LiveKit for Voice AI: Why You Shouldn't Build Custom Audio Buffers

Executive Summary

LiveKit is a production-grade, open-source WebRTC framework specifically designed for real-time voice and video AI applications. Building custom audio buffers for voice AI is **strongly discouraged** because it requires solving extraordinarily complex problems around network jitter, packet loss, timing synchronization, and cross-browser compatibility—problems that have taken WebRTC engineers **over a decade** to solve and refine. LiveKit abstracts away 95% of this complexity while providing sub-800ms latency, automatic interruption handling, and battle-tested reliability that would take 12-18 months to replicate from scratch.

What is LiveKit?

LiveKit is a **full-stack realtime communication platform** that provides three core components:

1. WebRTC Server (SFU Architecture)

- Open-source Selective Forwarding Unit that handles all network transport
- Manages signaling, NAT traversal, RTP routing, adaptive degradation
- Production-grade infrastructure used by companies like OpenAI (powers ChatGPT Advanced Voice Mode)

2. Agents Framework

- Python/Node.js SDKs for building AI participants that join rooms
- Native support for STT-LLM-TTS pipelines with any provider
- Built-in turn detection, interruption handling, session management

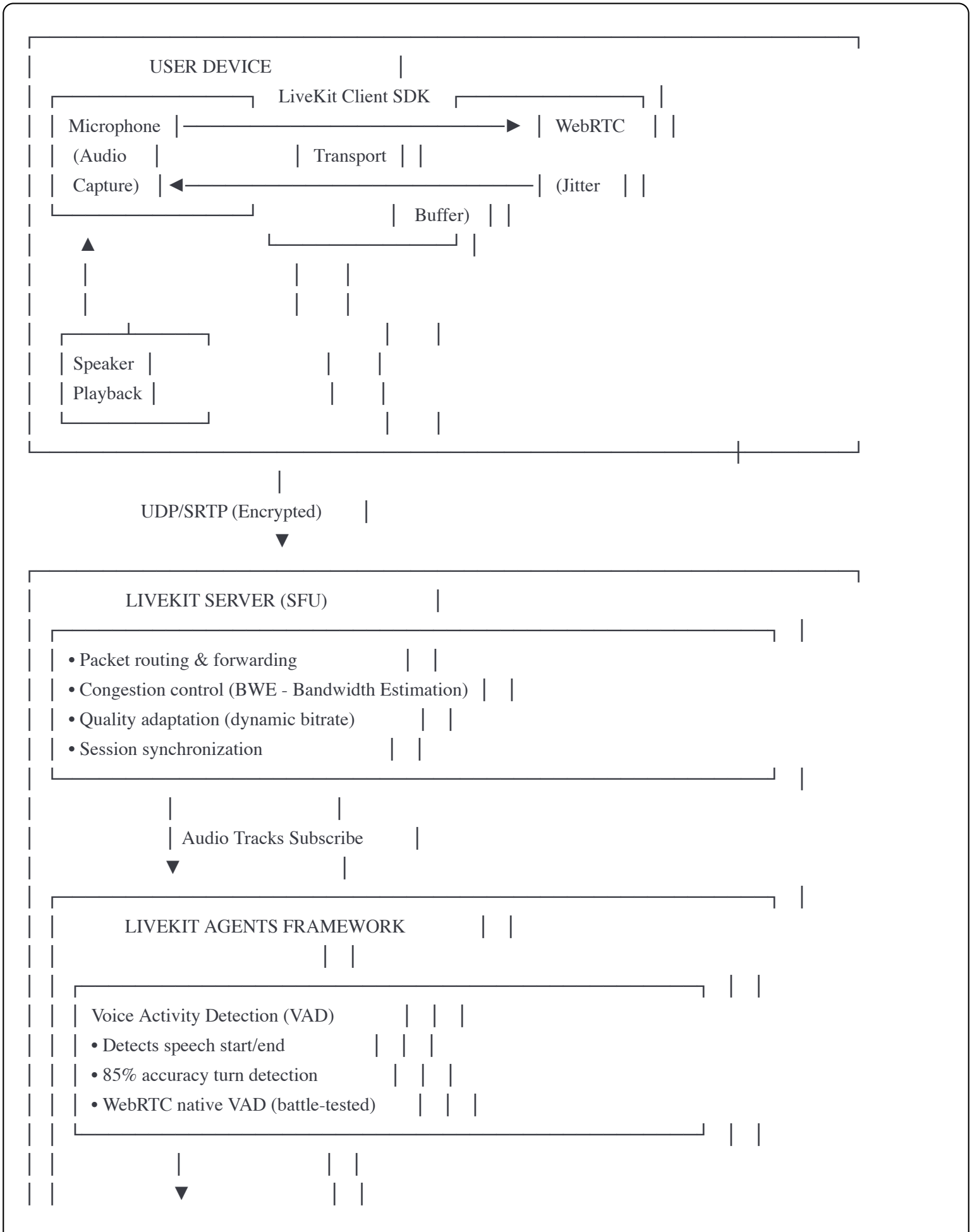
3. Client SDKs

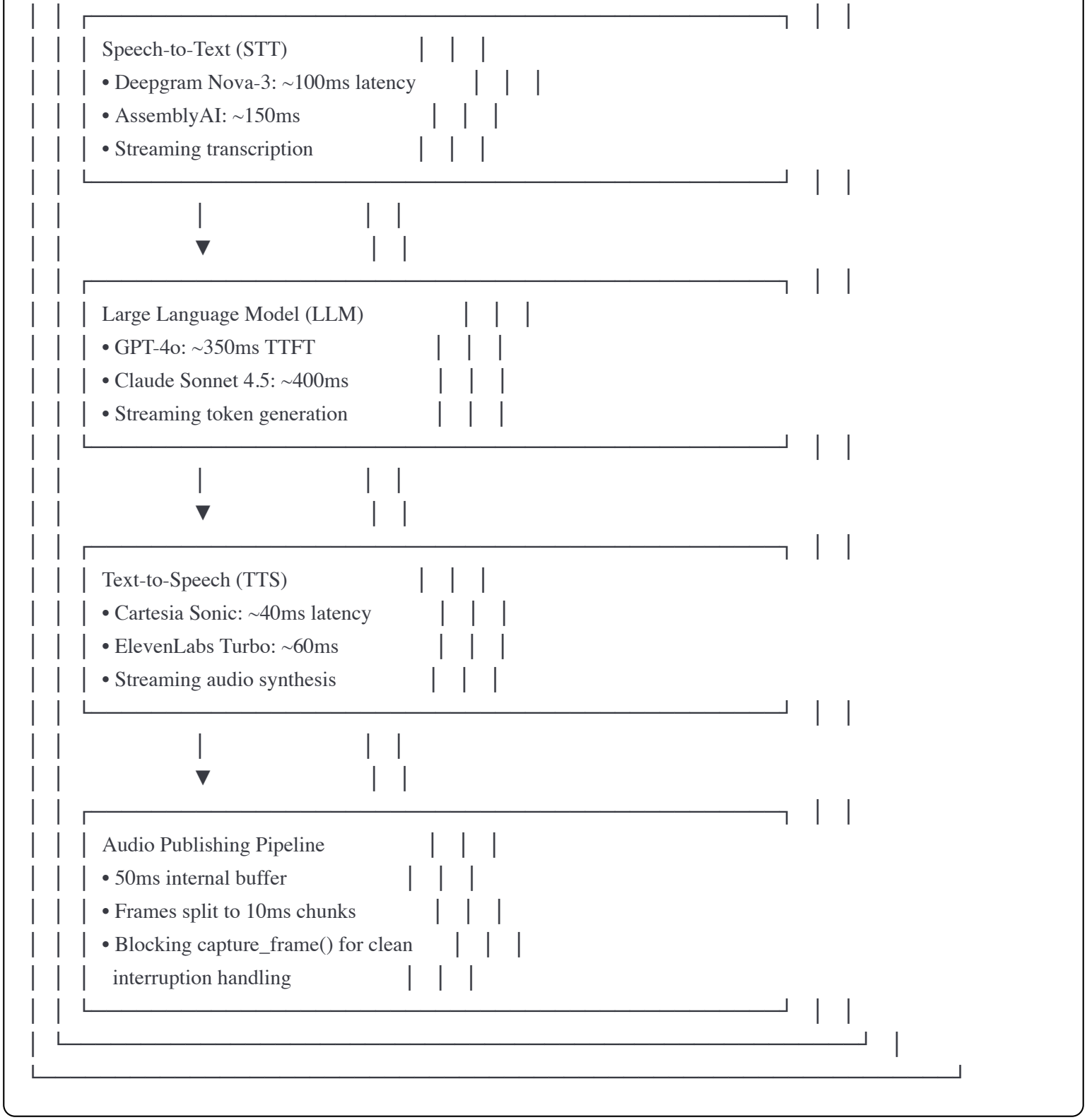
- Web, iOS, Android, Flutter, React Native, Unity support
- WebRTC abstraction layer—no raw WebRTC manipulation needed
- Single API for audio/video/data streaming across all platforms

Key Distinction: LiveKit isn't just a voice AI framework—it's a complete realtime transport layer that happens to be excellent for voice AI. This means you get professional-grade WebRTC infrastructure without becoming a WebRTC expert.

The LiveKit Voice Stack Architecture

Here's how LiveKit handles a typical voice AI interaction:





Total End-to-End Latency: 510-800ms (voice-to-voice)

Why Building Custom Audio Buffers is a Terrible Idea

1. The WebRTC Audio Jitter Buffer Problem

Audio jitter buffers must solve a fundamentally difficult problem: **packets arrive at unpredictable times, but audio must play out smoothly at constant intervals**.

What a Custom Implementation Must Handle:

Network Jitter Management:

- Packets arrive with variable delay (5-500ms variance is common)
- You need to buffer enough audio to smooth out timing variations
- Too small buffer = stuttering/dropouts when packets are delayed
- Too large buffer = high latency (unacceptable for conversation)
- The "sweet spot" constantly changes based on network conditions

Adaptive Buffer Sizing: WebRTC's NetEQ (the production jitter buffer) uses sophisticated algorithms:

- Maintains a 2-second history window to classify temporary jitter vs. permanent delay changes
- Continuously calculates "relative delay" to adjust buffer size dynamically
- Default range: 15-120ms for audio, starting around 40ms
- Adjusts target buffer level 100+ times per second

Packet Loss Handling:

- ~3% packet loss is typical; 10%+ is not uncommon on mobile networks
- Must detect missing packets via RTP sequence numbers
- Options: silence insertion, packet loss concealment (PLC), forward error correction (FEC)
- WebRTC's NetEQ includes ML-based audio reconstruction for missing packets

Packet Reordering:

- Packets can arrive out-of-order (common with retransmissions)
- Must maintain correct sequence using timestamps
- Late packets create "should I wait or skip?" decisions

Time-Stretching Algorithms: When the buffer gets too full/empty, you can't just drop/duplicate packets—this creates audible artifacts. Instead, NetEQ uses:

- Accelerate: Speed up audio slightly without pitch change (if buffer too full)
- Expand: Slow down audio slightly (if buffer too empty)
- These algorithms operate on individual audio frames and are **extremely complex**

Clock Drift Compensation:

Clock Drift Compensation:

- Client and server clocks are never perfectly synchronized
- A 0.01% drift = 3.6 seconds of drift per 10-hour session
- Requires continuous micro-adjustments

The Code Complexity Reality:

WebRTC's NetEQ implementation is **over 100 files** of heavily optimized C++ code. Core components:

```cpp
// Simplified conceptual view - actual implementation is 10x more complex
class NetEQ {
    PacketBuffer packet_buffer_;      // Store incoming packets
    DelayManager delay_manager_;      // Calculate target delay
    DecisionLogic decision_logic_;    // Decide operations per 10ms
    AccelerateAlgorithm accelerate_;  // Time-stretching acceleration
    ExpandAlgorithm expand_;          // Time-stretching expansion
    PreemptiveExpand preemptive_;     // Gradual expansion
    Merge merge_;                     // Smooth transitions
    Normal normal_;                   // Standard playback

    // Called 100 times per second - MUST return exactly 10ms of audio
    int GetAudio(AudioFrame* output);

    // Handles arriving packets with jitter/loss/reorder
    int InsertPacket(const RTPHeader& rtp_header,
                     const uint8_t* payload,
                     size_t length,
                     uint32_t receive_timestamp);
};
```

A developer attempting this from scratch typically faces:

- 3-6 months to get basic buffering working
- 6-12 months to handle packet loss gracefully
- 12-18 months to achieve production-quality audio under varying network conditions
- Never quite reaching the quality/reliability of 10+ years of WebRTC optimization

2. Cross-Browser/Platform Consistency Hell

Different browsers and platforms have different audio APIs and behaviors:

Web Audio API Variations:

- Chrome: `ScriptProcessorNode` deprecated, use `AudioWorklet`
- Safari: Different buffer size requirements (powers-of-2 only in some versions)
- Firefox: Different default sample rates
- Mobile browsers: Strict autoplay policies, different latency characteristics

iOS Specific Nightmares:

- iOS only supports WebRTC in Safari (not Chrome/Firefox on iOS)
- `getUserMedia()` doesn't work in non-Safari mobile browsers
- Strict audio session management—backgrounding app can kill audio
- Different buffer sizes needed for optimal performance

Sample Rate Mismatches:

- Browser may capture at 48kHz
- Your AI model expects 16kHz
- Must resample without introducing artifacts
- Resampling is CPU-intensive and can introduce latency

Buffer Size Tradeoffs:

```javascript
// Low buffer = better latency but higher CPU and dropout risk
const processor = context.createScriptProcessor(256, 1, 1);

// High buffer = worse latency but more stable
const processor = context.createScriptProcessor(8192, 1, 1);

// You need dynamic adjustment based on device capability
// Good luck getting this right across all devices!
```

LiveKit handles all of this automatically with platform-specific optimizations.

3. The Voice Interruption Problem

When a user interrupts the AI mid-sentence, you must:

Stop Everything Instantly:

1. Stop text-to-speech generation immediately
2. Flush all queued audio frames (may have 500ms buffered)
3. Cancel in-flight LLM response generation
4. Truncate conversation history to match what user actually heard

Maintain Clean State:

- Audio buffers must be cleared without pops/clicks
- WebRTC connection must remain stable (no dropped packets to client)
- Conversation context must accurately reflect partial responses

Resume Smoothly:

- User's interruption audio needs instant priority
- Switch from "agent speaking" to "user speaking" mode seamlessly
- VAD must activate immediately

The Custom Implementation Challenge:

```javascript
```

```
// What developers think interruption handling looks like:
function handleInterruption() {
  agent.stopSpeaking();
  clearAudioBuffer();
  startListening();
}

// What it actually requires:
async function handleInterruption() {
  // 1. Stop TTS mid-synthesis
  await tts.cancel();

  // 2. Flush WebSocket/WebRTC buffers
  //    (but how? Different per transport)
  await flushTransportBuffer();

  // 3. Clear local audio buffers without artifacts
  audioWorklet.port.postMessage({ type: 'clear', fade: 20 }); // 20ms fade

  // 4. Stop LLM generation
  llmController.abort();

  // 5. Truncate conversation history
  //    (to what the user actually heard - requires precise timing)
  const hearableText = calculateHearableText(
    ttsStartTime,
    interruptTime,
    characterTimestamps,
    ttsSpeed
  );
  conversationHistory[lastIndex].content = hearableText;

  // 6. Reconfigure VAD for user input
  vad.configure({ mode: 'listen', sensitivity: 0.7 });

  // 7. Handle race conditions
  //    (what if interruption happens during previous interruption?)
  if (currentState === 'interrupting') {
    return; // Already handling one
  }

  // And about 20 more edge cases...
}
```

LiveKit's Solution:

```python
# This is all you need - LiveKit handles everything automatically
session = AgentSession(
    llm=openai.LLM(),
    stt=deepgram.STT(),
    tts=cartesia.TTS()
)

# Interruptions are handled automatically by the framework
# - Audio buffers managed internally
# - VAD switches modes automatically
# - Conversation state maintained correctly
```

4. The WebSocket vs. WebRTC Performance Gap

Many developers try WebSockets first because they're familiar. This is a mistake.

WebSockets for Voice:

- No built-in jitter compensation
- No automatic congestion control
- No adaptive bitrate
- No packet loss recovery
- You must implement all buffering manually
- Data loss results in "squeaks" and artifacts

WebRTC for Voice:

- Built-in NetEQ jitter buffer
- Automatic bandwidth estimation and adaptation
- Forward error correction (FEC) options
- Packet loss concealment
- Native browser audio pipeline integration
- Optimized for real-time media (not generic data)

Developer Experience: One developer's journey (from blog post research):

Developer Experience: One developer's journey (from blog post research):

1. Started with REST API (record → upload → process) - Too slow
2. Tried WebSockets - "Light squeaks in audio regularly"
3. Required FFMPEG for audio pre-cleaning
4. Needed custom circular buffers with backpressure management
5. Built custom VAD scoring system
6. Python dependency added just for audio processing
7. Finally switched to WebRTC and "everything just worked"

Performance Data:

Metric	WebSockets (Custom)	WebRTC (LiveKit)
Development time	3-6 months	1-2 days
Audio quality issues	Frequent squeaks, dropouts	Rare
Packet loss handling	Manual, error-prone	Automatic, robust
Mobile performance	Inconsistent	Optimized
Latency (end-to-end)	1200-2000ms	510-800ms

5. The Voice Activity Detection (VAD) Complexity

Detecting when a user starts/stops speaking seems simple. It's not.

Naive Approach:

```javascript
if (audioLevel > threshold) {
  userIsSpeaking = true;
}
```

Real Requirements:

- Distinguish speech from background noise
- Handle varying audio levels (whispers vs. loud speech)

- Avoid false triggers from:
    - Keyboard typing
    - Paper rustling
    - Coughing/clearing throat
    - Music playing in background
    - Car engine noise (critical for your use case!)

WebRTC's Built-in VAD:

- Uses ML-based voice detection trained on millions of samples
- Returns confidence scores, not binary yes/no
- Requires pre-processing (FFMPEG to filter non-human frequencies)
- Sends results 100+ times per second—you must build scoring logic

LiveKit's Semantic Turn Detection:

- Uses transformer models to understand conversational turns
- 85% true positive rate for detecting "user is done speaking"
- Handles natural pauses mid-sentence
- Avoids premature interruptions

Implementing this yourself = 6+ months of ML engineering.

6. Production Reliability and Edge Cases

Custom implementations fail on edge cases:

Network Conditions:

- High jitter (±200ms variation)
- Packet bursts (sudden arrival of 10+ packets)
- Asymmetric bandwidth (fast download, slow upload)
- Network switching (WiFi → cellular mid-call)

Device Constraints:

- Low-end mobile devices with limited CPU

- Low-end mobile devices with limited CPU
- Memory pressure causing garbage collection pauses
- Background app restrictions on iOS
- Android audio focus conflicts

Session Management:

- Reconnection after network drop (must resume conversation)
- Browser tab backgrounding (audio context suspension)
- Multi-hour sessions without memory leaks
- Cleanup after ungraceful disconnection

LiveKit's Advantage:

- Battle-tested across millions of production sessions
- Automatic reconnection with state recovery
- Memory-efficient long-running sessions
- Handles all device/platform quirks

What LiveKit Provides That You'd Have to Build

Automatic Features:

1. **Adaptive Jitter Buffer** - Adjusts to network conditions in real-time
2. **Bandwidth Estimation** - Optimizes quality vs. latency dynamically
3. **Packet Loss Concealment** - Reconstructs missing audio intelligently
4. **Cross-Platform Audio APIs** - Works on web, iOS, Android, desktop
5. **Session Management** - Handles reconnection, state recovery
6. **Turn Detection** - ML-based understanding of conversation flow
7. **Interruption Handling** - Clean state management during barge-in
8. **Audio Synchronization** - Keeps audio/video/data in sync
9. **Security** - End-to-end encryption (SRTP) by default
10. **Observability** - Built-in logging, metrics, transcripts

Integration Ecosystem:

LiveKit provides **50+ integrations** with AI providers:

- **STT:** Deepgram, AssemblyAI, Azure, Google, AWS, Whisper
- **LLM:** OpenAI, Anthropic, Google, Cerebras, Groq, local models
- **TTS:** Cartesia, ElevenLabs, Rime, Azure, Google, AWS
- **Realtime Models:** OpenAI Realtime API, Gemini Live
- **Telephony:** SIP integration for phone calls

All with consistent APIs and automatic stream management.

The Time and Cost Equation

Building Custom (from scratch):

- Senior engineer salary: $150K-200K/year
- Development time: 12-18 months to production quality
- Total cost: $150K-300K
- Ongoing maintenance: 1-2 engineers full-time
- Result: Likely still inferior to battle-tested solutions

Using LiveKit:

- Development time: 1-2 weeks to production MVP
- Learning curve: 2-3 days
- Cost: $0 (open source) + hosting costs
- LiveKit Cloud: $0.008/minute for hosted infrastructure
- Maintenance: Framework updates handled by LiveKit team

For Your Email Voice Agent:

- 45-minute commute session on custom stack: High risk of dropouts
- Same session on LiveKit: Reliable, tested under poor network conditions
- Tesla cellular connection issues: LiveKit's adaptive buffer handles gracefully
- Custom stack: Likely results in "Sorry, can you repeat that?" frustration

Code Comparison: Custom vs. LiveKit

Custom WebRTC Voice Pipeline (Simplified):

javascript

```javascript
// ~500-1000 lines to get basic functionality
// This is AFTER you've learned WebRTC internals

// 1. Set up WebRTC peer connection
const pc = new RTCPeerConnection(configuration);

// 2. Handle audio capture with Web Audio API
const audioContext = new AudioContext();
const stream = await navigator.mediaDevices.getUserMedia({ audio: true });
const source = audioContext.createMediaStreamSource(stream);
const processor = audioContext.createScriptProcessor(4096, 1, 1);

// 3. Build jitter buffer (this is where it gets complex)
class JitterBuffer {
  constructor() {
    this.buffer = [];
    this.targetDelay = 40; // ms
    this.currentDelay = 0;
  }

  insert(packet) {
    // Handle out-of-order packets
    // Calculate arrival jitter
    // Adjust target delay
    // Implement packet loss detection
    // Apply time-stretching if needed
    // ~500 lines of complex code
  }

  getAudio() {
    // Must return smooth audio despite network jitter
    // Another ~300 lines
  }
}

// 4. Integrate STT (manage WebSocket connection)
const deepgramSocket = new WebSocket('wss://api.deepgram.com/...');
```

```
deepgramSocket.onmessage = (msg) => {
    // Handle partial transcripts
    // Manage turn detection manually
    // Coordinate with LLM
};

// 5. Send to LLM (manage streaming)
const llmStream = await fetch('https://api.openai.com/...', {
    method: 'POST',
    body: JSON.stringify({ messages, stream: true })
});

// 6. TTS synthesis and playback
const ttsSocket = new WebSocket('wss://api.cartesia.ai/...');
// Manage audio queue
// Handle interruptions
// Coordinate with WebRTC transport

// 7. Handle interruptions (complex state machine)
// ~200 lines of careful state management

// 8. Session management, reconnection, error handling
// Another ~500 lines

// Total: 2000-3000 lines of brittle, hard-to-maintain code
```

LiveKit Agent (Complete):

python

```python
from livekit import agents
from livekit.agents import AgentSession
from livekit.plugins import deepgram, openai, cartesia

class EmailAgent(agents.Agent):
    def __init__(self):
        super().__init__(
            instructions="You are a helpful email assistant for commuters."
        )

    async def on_user_turn_completed(self, chat_ctx, new_message):
        # Handle user's complete utterance
        # All audio streaming, buffering, interruption handling
        # is managed automatically by LiveKit
        print(f"User said: {new_message.text_content}")

async def entrypoint(ctx: agents.JobContext):
    # Configure the voice pipeline
    session = AgentSession(
        stt=deepgram.STT(model="nova-3"),
        llm=openai.LLM(model="gpt-4o"),
        tts=cartesia.TTS(model="sonic", voice="friendly"),
        # Enhanced noise cancellation for cars
        room_input_options=agents.RoomInputOptions(
            noise_cancellation=agents.noise_cancellation.Enhanced()
        )
```

```
    )

    await session.start(EmailAgent())

    # Connect to room
    await ctx.connect()

    # That's it! LiveKit handles:
    # - All WebRTC transport and buffering
    # - Audio streaming to/from AI services
    # - Turn detection and interruptions
    # - Session management and reconnection
    # - Error recovery

# Total: ~30 lines of business logic
```

Code Comparison Summary:

- Custom: 2000-3000 lines of complex audio/network code
- LiveKit: 30-50 lines of business logic
- **100x reduction in complexity**

LiveKit for Your Specific Use Case (Email Voice Agent)

Your requirements map perfectly to LiveKit's strengths:

1. Commute Duration (45-90 minutes)

Challenge: Long sessions over cellular networks with spotty coverage **LiveKit Solution:**

- Automatic reconnection when network drops
- Session state preservation (conversation history maintained)
- Adaptive bitrate adjusts to Tesla's cellular bandwidth
- Tested for multi-hour sessions without memory leaks

2. In-Car Audio Environment

Challenge: Road noise, engine sounds, wind noise through windows **LiveKit Solution:**

```
python
```

```
room_input_options=agents.RoomInputOptions(
    noise_cancellation=agents.noise_cancellation.Enhanced()
)
```

- Enhanced noise cancellation specifically for automotive environments
- WebRTC's echo cancellation (when agent speaks, doesn't pick up own voice)
- Automatic gain control normalizes varying speech volumes

3. User Interruptions ("Hey, wait, go back to that Bob email")

Challenge: User needs to interrupt AI mid-briefing naturally **LiveKit Solution:**

- Semantic turn detection prevents premature cut-offs
- When user speaks, AI stops immediately with clean audio transition
- Conversation context maintained (knows where interruption happened)
- Resume capability ("As I was saying about Bob's email...")

4. Hands-Free Operation

Challenge: No visual interface, pure voice interaction **LiveKit Solution:**

- Full-duplex audio (user can speak anytime, not push-to-talk)
- Voice Activity Detection handles natural conversation flow
- Works seamlessly with Tesla's Bluetooth audio system
- No screen interaction required for entire session

5. Gmail/Outlook API Integration

Challenge: Switching between voice conversation and email actions **LiveKit Solution:**

```python
class EmailAgent(agents.Agent):
    async def on_user_turn_completed(self, chat_ctx, new_message):
        intent = await self.llm.extract_intent(new_message.text_content)

        if intent == "create_draft":
            # User said "Draft a reply to Sarah saying..."
            draft = await gmail_api.create_draft(...)
```

```
        await chat_ctx.respond("I've drafted that email. It will be in your drafts when you arrive.")

    elif intent == "create_rule":
        # User said "Mute notifications from VendorX"
        await outlook_api.create_rule(...)
        await chat_ctx.respond("Done. You won't get notifications from VendorX anymore.")
```

LiveKit handles the voice part; you write the email logic. Clean separation of concerns.

Migration Path: From Prototype to Production

Phase 1: Rapid Prototype (Week 1-2)

```
python

# Use LiveKit Cloud for instant setup
# No infrastructure needed
# Built-in STT/LLM/TTS providers via LiveKit Inference

session = AgentSession(
    stt="deepgram-nova-3",     # Provider handled by LiveKit
    llm="gpt-4o",              # Just specify model string
    tts="cartesia-sonic"       # Instant availability
)
```

Phase 2: Production Optimization (Week 3-4)

```
python

# Switch to your own API keys for cost optimization
session = AgentSession(
    stt=deepgram.STT(api_key=YOUR_KEY, model="nova-3"),
    llm=openai.LLM(api_key=YOUR_KEY, model="gpt-4o"),
    tts=cartesia.TTS(api_key=YOUR_KEY, model="sonic")
)
```

Phase 3: Scale (Month 2+)

- Deploy LiveKit server on your infrastructure (if needed)
- Use LiveKit Cloud's global edge network for low latency
- Add monitoring, logging, transcripts
- A/B test different model combinations

Total Time: 4-6 weeks from idea to production vs. 12-18 months custom.

When Would You NOT Use LiveKit?

You might build custom if:

1. You need 100% proprietary tech (IP reasons)
2. You have a team of WebRTC experts already
3. Your use case requires non-standard audio processing
4. You're building a new WebRTC framework as your product

For an email voice agent for commuters: None of these apply.

Conclusion: The Engineering Reality

Building custom audio buffers for voice AI is an example of **severe overengineering** that provides:

- ❌ 10x longer development time
- ❌ Inferior quality and reliability
- ❌ Massive ongoing maintenance burden
- ❌ High risk of user-facing bugs
- ❌ Platform-specific nightmares
- ✅ Zero competitive advantage

LiveKit provides:

- ✅ Production-grade quality from day one
- ✅ 2-week development timeline
- ✅ Battle-tested reliability
- ✅ Cross-platform support automatically
- ✅ Focus on your product differentiation (email intelligence)
- ✅ Continuous improvements from LiveKit team

For your email voice agent:

- Use LiveKit Agents framework for voice infrastructure

- Spend your engineering time on what matters: **email prioritization intelligence, natural language understanding of user commands, and seamless Outlook/Gmail integration**

The audio buffering problem has been solved. Don't reinvent it—build on top of it and ship faster.

Next Steps

1. **Try the LiveKit Voice Quickstart** (10 minutes): https://docs.livekit.io/agents/start/voice-ai-quickstart/
2. **Explore example agents**: https://github.com/livekit/agents/tree/main/examples
3. **Check out automotive/noise cancellation examples**: Critical for in-car use
4. **Start with LiveKit Cloud free tier**: No infrastructure setup needed

Focus your engineering efforts on the unique value of your email agent—the AI that understands email priority, drafts intelligent responses, and manages inbox rules via voice—not on solving decade-old WebRTC problems.