TECHNICAL GUIDE

...And How We Fixed It

Production Patterns for Groq Orpheus TTS

Companion to: "We Shipped Groq Orpheus TTS to Production - Here's What Broke"

Ayush Jain & Bijoy Roy | Katonic AI Engineering Team
February 2026

SECTION 01

Contents

Overview

The Problem: What Broke

Root Cause Analysis

The Solution: Three Laws of State Management

Backend Implementation Patterns

Frontend Implementation Patterns

Testing Strategies

Production Checklist

Resources

SECTION 02

Overview

This guide provides the complete implementation patterns for deploying Groq Orpheus TTS in production. It is the technical companion to our blog post detailing what went wrong when we first shipped voice AI to production.

What You'll Learn

- WebSocket lifecycle management patterns that prevent resource leaks
- Audio streaming architecture using Web Audio API timeline scheduling
- Idempotent cleanup functions that eliminate race conditions
- Testing strategies that catch production bugs before deployment
- A production-ready checklist for voice AI deployments

Prerequisites

- Groq API access (api.groq.com)
- Familiarity with WebSocket programming
- Basic understanding of Web Audio API
- Node.js backend (examples in JavaScript/TypeScript)

SECTION 03

The Problem: What Broke

When we deployed Groq Orpheus TTS to production, we encountered a pattern of failures that did not reproduce in development:

Issue	Frequency	User Report
Second message silent	~30%	First plays, second is silent
Audio never completes	Intermittent	Agent stops mid-sentence
UI stuck on 'speaking'	~15%	Speaking indicator never stops
STOP doesn't work	Frequent	Clicked stop but audio continued
Memory leaks	Over time	Server monitoring alert

Key Insight

These weren't five separate bugs. They were symptoms of one architectural problem: unclear state ownership.

SECTION 04

Root Cause Analysis

Backend Issues

- Every TTS request created a new WebSocket without closing previous ones
- Event listeners accumulated because old ones were never removed
- Stream completion was inferred rather than explicitly signaled
- STOP commands were treated as advisory rather than authoritative

Frontend Issues

- Complex buffering logic attempted to be 'smart' about audio chunks
- Multiple async code paths competed for control
- Playback timing based on heuristics rather than deterministic scheduling
- UI state depended on assumptions rather than explicit signals

SECTION 05

The Solution: Three Laws of State Management

Law 1: Ownership

Every resource must have exactly one owner at any given time. When ownership transfers, the previous owner must explicitly release the resource.

Law 2: Cleanup

All cleanup functions must be idempotent. It must be safe to call cleanup multiple times without side effects.

Law 3: Completion

Stream completion must be explicit, never inferred. The system must guarantee exactly-once delivery of completion signals.

SECTION 06

Backend Implementation Patterns

Pattern 1: Single WebSocket Per Session

Store the connection reference on the socket itself. When a new TTS request arrives, explicitly close the previous connection before creating a new one.

Pattern 2: Centralized Idempotent Cleanup

Create a single cleanup function that can be called multiple times without side effects. This eliminates race conditions.

Key Implementation Points:

- Store connection reference: socket.data.groqWs = groqWs
- Always cleanup before create: cleanupExternalConnection(socket)
- Remove old listeners: socket.removeAllListeners('stop')
- Track end signal: socket.data.endSignalSent flag
- Verify ownership before emitting: if (socket.data.groqWs === groqWs)

Pattern 3: Authoritative STOP

When the client sends STOP, it must be authoritative - immediately close the WebSocket, terminate the stream, and notify completion. No ambiguity.

SECTION 07

Frontend Implementation Patterns

Pattern 1: Timeline-Driven Audio Playback

Schedule each audio chunk at a precise time on the AudioContext timeline. Never use 'start now' - always calculate the next start time.

Key Implementation Points:

- Track next start time: this.nextStartTime = startTime + duration
- Prevent scheduling in past: Math.max(nextStartTime, currentTime)
- Track active sources: this.activeSources = new Set()
- PCM-native: Convert Int16 to Float32 directly, skip decodeAudioData

Pattern 2: Message-Scoped Audio Sessions

Every new message must reset the audio timeline completely. Clear previous sources and start fresh. This permanently fixes the 'second message silent' bug.

Pattern 3: Explicit Stream Completion

Playback completes only when BOTH conditions are met:

- The end-of-stream signal has been received
- All scheduled audio sources have finished playing

Add a safety timeout in case the browser's onended event doesn't fire.

SECTION 08

Testing Strategies

Standard unit tests are insufficient for real-time systems. Implement these test patterns:

Test 1: Multi-Message Stress Test

Send 100 consecutive messages and verify no resource leaks. Check WebSocket count and memory usage before and after.

Test 2: Rapid Interrupt Test

Issue STOP commands at random points during streaming. Verify immediate cleanup and proper end signal delivery.

Test 3: Second Message Test

Specifically test that message 2 plays correctly after message 1 completes. Both should have audio chunks.

Test 4: Chaos Test

Introduce random latency (50-500ms) and packet loss (10%) to verify graceful degradation. System should remain functional after chaos ends.

SECTION 09

Production Checklist

Requirement	Verification
Single WebSocket per session	Connection count monitoring
Cleanup on new session	Log cleanup calls
Idempotent cleanup	Call cleanup 3x in tests
Exactly-once end signal	Count signals in tests
Authoritative STOP	Verify immediate termination
Timeline-driven playback	No overlap in audio
Message-scoped sessions	Test rapid consecutive messages
Safety timeout	Simulate browser event failures

SECTION 10

Resources

Groq Documentation

- Groq TTS API: console.groq.com/docs/text-to-speech
- Orpheus TTS Guide: console.groq.com/docs/text-to-speech/orpheus

Orpheus Model

- Canopy Labs GitHub: github.com/canopyai/Orpheus-TTS
- Model: canopylabs/orpheus-arabic-saudi
- Voices: Fahad, Sultan, Lulwa, Noura

Web Audio API

- MDN Web Audio API: developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API
- AudioContext scheduling: developer.mozilla.org/en-US/docs/Web/API/AudioBufferSourceNode/start

WebSocket Best Practices

- Ably Guide: ably.com/topic/websocket-architecture-best-practices

Katonic

The enterprise operating system for full-stack AI agents.

Any framework. Any model. Any cloud. Your code.

www.katonic.ai