    }
  }, [isHost, onMessage]);

  const requestToSpeak = async () => {
    if (!isHost) {
      // Send request to host
      await sendMessage({
        type: 'speak_request',
        userId: userId,
        userName: userName,
        text: 'Requesting to speak'
      });
      setMyRequestStatus('pending');
    }
  };

  const approveSpeaker = async (userId: string) => {
    await sendMessage({
      type: 'speak_approved',
      userId: userId
    });
    setSpeakingRequests(prev => prev.filter(req => req.userId !== userId));
  };
```

```
  const denySpeaker = async (userId: string) => {
    await sendMessage({
      type: 'speak_denied',
      userId: userId
    });
    setSpeakingRequests(prev => prev.filter(req => req.userId !== userId));
  };

  // Listen for host response (for audience)
  useEffect(() => {
    if (!isHost) {
      const unsubscribe = onMessage((message: any) => {
        if (message.type === 'speak_approved' && message.userId === userId) {
          setMyRequestStatus('approved');
          // Grant speaker permissions
          setClientRole('host');
        } else if (message.type === 'speak_denied' && message.userId === userId)
{
          setMyRequestStatus('denied');
        }
      });
      return unsubscribe;
    }
  }, [isHost, userId, onMessage]);

  return (
    <div className="live-session">
      <div className="role-controls">
        <h3>Your Role: {role === 'host' ? '🎙️ Host' : '👂 Listener'}</h3>
        {!isHost && myRequestStatus === 'none' && (
          <button onClick={requestToSpeak}>Request to Speak</button>
        )}
        {myRequestStatus === 'pending' && (
          <div>⏳ Request pending approval...</div>
        )}
        {myRequestStatus === 'approved' && (
          <div>✅ You can now speak! Your mic is active.</div>
        )}
        {myRequestStatus === 'denied' && (
          <div>❌ Your request was denied. Please try again later.</div>
        )}
      </div>

      {isHost && (
        <div className="speaking-requests">
          <h4>Speaking Requests</h4>
          {speakingRequests.length === 0 ? (
            <p>No pending requests</p>
          ) : (
            speakingRequests.map((req, index) => (
              <div key={index} className="request">
                <span>{req.userName} wants to speak</span>
                <button onClick={() => approveSpeaker(req.userId)}>✅
Approve</button>
                <button onClick={() => denySpeaker(req.userId)}>❌ Deny</button>
              </div>
            ))
          )}
        </div>
      )}
    </div>
  );
}
```

5.2 Using Supabase for Persistent Speaking Requests

```
// Database-driven speaking requests
import { useSupabase } from '@supabase/auth-helpers-react';

export function SpeakingRequestSystem({ showId, userId, channelName }: {
  showId: string;
  userId: string;
  channelName: string;
}) {
  const supabase = useSupabase();
  const [requests, setRequests] = useState<any[]>([]);
  const [myRequest, setMyRequest] = useState<any>(null);

  useEffect(() => {
    // Subscribe to requests
    const subscription = supabase
      .channel(`speaking_requests:${showId}`)
      .on('postgres_changes',
        {
          event: '*',
          schema: 'public',
          table: 'speaking_requests',
          filter: `show_id=eq.${showId}`
        },
        (payload) => {
          refreshRequests();
        }
      )
      .subscribe();

    refreshRequests();
    return () => subscription.unsubscribe();
  }, [showId]);

  const refreshRequests = async () => {
    const { data } = await supabase
      .from('speaking_requests')
      .select('*')
      .eq('show_id', showId)
      .order('requested_at', { ascending: false });

    setRequests(data || []);
    setMyRequest(data?.find(r => r.user_id === userId) || null);
  };

  const requestToSpeak = async (message: string) => {
    const { data } = await supabase
      .from('speaking_requests')
      .insert({
        show_id: showId,
        user_id: userId,
        username: 'User', // Get from auth
        message,
        status: 'pending'
      })
      .select()
      .single();

    if (data) {
      setMyRequest(data);
    }
  };
```

```
  const approveRequest = async (requestId: string) => {
    await supabase
      .from('speaking_requests')
      .update({ status: 'approved', resolved_at: new Date().toISOString() })
      .eq('id', requestId);
  };

  const denyRequest = async (requestId: string) => {
    await supabase
      .from('speaking_requests')
      .update({ status: 'denied', resolved_at: new Date().toISOString() })
      .eq('id', requestId);
  };

  return (
    <div>
      {/* Render request UI */}
    </div>
  );
}
```

---

6. Auto-DJ & Playlist Management

6.1 Auto-DJ Architecture

Important: Auto-DJ must run on a persistent server, not Vercel.

Option A: External Bot (Recommended)

Deploy a Node.js bot on Railway/Render/EC2:

```
// bot/src/index.ts
import express from 'express';
import AgoraRTC from 'agora-rtc-sdk-ng';
import { createClient } from '@supabase/supabase-js';

const app = express();
app.use(express.json());

const supabase = createClient(
  process.env.SUPABASE_URL!,
  process.env.SUPABASE_KEY!
);

const rtcClient = AgoraRTC.createClient({ mode: 'live', codec: 'vp8' });

interface Track {
  id: string;
  title: string;
  artist: string;
  storage_url: string;
  duration: number;
}

let currentPlaylist: Track[] = [];
let isPlaying = false;
let currentIndex = 0;
let currentChannelName = '';
let playbackTimer: NodeJS.Timeout | null = null;
```

```
// Start Auto-DJ
app.post('/start', async (req, res) => {
  const { channelName, playlistId } = req.body;

  if (!channelName || !playlistId) {
    return res.status(400).json({ error: 'Missing channelName or playlistId' });
  }

  try {
    // Fetch playlist tracks
    const { data: tracks, error } = await supabase
      .from('playlist_tracks')
      .select(`
        track_id,
        sort_order,
        tracks (
          id,
          title,
          artist,
          storage_url,
          duration
        )
      `)
      .eq('playlist_id', playlistId)
      .order('sort_order');

    if (error) throw error;

    currentPlaylist = tracks.map((pt: any) => pt.tracks);
    currentIndex = 0;
    currentChannelName = channelName;

    // Join Agora channel
    await rtcClient.join(
      process.env.AGORA_APP_ID!,
      channelName,
      generateToken(channelName, 0),
      0
    );

    isPlaying = true;
    playNextTrack();

    res.json({
      status: 'started',
      trackCount: currentPlaylist.length
    });

  } catch (error) {
    console.error('Start error:', error);
    res.status(500).json({ error: 'Failed to start Auto-DJ' });
  }
});

// Stop Auto-DJ
app.post('/stop', async (req, res) => {
  isPlaying = false;
  if (playbackTimer) {
    clearTimeout(playbackTimer);
    playbackTimer = null;
  }
  await rtcClient.leave();
  res.json({ status: 'stopped' });
});
```

```
// Get current status
app.get('/status', (req, res) => {
  res.json({
    isPlaying,
    currentTrack: currentPlaylist[currentIndex] || null,
    playlistLength: currentPlaylist.length,
    channelName: currentChannelName
  });
});

function playNextTrack() {
  if (!isPlaying || currentPlaylist.length === 0) return;

  const track = currentPlaylist[currentIndex];

  // Play audio track via Agora
  // Implementation depends on Agora SDK version

  // Update "Now Playing" in Supabase
  supabase
    .from('channels')
    .update({ current_track_id: track.id })
    .eq('slug', currentChannelName);

  // Schedule next track
  playbackTimer = setTimeout(() => {
    currentIndex = (currentIndex + 1) % currentPlaylist.length;
    playNextTrack();
  }, track.duration * 1000);
}

function generateToken(channelName: string, uid: number): string {
  // Implement token generation
  // Same as in API routes
}

app.listen(3000, () => console.log('Auto-DJ Bot running on port 3000'));
```

6.2 Bot Dependencies

```
{
  "dependencies": {
    "agora-rtc-sdk-ng": "^1.2.0",
    "express": "^4.18.2",
    "@supabase/supabase-js": "^2.49.0"
  }
}
```

6.3 Auto-DJ Control Endpoints

Create `app/api/auto-dj/start/route.ts`:

```
import { NextRequest, NextResponse } from 'next/server';

const BOT_URL = process.env.AUTO_DJ_BOT_URL!;
const BOT_SECRET = process.env.AUTO_DJ_BOT_SECRET!;

export async function POST(req: NextRequest) {
  try {
    const { channelName, playlistId } = await req.json();
```

```
    if (!channelName || !playlistId) {
      return NextResponse.json({ error: 'Missing required fields' }, { status:
400 });
    }

    const response = await fetch(`${BOT_URL}/start`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        'X-Bot-Secret': BOT_SECRET
      },
      body: JSON.stringify({ channelName, playlistId })
    });

    if (!response.ok) {
      const error = await response.text();
      console.error('Bot start error:', error);
      return NextResponse.json({ error: 'Failed to start Auto-DJ bot' },
{ status: 502 });
    }

    const data = await response.json();
    return NextResponse.json(data);

  } catch (error) {
    console.error('Auto-DJ start error:', error);
    return NextResponse.json({ error: 'Internal server error' }, { status:
500 });
  }
}
```

---

7. Database & Storage

7.1 Set Up Supabase

Create `lib/supabase.ts`:

```
import { createClient } from '@supabase/supabase-js';

const supabaseUrl = process.env.NEXT_PUBLIC_SUPABASE_URL!;
const supabaseAnonKey = process.env.NEXT_PUBLIC_SUPABASE_ANON_KEY!;

export const supabase = createClient(supabaseUrl, supabaseAnonKey);
```

7.2 Create Database Tables

Run this in Supabase SQL Editor:

```
-- ============================================
-- ENABLE EXTENSIONS
-- ============================================
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

-- ============================================
-- USERS TABLE (extends Supabase Auth)
-- ============================================
CREATE TABLE users (
  id UUID PRIMARY KEY REFERENCES auth.users(id) ON DELETE CASCADE,
  email TEXT UNIQUE NOT NULL,
```

```
  username TEXT UNIQUE NOT NULL,
  display_name TEXT,
  role TEXT DEFAULT 'listener' CHECK (role IN ('admin', 'dj', 'host',
'listener')),
  avatar_url TEXT,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_users_role ON users(role);
CREATE INDEX idx_users_username ON users(username);

-- ============================================
-- CHANNELS TABLE (State Management)
-- ============================================
CREATE TABLE channels (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name TEXT UNIQUE NOT NULL,
  slug TEXT UNIQUE NOT NULL,
  description TEXT,
  current_mode TEXT DEFAULT 'auto-dj' CHECK (current_mode IN ('live', 'auto-dj',
'off')),
  current_show_id UUID,
  current_track_id UUID,
  listener_count INTEGER DEFAULT 0,
  is_recording BOOLEAN DEFAULT false,
  recording_resource_id TEXT,
  recording_sid TEXT,
  updated_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_channels_slug ON channels(slug);

-- ============================================
-- SHOWS TABLE
-- ============================================
CREATE TABLE shows (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  title TEXT NOT NULL,
  host_id UUID REFERENCES users(id) ON DELETE SET NULL,
  description TEXT,
  channel_id UUID REFERENCES channels(id) ON DELETE CASCADE NOT NULL,
  scheduled_start TIMESTAMPTZ,
  scheduled_end TIMESTAMPTZ,
  is_live BOOLEAN DEFAULT false,
  mode TEXT DEFAULT 'live' CHECK (mode IN ('live', 'auto-dj', 'hybrid')),
  thumbnail_url TEXT,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_shows_host ON shows(host_id);
CREATE INDEX idx_shows_channel ON shows(channel_id);
CREATE INDEX idx_shows_scheduled ON shows(scheduled_start, scheduled_end);

-- ============================================
-- SCHEDULE TABLE
-- ============================================
CREATE TABLE schedule (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  show_id UUID REFERENCES shows(id) ON DELETE CASCADE,
  day_of_week INTEGER CHECK (day_of_week BETWEEN 0 AND 6),
  start_time TIME NOT NULL,
  end_time TIME NOT NULL,
```

```
  timezone TEXT DEFAULT 'UTC',
  mode TEXT NOT NULL CHECK (mode IN ('live', 'auto-dj', 'hybrid')),
  is_active BOOLEAN DEFAULT true,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_schedule_day ON schedule(day_of_week, start_time);

-- ============================================
-- TRACKS TABLE (Music Library)
-- ============================================
CREATE TABLE tracks (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  title TEXT NOT NULL,
  artist TEXT NOT NULL,
  album TEXT,
  duration INTEGER NOT NULL, -- in seconds
  storage_path TEXT NOT NULL, -- S3/Supabase Storage path
  storage_url TEXT, -- cached public URL
  license_type TEXT DEFAULT 'royalty-free' CHECK (license_type IN ('royalty-
free', 'licensed', 'community', 'original')),
  license_reference TEXT,
  genre TEXT,
  play_count INTEGER DEFAULT 0,
  last_played_at TIMESTAMPTZ,
  uploaded_by UUID REFERENCES users(id) ON DELETE SET NULL,
  is_active BOOLEAN DEFAULT true,
  created_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_tracks_active ON tracks(is_active);
CREATE INDEX idx_tracks_uploaded_by ON tracks(uploaded_by);

-- ============================================
-- PLAYLISTS TABLE
-- ============================================
CREATE TABLE playlists (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name TEXT NOT NULL,
  description TEXT,
  is_default BOOLEAN DEFAULT false,
  is_fallback BOOLEAN DEFAULT false,
  created_by UUID REFERENCES users(id) ON DELETE SET NULL,
  created_at TIMESTAMPTZ DEFAULT NOW()
);

-- ============================================
-- PLAYLIST_TRACKS TABLE
-- ============================================
CREATE TABLE playlist_tracks (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  playlist_id UUID REFERENCES playlists(id) ON DELETE CASCADE,
  track_id UUID REFERENCES tracks(id) ON DELETE CASCADE,
  sort_order INTEGER NOT NULL DEFAULT 0,
  added_by UUID REFERENCES users(id) ON DELETE SET NULL,
  added_at TIMESTAMPTZ DEFAULT NOW(),
  UNIQUE(playlist_id, track_id, sort_order)
);

CREATE INDEX idx_playlist_tracks_order ON playlist_tracks(playlist_id,
sort_order);

-- ============================================
```

```
-- EPISODES TABLE
-- ============================================
CREATE TABLE episodes (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  show_id UUID REFERENCES shows(id) ON DELETE SET NULL,
  title TEXT NOT NULL,
  description TEXT,
  channel_id UUID REFERENCES channels(id) ON DELETE SET NULL,
  audio_url TEXT, -- HLS/m3u8 or MP3 URL
  audio_storage_path TEXT,
  duration INTEGER, -- in seconds
  file_size BIGINT,
  mode TEXT NOT NULL CHECK (mode IN ('live', 'auto-dj')),
  recording_resource_id TEXT,
  recording_sid TEXT,
  listeners_peak INTEGER DEFAULT 0,
  listeners_avg INTEGER DEFAULT 0,
  is_published BOOLEAN DEFAULT true,
  published_at TIMESTAMPTZ,
  created_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_episodes_show ON episodes(show_id);
CREATE INDEX idx_episodes_channel ON episodes(channel_id);
CREATE INDEX idx_episodes_published ON episodes(is_published, published_at);

-- ============================================
-- RECORDINGS TABLE
-- ============================================
CREATE TABLE recordings (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  episode_id UUID REFERENCES episodes(id) ON DELETE CASCADE,
  channel_name TEXT NOT NULL,
  resource_id TEXT NOT NULL,
  sid TEXT NOT NULL,
  status TEXT DEFAULT 'starting' CHECK (status IN ('starting', 'recording',
'stopped', 'failed', 'completed')),
  file_list JSONB, -- Agora file list response
  storage_vendor INTEGER, -- 1=AWS, 2=Aliyun, etc.
  started_at TIMESTAMPTZ DEFAULT NOW(),
  stopped_at TIMESTAMPTZ,
  created_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_recordings_sid ON recordings(sid);
CREATE INDEX idx_recordings_episode ON recordings(episode_id);

-- ============================================
-- SPEAKING_REQUESTS TABLE
-- ============================================
CREATE TABLE speaking_requests (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  show_id UUID REFERENCES shows(id) ON DELETE CASCADE,
  user_id UUID REFERENCES users(id) ON DELETE CASCADE,
  username TEXT NOT NULL,
  message TEXT,
  status TEXT DEFAULT 'pending' CHECK (status IN ('pending', 'approved',
'denied', 'cancelled')),
  requested_at TIMESTAMPTZ DEFAULT NOW(),
  resolved_at TIMESTAMPTZ
);

CREATE INDEX idx_speaking_requests_show ON speaking_requests(show_id, status);
```

```
-- ============================================
-- LISTENER_SESSIONS TABLE
-- ============================================
CREATE TABLE listener_sessions (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  channel_id UUID REFERENCES channels(id) ON DELETE CASCADE,
  user_id UUID REFERENCES users(id) ON DELETE SET NULL,
  anonymous_id TEXT, -- for non-authenticated listeners
  joined_at TIMESTAMPTZ DEFAULT NOW(),
  left_at TIMESTAMPTZ,
  ip_hash TEXT, -- hashed IP for deduplication
  user_agent TEXT
);

CREATE INDEX idx_listener_sessions_channel ON listener_sessions(channel_id,
joined_at);

-- ============================================
-- ROW LEVEL SECURITY POLICIES
-- ============================================

-- Users: read own, admin read all
ALTER TABLE users ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Users can read own profile"
  ON users FOR SELECT
  USING (auth.uid() = id OR EXISTS (
    SELECT 1 FROM users u WHERE u.id = auth.uid() AND u.role = 'admin'
  ));

CREATE POLICY "Users can update own profile"
  ON users FOR UPDATE
  USING (auth.uid() = id);

-- Shows: public read, DJ/admin write
ALTER TABLE shows ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Shows are viewable by everyone"
  ON shows FOR SELECT USING (true);

CREATE POLICY "DJs can manage own shows"
  ON shows FOR ALL
  USING (auth.uid() = host_id OR EXISTS (
    SELECT 1 FROM users u WHERE u.id = auth.uid() AND u.role IN ('admin', 'dj')
  ));

-- Schedule: public read, DJ/admin write
ALTER TABLE schedule ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Schedule is viewable by everyone"
  ON schedule FOR SELECT USING (true);

CREATE POLICY "DJs can manage schedule"
  ON schedule FOR ALL
  USING (EXISTS (
    SELECT 1 FROM users u WHERE u.id = auth.uid() AND u.role IN ('admin', 'dj')
  ));

-- Tracks: public read active, uploader/admin write
ALTER TABLE tracks ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Active tracks are viewable by everyone"
  ON tracks FOR SELECT USING (is_active = true);
```

```
CREATE POLICY "Admins can manage tracks"
  ON tracks FOR ALL
  USING (EXISTS (
    SELECT 1 FROM users u WHERE u.id = auth.uid() AND u.role = 'admin'
  ));

-- Playlists: public read, admin write
ALTER TABLE playlists ENABLE ROW LEVEL SECURITY;
ALTER TABLE playlist_tracks ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Playlists are viewable by everyone"
  ON playlists FOR SELECT USING (true);

CREATE POLICY "Playlist tracks are viewable by everyone"
  ON playlist_tracks FOR SELECT USING (true);

-- Episodes: public read published, admin write
ALTER TABLE episodes ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Published episodes are viewable by everyone"
  ON episodes FOR SELECT USING (is_published = true);

CREATE POLICY "Admins can manage episodes"
  ON episodes FOR ALL
  USING (EXISTS (
    SELECT 1 FROM users u WHERE u.id = auth.uid() AND u.role = 'admin'
  ));

-- Speaking requests: host/DJ read, user create own
ALTER TABLE speaking_requests ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Hosts can view show requests"
  ON speaking_requests FOR SELECT
  USING (EXISTS (
    SELECT 1 FROM shows s WHERE s.id = show_id AND s.host_id = auth.uid()
  ) OR auth.uid() = user_id);

CREATE POLICY "Listeners can create requests"
  ON speaking_requests FOR INSERT
  WITH CHECK (auth.uid() = user_id);

-- Channels: public read, admin write
ALTER TABLE channels ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Channels are viewable by everyone"
  ON channels FOR SELECT USING (true);

-- Recordings: admin only
ALTER TABLE recordings ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Recordings admin only"
  ON recordings FOR ALL
  USING (EXISTS (
    SELECT 1 FROM users u WHERE u.id = auth.uid() AND u.role = 'admin'
  ));

-- Listener sessions: admin only
ALTER TABLE listener_sessions ENABLE ROW LEVEL SECURITY;

CREATE POLICY "Listener sessions admin only"
  ON listener_sessions FOR ALL
  USING (EXISTS (
    SELECT 1 FROM users u WHERE u.id = auth.uid() AND u.role = 'admin'
  ));
```

```
-- ============================================
-- REALTIME PUBLICATION
-- ============================================
ALTER PUBLICATION supabase_realtime ADD TABLE channels;
ALTER PUBLICATION supabase_realtime ADD TABLE speaking_requests;
```

7.3 Save Recording to Database

```
// After recording stops
async function saveEpisode(episodeData: any) {
  const { data, error } = await supabase
    .from('episodes')
    .insert([{
      title: episodeData.title,
      audio_url: episodeData.audioUrl,
      duration: episodeData.duration,
      channel_name: episodeData.channelName
    }]);

  if (error) {
    console.error('Error saving episode:', error);
    return null;
  }
  return data;
}
```

7.4 List Past Episodes

```
async function getEpisodes() {
  const { data, error } = await supabase
    .from('episodes')
    .select('*')
    .order('created_at', { ascending: false });

  if (error) {
    console.error('Error fetching episodes:', error);
    return [];
  }
  return data;
}
```

---

8. Deployment

8.1 Deploy to Vercel

```
# Build your app
npm run build

# Deploy to Vercel
vercel --prod
```

8.2 Add Environment Variables to Vercel

In Vercel Dashboard:

1. Go to your project → Settings → Environment Variables

2. Add all variables from your `.env.local` file
3. Click Save

8.3 Deploy Auto-DJ Bot

1. Deploy the bot code to Railway/Render/EC2
2. Set up environment variables on the bot platform
3. Ensure the bot URL is accessible from Vercel

8.4 Set Up Icecast Server

1. Install Icecast on a VPS
2. Configure mount points and authentication
3. Set up SSL certificates
4. Update `NEXT_PUBLIC_ICECAST_URL` in environment variables

8.5 Test Your Deployment

1. Open your Vercel URL
2. Start a broadcast
3. Have another device join as a listener
4. Test volume controls and mute
5. Test recording
6. Test talk show features
7. Test Auto-DJ functionality

---

9. Troubleshooting

Issue	**Solution**
No audio	Check microphone permissions in browser
Token error	Verify App ID and Certificate are correct
Recording fails	Check cloud storage credentials and permissions
Volume not working	Use correct methods: `setVolume()` for local, `setPlaybackVolume()` for global
Can't hear remote user	Ensure `useRemoteAudioTrack` is used correctly
Webhook not firing	Check endpoint URL is public and accessible
Broadcast lags	Check network quality and reduce bitrate if needed

Issue	Solution
Auto-DJ not starting	Verify bot is running and URL is correct
RTM not working	Use Signaling v2, not deprecated RTM v1

---

10. Checklist

Pre-Launch Checklist

- ☐ Agora App ID and Certificate are set up
- ☐ Token generation is working securely on server
- ☐ DJ can start/stop broadcast
- ☐ Listeners can join and hear audio
- ☐ Volume controls work (DJ and listener)
- ☐ Mute functionality works
- ☐ Cloud Recording starts and stops
- ☐ Recordings save to cloud storage
- ☐ Webhook receives recording completion
- ☐ Episodes save to database
- ☐ Users can see past episodes
- ☐ Role switching works (host ↔ audience)
- ☐ "Request to Speak" works
- ☐ Environment variables are set in Vercel
- ☐ Build passes on Vercel
- ☐ Production deployment is live
- ☐ Auto-DJ bot is deployed and running
- ☐ Icecast server is configured
- ☐ Emergency Fill logic is tested

Post-Launch Monitoring

- ☐ Set up Vercel Analytics
- ☐ Add error tracking (Sentry)
- ☐ Monitor Agora usage/costs
- ☐ Test with multiple concurrent users

- ☐ Test with different devices/browsers
- ☐ Monitor Auto-DJ bot performance
- ☐ Set up backup/fallback mechanisms

Helpful Resources

- Agora Documentation
- Vercel Documentation
- Supabase Documentation
- Next.js Documentation
- Icecast Documentation

Congratulations! You now have a complete, community-ready radio app built on Vercel, Agora, Supabase, and Icecast with full Auto-DJ support! 🎙️📻