Vocal Bridge Developer Guide

Everything you need to integrate voice agents into your application.

Overview

Vocal Bridge provides voice AI agents that you can integrate into any application using WebRTC. Your users can have real-time voice conversations with AI agents through web browsers, mobile apps, or any platform that supports WebRTC.

Real-time Voice

Sub-second latency voice AI using WebRTC

Secure API Keys

Production-ready authentication

Multi-platform

JavaScript, Python, React, and more

Quick Start

Get your voice agent working in 3 steps:

1 **Create an API Key**

Go to your agent's page, open Developer Mode, and click "Create API Key" in the API Keys section.

2 **Generate a Token (Server-side)**

Call the token endpoint from your backend to get a LiveKit access token.

```
curl -X POST "http://vocalbridgeai.com/api/v1/token" \
  -H "X-API-Key: YOUR_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{"participant_name": "User"}'
```

Copy

3 Connect from Your Client

Use the LiveKit SDK to connect and enable the microphone.

```
import { Room } from 'livekit-client';

const room = new Room();
await room.connect(livekit_url, token);
await room.localParticipant.setMicrophoneEnabled(true);
```

Copy

Authentication

Vocal Bridge uses API keys for authentication. API keys allow your backend server to generate access tokens without requiring user login.

> ⚠ **Security:** Never expose your API key in client-side code. Always call the token endpoint from your backend server.

API Key Format

API keys start with `vb_` followed by a secure random string:

```
vb_abc123def456...
```

Using API Keys

Include your API key in requests using either method:

```
# Option 1: X-API-Key header (recommended)
curl -H "X-API-Key: vb_your_api_key" http://vocalbridgeai.com/api/v1/token

# Option 2: Authorization header
curl -H "Authorization: Bearer vb_your_api_key" http://vocalbridgeai.com/api/v1/token
```

API Reference

POST `/api/v1/token`

Generate a LiveKit access token for connecting to the agent.

REQUEST HEADERS

`X-API-Key`	Your API key (required)
`Content-Type`	application/json

REQUEST BODY (OPTIONAL)

FIELD	TYPE	DESCRIPTION
`participant_name`	string	Display name for the participant (default: "API Client")
`session_id`	string	Custom session ID (default: auto-generated)

RESPONSE

Copy

```
{
  "livekit_url": "wss://tutor-j7bhwjbm.livekit.cloud",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "room_name": "user-abc-agent-xyz-api-12345",
  "participant_identity": "api-client-xxxx-12345",
  "expires_in": 3600,
  "agent_mode": "cascaded_concierge"
}
```

GET `/api/v1/agent`

Get information about the agent associated with your API key.

RESPONSE

Copy

```
{
  "id": "uuid",
  "name": "My Voice Agent",
  "mode": "cascaded_concierge",
  "deployment_status": "active",
  "phone_number": "+1234567890",
  "greeting": "Hello! How can I help you?",
  "background_enabled": true,
  "hold_enabled": false,
  "hangup_enabled": false,
  "created_at": "2025-01-14T12:00:00Z"
}
```

JavaScript SDK

Use the LiveKit JavaScript SDK to connect from web browsers.

Installation

Copy

```
npm install livekit-client
```

Complete Example

Copy

```
import { Room, RoomEvent, Track } from 'livekit-client';

class VoiceAgent {
  constructor() {
    this.room = new Room();
    this.setupEventHandlers();
  }

  setupEventHandlers() {
    // Handle incoming audio from the agent
    this.room.on(RoomEvent.TrackSubscribed, (track, publication, participant) => {
      if (track.kind === Track.Kind.Audio) {
        const audioElement = track.attach();
        document.body.appendChild(audioElement);
        console.log('Agent audio connected');
      }
    });

    // Handle connection state changes
    this.room.on(RoomEvent.ConnectionStateChanged, (state) => {
      console.log('Connection state:', state);
    });
    // Handle disconnection
    this.room.on(RoomEvent.Disconnected, () => {
      console.log('Disconnected from room');
    });
  }

  async connect() {
    // Get token from your backend
    const response = await fetch('/api/voice-token');
    const { livekit_url, token } = await response.json();

    // Connect to the room
    await this.room.connect(livekit_url, token);
    console.log('Connected to room:', this.room.name);

    // Enable microphone
    await this.room.localParticipant.setMicrophoneEnabled(true);
    console.log('Microphone enabled - start speaking!');
  }

  async disconnect() {
    await this.room.disconnect();
  }
}

// Usage
const agent = new VoiceAgent();
await agent.connect();
```

Backend Token Endpoint (Node.js/Express)

Copy

```
// server.js
const express = require('express');
const app = express();

const VOCAL_BRIDGE_API_KEY = process.env.VOCAL_BRIDGE_API_KEY;
const VOCAL_BRIDGE_URL = 'http://vocalbridgeai.com';

app.get('/api/voice-token', async (req, res) => {
  try {
    const response = await fetch(`${VOCAL_BRIDGE_URL}/api/v1/token`, {
      method: 'POST',
      headers: {
        'X-API-Key': VOCAL_BRIDGE_API_KEY,
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        participant_name: req.user?.name || 'User'
      })
    });

    const data = await response.json();
    res.json(data);
  } catch (error) {
    console.error('Failed to get token:', error);
    res.status(500).json({ error: 'Failed to get voice token' });
  }
});

app.listen(3000);
```

Python SDK

Use the LiveKit Python SDK for server-side or desktop applications.

Installation

```
pip install livekit requests
```

Copy

Complete Example

Copy

```
import asyncio
import os
import requests
from livekit import rtc

VOCAL_BRIDGE_API_KEY = os.environ.get('VOCAL_BRIDGE_API_KEY')
VOCAL_BRIDGE_URL = 'http://vocalbridgeai.com'


def get_voice_token(participant_name: str = 'Python Client'):
    """Get a voice token from Vocal Bridge API."""
    response = requests.post(
        f'{VOCAL_BRIDGE_URL}/api/v1/token',
        headers={
            'X-API-Key': VOCAL_BRIDGE_API_KEY,
            'Content-Type': 'application/json'
        },
        json={'participant_name': participant_name}
    )
    response.raise_for_status()
    return response.json()


async def main():
```

```python
# Get token
token_data = get_voice_token()
print(f"Connecting to room: {token_data['room_name']}")

# Create room
room = rtc.Room()

# Set up event handlers
@room.on("track_subscribed")
def on_track_subscribed(track, publication, participant):
    if track.kind == rtc.TrackKind.KIND_AUDIO:
        print("Agent audio connected!")
        # Process audio stream
        audio_stream = rtc.AudioStream(track)
        # ... handle audio frames

@room.on("disconnected")
def on_disconnected():
    print("Disconnected from room")

# Connect
await room.connect(token_data['livekit_url'], token_data['token'])
print(f"Connected! Room: {room.name}")

# Publish microphone (requires audio input device)
source = rtc.AudioSource(sample_rate=48000, num_channels=1)
track = rtc.LocalAudioTrack.create_audio_track("microphone", source)
await room.local_participant.publish_track(track)
print("Microphone enabled - start speaking!")

# Keep running
try:
    while True:
        await asyncio.sleep(1)
except KeyboardInterrupt:
        await room.disconnect()


if __name__ == '__main__':
    asyncio.run(main())
```

Flask Backend Example

Copy

```
# app.py
from flask import Flask, jsonify
import requests
import os

app = Flask(__name__)

VOCAL_BRIDGE_API_KEY = os.environ.get('VOCAL_BRIDGE_API_KEY')
VOCAL_BRIDGE_URL = 'http://vocalbridgeai.com'


@app.route('/api/voice-token')
def get_voice_token():
    response = requests.post(
        f'{VOCAL_BRIDGE_URL}/api/v1/token',
        headers={
            'X-API-Key': VOCAL_BRIDGE_API_KEY,
            'Content-Type': 'application/json'
        },
        json={'participant_name': 'Web User'}
    )
    return jsonify(response.json())
```

```
if __name__ == '__main__':
    app.run(port=5000)
```

React Integration

Use the LiveKit React Components for easy integration with React applications.

Use the same backend token endpoint from the JavaScript SDK section to get LiveKit tokens.

Installation

Copy

```
npm install @livekit/components-react livekit-client
```

React Hook Example

Copy

```
// useVoiceAgent.ts
import { useState, useCallback, useEffect } from 'react';
import { Room, RoomEvent, Track } from 'livekit-client';

interface VoiceAgentState {
  isConnected: boolean;
  isConnecting: boolean;
  isMicEnabled: boolean;
  error: string | null;
}

export function useVoiceAgent() {
  const [room] = useState(() => new Room());
  const [state, setState] = useState<VoiceAgentState>({
    isConnected: false,
    isConnecting: false,
    isMicEnabled: false,
    error: null
  });

  useEffect(() => {
    // Handle agent audio
    const handleTrackSubscribed = (track: any) => {
      if (track.kind === Track.Kind.Audio) {
        const audioEl = track.attach();
        document.body.appendChild(audioEl);
      }
    };

    const handleDisconnected = () => {
      setState(s => ({ ...s, isConnected: false, isMicEnabled: false }));
    };

    room.on(RoomEvent.TrackSubscribed, handleTrackSubscribed);
    room.on(RoomEvent.Disconnected, handleDisconnected);

    return () => {
      room.off(RoomEvent.TrackSubscribed, handleTrackSubscribed);
      room.off(RoomEvent.Disconnected, handleDisconnected);
      room.disconnect();
    };
  }, [room]);

  const connect = useCallback(async () => {
    setState(s => ({ ...s, isConnecting: true, error: null }));

    try {
      // Get token from your backend
      const res = await fetch('/api/voice-token');
      const { livekit_url, token } = await res.json();
```

```
      await room.connect(livekit_url, token);
      await room.localParticipant.setMicrophoneEnabled(true);

      setState(s => ({
        ...s,
        isConnected: true,
        isConnecting: false,
        isMicEnabled: true
      }));
    } catch (err) {
      setState(s => ({
        ...s,
        isConnecting: false,
        error: err instanceof Error ? err.message : 'Connection failed'
      }));
    }
  }, [room]);

  const disconnect = useCallback(async () => {
    await room.disconnect();
  }, [room]);

  const toggleMic = useCallback(async () => {
    const enabled = !state.isMicEnabled;
    await room.localParticipant.setMicrophoneEnabled(enabled);
    setState(s => ({ ...s, isMicEnabled: enabled }));
  }, [room, state.isMicEnabled]);

  return { ...state, connect, disconnect, toggleMic };
}

// VoiceAgentButton.tsx
import { useVoiceAgent } from './useVoiceAgent';

export function VoiceAgentButton() {
```

```
const { isConnected, isConnecting, isMicEnabled, error, connect, disconnect, toggleMic } = useVoiceAgent();

if (error) {
  return <div className="text-red-500">Error: {error}</div>;
}

if (!isConnected) {
  return (
    <button
      onClick={connect}
      disabled={isConnecting}
      className="px-4 py-2 bg-indigo-600 text-white rounded-lg"
    >
      {isConnecting ? 'Connecting...' : 'Start Voice Chat'}
    </button>
  );
}

return (
  <div className="flex gap-2">
    <button
      onClick={toggleMic}
      className={`px-4 py-2 rounded-lg ${isMicEnabled ? 'bg-green-600' : 'bg-gray-600'} text-white`}
    >
      {isMicEnabled ? 'Mute' : 'Unmute'}
    </button>
    <button
      onClick={disconnect}
      className="px-4 py-2 bg-red-600 text-white rounded-lg"
    >
      End Call
    </button>
  </div>
  );
}
```

Handling Client Actions (React)

Copy

```
// Add to useVoiceAgent hook for bidirectional communication

import { RoomEvent } from 'livekit-client';

// Inside useVoiceAgent hook:

// Handle actions FROM the agent (Agent to App)
useEffect(() => {
  const handleData = (
    payload: Uint8Array,
    participant: any,
    kind: any,
    topic?: string
  ) => {
    if (topic === 'client_actions') {
      const data = JSON.parse(new TextDecoder().decode(payload));
      if (data.type === 'client_action') {
        handleAgentAction(data.action, data.payload);
      }
    }
  };

  room.on(RoomEvent.DataReceived, handleData);
  return () => { room.off(RoomEvent.DataReceived, handleData); };
}, [room]);

function handleAgentAction(action: string, payload: any) {
  switch (action) {
    case 'navigate':
      // Navigate to a route
      window.location.href = payload.url;
      break;
    case 'show_product':
      // Show a product modal
      setProductId(payload.productId);
      break;
    default:
      console.log('Unknown action:', action, payload);
  }
}

// Send actions TO the agent (App to Agent)
const sendActionToAgent = useCallback(async (
  action: string,
  payload: Record<string, any> = {}
) => {
  const message = JSON.stringify({
    type: 'client_action',
    action,
    payload
  });
  await room.localParticipant.publishData(
    new TextEncoder().encode(message),
    { reliable: true, topic: 'client_actions' }
  );
}, [room]);

// Example usage in component:
// <button onClick={() => sendActionToAgent('button_clicked', { buttonId: 'buy' })}>
//   Buy Now
// </button>
```

Next.js API Route Example

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

Copy

```
// app/api/voice-token/route.ts (Next.js App Router)
import { NextResponse } from 'next/server';

const VOCAL_BRIDGE_API_KEY = process.env.VOCAL_BRIDGE_API_KEY!;
const VOCAL_BRIDGE_URL = 'http://vocalbridgeai.com';

export async function GET() {
  try {
    const response = await fetch(`${VOCAL_BRIDGE_URL}/api/v1/token`, {
      method: 'POST',
      headers: {
        'X-API-Key': VOCAL_BRIDGE_API_KEY,
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        participant_name: 'Web User',
      }),
    });

    if (!response.ok) {
      throw new Error('Failed to get token');
    }

    const data = await response.json();
    return NextResponse.json(data);
  } catch (error) {
    return NextResponse.json(
      { error: 'Failed to get voice token' },
      { status: 500 }
    );
  }
}
```

Flutter SDK

Use the LiveKit Flutter SDK to build voice-enabled mobile apps for iOS and Android.

Installation

Add to your `pubspec.yaml`:

Copy

```yaml
dependencies:
  livekit_client: ^2.3.0
  http: ^1.2.0
```

Complete Example

Copy

```dart
import 'dart:convert';
import 'package:http/http.dart' as http;
import 'package:livekit_client/livekit_client.dart';

class VoiceAgentService {
  Room? _room;
  EventsListener<RoomEvent>? _listener;

  // Get token from your backend (recommended for production)
  // Your backend should call Vocal Bridge API with your API key
  Future<Map<String, dynamic>> _getTokenFromBackend() async {
    final response = await http.get(
      Uri.parse('https://your-backend.com/api/voice-token'),
    );
    return jsonDecode(response.body);
  }

  // Alternative: Call Vocal Bridge API directly (for testing/prototyping only)
  // WARNING: Never expose API keys in production mobile apps!
  Future<Map<String, dynamic>> _getTokenDirect(String apiKey) async {
    final response = await http.post(
      Uri.parse('http://vocalbridgeai.com/api/v1/token'),
      headers: {
        'X-API-Key': apiKey,
        'Content-Type': 'application/json',
      },
      body: jsonEncode({'participant_name': 'Mobile User'}),
    );
    return jsonDecode(response.body);
  }

  // Connect to the voice agent
  Future<void> connect() async {
    // Use _getTokenFromBackend() in production
    final tokenData = await _getTokenFromBackend();
    final livekitUrl = tokenData['livekit_url'] as String;
    final token = tokenData['token'] as String;

    _room = Room();

    // Listen for agent audio
    _listener = _room!.createListener();
    _listener!.on<TrackSubscribedEvent>((event) {
      if (event.track.kind == TrackType.AUDIO) {
        // Audio is automatically played by LiveKit SDK
        print('Agent audio track subscribed');
      }
    });
```

```dart
    // Handle connection state
    _listener!.on<RoomDisconnectedEvent>((event) {
      print('Disconnected from room');
    });

    // Connect to the room
    await _room!.connect(livekitUrl, token);
    print('Connected to room: ${_room!.name}');

    // Enable microphone
    await _room!.localParticipant?.setMicrophoneEnabled(true);
    print('Microphone enabled - start speaking!');
  }

  // Disconnect from the agent
  Future<void> disconnect() async {
    await _room?.disconnect();
    _room = null;
    _listener = null;
  }

  // Check if connected
  bool get isConnected => _room?.connectionState == ConnectionState.connected;
}

// Usage in a Flutter widget
class VoiceAgentButton extends StatefulWidget {
  @override
  _VoiceAgentButtonState createState() => _VoiceAgentButtonState();
}

class _VoiceAgentButtonState extends State<VoiceAgentButton> {
  final _voiceAgent = VoiceAgentService();
  bool _isConnecting = false;
  Future<void> _toggleConnection() async {
    setState(() => _isConnecting = true);
    try {
      if (_voiceAgent.isConnected) {
        await _voiceAgent.disconnect();
      } else {
        await _voiceAgent.connect();
      }
    } finally {
      setState(() => _isConnecting = false);
    }
  }

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: _isConnecting ? null : _toggleConnection,
      child: Text(_isConnecting
          ? 'Connecting...'
          : _voiceAgent.isConnected
              ? 'End Call'
              : 'Start Voice Chat'),
    );
  }
}
```

Handling Client Actions (Flutter)

Copy

```
// Add to VoiceAgentService class

// Handle actions from the agent
void _setupClientActionHandler() {
  _listener!.on<DataReceivedEvent>((event) {
    if (event.topic == 'client_actions') {
      final data = jsonDecode(utf8.decode(event.data));
      if (data['type'] == 'client_action') {
        _handleAgentAction(data['action'], data['payload']);
      }
    }
  });
}

void _handleAgentAction(String action, Map<String, dynamic> payload) {
  switch (action) {
    case 'navigate':
      // Navigate to a screen
      print('Navigate to: ${payload['screen']}');
      break;
    case 'show_product':
      // Show a product card
      print('Show product: ${payload['productId']}');
      break;
    default:
      print('Unknown action: $action');
  }
}

// Send actions to the agent
Future<void> sendActionToAgent(String action, [Map<String, dynamic>? payload]) async {
  final message = jsonEncode({
    'type': 'client_action',
    'action': action,
    'payload': payload ?? {},
  });
  await _room?.localParticipant?.publishData(
    utf8.encode(message),
    reliable: true,
    topic: 'client_actions',
  );
}

// Example: Notify agent that user tapped a button
// await sendActionToAgent('button_tapped', {'buttonId': 'buy_now'});
```

Platform Setup

iOS Configuration

Add to `ios/Runner/Info.plist`:

```
<key>NSMicrophoneUsageDescription</key>
<string>This app needs microphone access for voice chat</string>
<key>UIBackgroundModes</key>
<array>
  <string>audio</string>
</array>
```

Android Configuration

Add to `android/app/src/main/AndroidManifest.xml`:

```
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS"/>
```

Client Actions

Client Actions enable bidirectional communication between your voice agent and your client application via LiveKit's data channel.

Directions

- **Agent to App**: The agent triggers actions in your client (e.g., navigate to a page, show a product card, update the UI).
- **App to Agent**: Your client sends events to the agent (e.g., user clicked a button, form submitted, data loaded).

Behavior (App to Agent)

Each **app_to_agent** action has a **behavior** that controls how the agent handles the inbound event:

- **respond** (default): The agent generates a reply when this event arrives. Use for events that require the agent to speak.
- **notify**: The event is silently added to conversation context. The agent sees it on its next turn but does *not* reply immediately. Use for informational updates.

This prevents feedback loops where an agent action triggers a client event which triggers another agent reply, and so on.

How It Works

Agent to App (Outbound)

1. Agent decides to trigger an action during conversation
2. Agent calls `trigger_client_action`
3. Action is published to LiveKit data channel
4. Your app receives and handles it

App to Agent (Inbound)

1. Your app publishes data to LiveKit
2. Agent receives and validates the event
3. Behavior is checked: **respond** or **notify**
4. Agent replies or silently absorbs the event

Receiving Actions from Agent (JavaScript)

Copy

```
import { Room, RoomEvent } from 'livekit-client';

const room = new Room();

// Listen for actions FROM the agent (Agent to App)
room.on(RoomEvent.DataReceived, (payload, participant, kind, topic) => {
  if (topic === 'client_actions') {
    const data = JSON.parse(new TextDecoder().decode(payload));
    if (data.type === 'client_action') {
      handleAgentAction(data.action, data.payload);
    }
  }
});

function handleAgentAction(action, payload) {
  switch (action) {
    case 'navigate':
      window.location.href = payload.url;
      break;
    case 'show_product':
      showProductModal(payload.product_id);
      break;
    default:
      console.log('Unknown action:', action, payload);
  }
}
```

Sending Actions to Agent (JavaScript)

Copy

```
// Send actions TO the agent (App to Agent)
// Behavior is configured on the agent side per action:
//   respond = agent replies immediately
//   notify  = silent context only (no reply)
function sendActionToAgent(action, payload = {}) {
  const message = JSON.stringify({
    type: 'client_action',
    action: action,
    payload: payload
  });
```

```
    room.localParticipant.publishData(
      new TextEncoder().encode(message),
      { reliable: true, topic: 'client_actions' }
    );
  }

  // Example: User clicked buy (behavior: respond — agent will reply)
  sendActionToAgent('user_clicked_buy', { productId: '123', quantity: 2 });

  // Example: Practice result (behavior: notify — agent absorbs silently)
  sendActionToAgent('practice_result', { score: 95, word: 'hello' });
```

Example Configuration

When configuring your agent, you can add client actions like:

ACTION NAME	DIRECTION	BEHAVIOR	DESCRIPTION
`show_product`	agent_to_app	—	Display product details in the app
`user_clicked_buy`	app_to_agent	respond	User clicked the buy button
`practice_result`	app_to_agent	notify	User completed a practice exercise

MCP Tools

The Model Context Protocol (MCP) allows your voice agent to connect to external tools and services. By providing an MCP server URL, your agent gains access to calendars, email, CRM systems, databases, and thousands of other integrations.

Quick Setup with Zapier
The easiest way to add tools is through Zapier MCP. Connect 7,000+ apps to your voice agent in minutes.

How MCP Works

1. Obtain an MCP server URL from Zapier or your own MCP server
2. Add the URL in your agent's configuration
3. The agent automatically discovers and loads available tools
4. During conversations, the agent can call these tools to fetch data or perform actions

Example Use Cases

Calendar Integration

Check availability, book appointments, send meeting invites via Google Calendar or Outlook

CRM Access

Look up customer info, create leads, update contact records in Salesforce, HubSpot, etc.

Email & Messaging

Send emails, Slack messages, or SMS notifications during or after calls

Database Queries

Query product catalogs, inventory, order status, or any custom database

Getting an MCP Server URL

Option 1: Zapier MCP (Recommended)

1. Go to zapier.com/mcp
2. Sign in and configure the apps you want to connect

3. Copy your MCP server URL (format: `https://actions.zapier.com/mcp/...`)
4. Paste into your agent's MCP Server URL field

Option 2: Custom MCP Server

Build your own MCP server using the MCP specification. Your server must support the Streamable HTTP transport.

Viewing Available Tools

After adding an MCP server URL to your agent, the available tools will be displayed in the agent's configuration page. The agent will automatically use these tools when relevant during conversations.

Post-Processing

Post-processing runs automatically after each call ends. Use it to summarize conversations, update CRM records, send follow-up emails, create tickets, or trigger any workflow based on what happened during the call.

⚠ **Automatic Execution**
Post-processing runs in the background after every call. No user action required. The transcript and call metadata are automatically available.

How It Works

1. Call ends (user hangs up or agent ends the call)
2. Full conversation transcript is captured
3. Post-processing LLM analyzes the transcript using your custom prompt
4. If MCP tools are configured, the LLM can call them to perform actions
5. Results are logged for review

Configuration Options

Post-Processing Prompt

Tell the LLM what to do with the conversation transcript. Be specific about the output format and actions to take.

```
Example: "Analyze this call transcript and: 1) Create a brief summary (2-3 sentences), 2) Extract
any action items mentioned, 3) Identify the caller's sentiment (positive/neutral/negative), 4) If
the caller requested a callback, create a task in the CRM."
```

Post-Processing MCP Server

Optionally connect a separate MCP server specifically for post-processing tasks. This allows the post-processing LLM to update CRMs, send emails, create tickets, or trigger any automation after the call.

Example Use Cases

Call Summaries

Generate structured summaries with key points, decisions, and next steps

CRM Updates

Automatically log call notes, update lead status, or create follow-up tasks

Follow-up Emails

Send personalized follow-up emails based on the conversation content

Escalation Alerts

Detect urgent issues and notify team members via Slack, email, or SMS

Available Context

The post-processing LLM has access to:

- **Full transcript** - Complete conversation with speaker labels
- **Call duration** - How long the call lasted
- **Timestamp** - When the call started and ended
- **Agent configuration** - The agent's system prompt and settings
- **MCP tools** - Any tools configured for post-processing

Troubleshooting

Connection fails with "403 Forbidden"

Your API key may be invalid or revoked. Check that you're using the correct API key and that it hasn't been revoked in the dashboard.

No audio from the agent

Make sure you're attaching the audio track to an audio element when it's subscribed. Also check that the audio element is not muted and that the browser has autoplay permissions.

```
room.on(RoomEvent.TrackSubscribed, (track) => {
  if (track.kind === 'audio') {
    const audioEl = track.attach();
    audioEl.play(); // May need user gesture first
    document.body.appendChild(audioEl);
  }
});
```

Microphone not working

The browser may not have microphone permissions. Request permission before calling `setMicrophoneEnabled(true)` :

```
// Request microphone permission first
await navigator.mediaDevices.getUserMedia({ audio: true });

// Then enable in LiveKit
await room.localParticipant.setMicrophoneEnabled(true);
```

Token expired

Tokens are valid for 1 hour. If you get a token expiration error, request a new token from your backend and reconnect.

CORS errors

Don't call the Vocal Bridge API directly from the browser. Instead, make requests from your backend server to avoid CORS issues and keep your API key secure.

Claude Code Plugin

The Vocal Bridge plugin for Claude Code lets you manage your voice agents directly from the command line. View call logs, update prompts, stream debug events, and iterate on your agent without leaving your terminal.

Works with Claude Code

Install the plugin in Claude Code to get native slash commands for managing your voice agent. Claude can automatically use these commands when you ask about your agent.

Installation

Install the plugin from the Vocal Bridge marketplace:

```
/plugin marketplace add vocalbridgeai/vocal-bridge-marketplace
/plugin install vocal-bridge@vocal-bridge
```

Copy

Getting Started

After installing, authenticate with your API key:

```
/vocal-bridge:login vb_your_api_key_here
```

Copy

Get your API key from your agent's detail page in the dashboard.

Available Commands

COMMAND	DESCRIPTION
`/vocal-bridge:login`	Authenticate with your API key
`/vocal-bridge:status`	Check authentication status
`/vocal-bridge:agent`	Show agent information (name, mode, phone number)
`/vocal-bridge:logs`	View call logs and transcripts
`/vocal-bridge:stats`	Show call statistics
`/vocal-bridge:prompt`	View or update system prompt
`/vocal-bridge:debug`	Stream real-time debug events
`/vocal-bridge:help`	Show all available commands

Example Workflow

Copy

```
# Check recent calls
/vocal-bridge:logs

# View a specific call transcript
/vocal-bridge:logs 550e8400-e29b-41d4-a716-446655440000

# Find failed calls
/vocal-bridge:logs --status failed

# Check statistics
/vocal-bridge:stats

# View current prompt
/vocal-bridge:prompt show

# Stream debug events while testing
/vocal-bridge:debug
```

Benefits

Stay in Flow

No context switching between terminal and browser

AI-Assisted

Claude can use commands automatically when you ask about your agent

Real-time Debug

Stream live events while making test calls

Quick Iteration

Update prompts and test changes rapidly

Links

- Plugin Repository - Source code and documentation
- Marketplace Repository - Plugin registry
- CLI on PyPI - Standalone CLI (also usable outside Claude Code)

Need more help? Contact support@vocalbridgeai.com