Custom Text-to-Speech (TTS) Integration with VAPI Assistant

Table of Contents

What This Guide Covers

This guide walks you through integrating your own Text-to-Speech (TTS) system with VAPI Assistant. Whether you're looking to use a specific voice model, support additional languages, or reduce costs, custom TTS integration gives you complete control over how your assistant sounds.

Why use custom TTS?

- **Brand-specific voices** that match your company's identity
- **Better audio quality** from advanced TTS models
- **Language coverage** for dialects and accents not available elsewhere
- **Cost control** using your preferred TTS provider
- **Voice personalization** for different user groups

This integration maintains VAPI's real-time performance while giving you complete flexibility over voice synthesis.

How VAPI Custom TTS Works

VAPI's custom TTS system operates through a simple webhook pattern:

1. **During a conversation**, VAPI needs to convert text to speech
2. **VAPI sends a POST request** to your TTS endpoint with the text and audio specifications
3. **Your system generates audio** and streams it back as raw PCM data
4. **VAPI plays the audio** to the caller in real-time

What You'll Need

- **A web server** that can receive POST requests and return audio data
- **Your TTS system** (could be a cloud API, local model, or custom solution)
- **VAPI account** with access to custom voice configuration

The entire system is designed to work seamlessly with your existing TTS infrastructure while maintaining the low-latency requirements of real-time voice conversations.

Setting Up Authentication

VAPI needs to securely communicate with your TTS endpoint. You have a couple of authentication options:

Option 1: Secret Header (Most Common)

VAPI includes a secret token in the `X-VAPI-SECRET` header with every request:

```
{
  "voice": {
    "provider": "custom-voice",
    "server": {
      "url": "https://your-tts-api.com/synthesize",
      "secret": "your-secret-token-here",
      "timeoutSeconds": 30
    }
  }
}
```

Option 2: Additional Headers

You can include extra headers for enhanced security or API versioning:

```
{
  "voice": {
    "provider": "custom-voice",
    "server": {
      "url": "https://your-tts-api.com/synthesize",
      "secret": "your-secret-token",
      "headers": {
        "X-API-Version": "v1",
        "X-Client-ID": "vapi-integration"
      }
    }
  }
}
```

Enterprise Note: OAuth2 authentication is also available through webhook credentials for larger deployments.

Building Your TTS Integration

Step 1: Configure Your VAPI Assistant

First, set up your assistant to use your custom TTS endpoint:

```
{
  "name": "My Custom Voice Assistant",
  "voice": {
    "provider": "custom-voice",
    "server": {
      "url": "https://your-tts-endpoint.com/api/synthesize",
      "secret": "your-webhook-secret",
      "timeoutSeconds": 45,
      "headers": {
        "Content-Type": "application/json",
        "X-API-Version": "v1"
      }
    },
    "fallbackPlan": {
      "voices": [
        {
          "provider": "eleven-labs",
          "voiceId": "21m00Tcm4TlvDq8ikWAM"
        }
      ]
    }
  }
}
```

Step 2: Build Your TTS Server

Here's a complete implementation that handles everything you need:

```
const express = require('express');
const crypto = require('crypto');

const app = express();
app.use(express.json({ limit: '50mb' }));

// Main TTS endpoint
app.post('/api/synthesize', async (req, res) => {
  const requestId = crypto.randomUUID();
  const startTime = Date.now();

  // Set up timeout protection
  const timeout = setTimeout(() => {
    if (!res.headersSent) {
```

```
      res.status(408).json({ error: 'Request timeout' });
    }
  }, 30000);

  try {
    console.log(`TTS request started: ${requestId}`);

    // Extract and validate the request
    const { message } = req.body;
    if (!message) {
      clearTimeout(timeout);
      return res.status(400).json({ error: 'Missing message object' });
    }

    const { type, text, sampleRate } = message;

    // Make sure this is a voice request
    if (type !== 'voice-request') {
      clearTimeout(timeout);
      return res.status(400).json({ error: 'Invalid message type' });
    }

    // Validate the text content
    if (!text || typeof text !== 'string') {
      clearTimeout(timeout);
      return res.status(400).json({ error: 'Invalid or missing text' });
    }

    if (!sampleRate || typeof sampleRate !== 'number') {
      clearTimeout(timeout);
      return res.status(400).json({ error: 'Invalid or missing sampleRate'
});
    }

    // Check for empty text
    if (text.trim().length === 0) {
      clearTimeout(timeout);
      return res.status(400).json({ error: 'Empty text provided' });
    }

    // Validate sample rate
    const validSampleRates = [8000, 16000, 22050, 24000, 44100];
    if (!validSampleRates.includes(sampleRate)) {
      clearTimeout(timeout);
      return res.status(400).json({
        error: 'Unsupported sample rate',
        supportedRates: validSampleRates,
      });
    }
```

```
    console.log(
      `TTS synthesis starting: ${requestId}, text_length=${text.length},
sample_rate=${sampleRate}`,
    );

    // Generate the audio (replace with your TTS implementation)
    const audioBuffer = await synthesizeAudio(text, sampleRate);

    // Make sure we got audio back
    if (!audioBuffer || audioBuffer.length === 0) {
      throw new Error('TTS synthesis produced no audio');
    }

    clearTimeout(timeout);

    // Send the audio back to VAPI
    res.setHeader('Content-Type', 'application/octet-stream');
    res.setHeader('Content-Length', audioBuffer.length);
    res.write(audioBuffer);
    res.end();

    const duration = Date.now() - startTime;
    console.log(
      `TTS request completed: ${requestId}, duration=${duration}ms,
audio_bytes=${audioBuffer.length}`,
    );
  } catch (error) {
    clearTimeout(timeout);
    const duration = Date.now() - startTime;
    console.error(
      `TTS request failed: ${requestId}, duration=${duration}ms,
error=${error.message}`,
    );

    if (!res.headersSent) {
      res.status(500).json({
        error: 'TTS synthesis failed',
        requestId,
      });
    }
  }
});

// Replace this with your actual TTS implementation
async function synthesizeAudio(text, sampleRate) {
  // This is just a demo - replace with your TTS system
  await new Promise((resolve) => setTimeout(resolve, 100));
```

```
  // Generate test audio (sine wave)
  const duration = Math.min(text.length * 0.1, 10); // Max 10 seconds
  const samples = Math.floor(duration * sampleRate);
  const buffer = Buffer.alloc(samples * 2); // 16-bit = 2 bytes per sample

  for (let i = 0; i < samples; i++) {
    const value = Math.sin((2 * Math.PI * 440 * i) / sampleRate) * 16000;
    buffer.writeInt16LE(Math.round(value), i * 2);
  }

  return buffer;
}

// Error handling
app.use((error, req, res, next) => {
  console.error('Unhandled error:', error.message);
  res.status(500).json({ error: 'Internal server error' });
});

// Start the server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`TTS server listening on port ${PORT}`);
});

module.exports = app;
```

Step 3: Handle Edge Cases and Limitations

Text Processing Considerations

Audio Format: VAPI only accepts raw PCM audio - no MP3, WAV, or other formats. Make sure your TTS system outputs the correct format.

Sample Rate Matching: The audio you generate must exactly match the requested sample rate, or playback will be distorted.

Handling Special Content

Text Cleaning:

```
function preprocessText(text) {
  // Handle SSML tags if your TTS supports them
  if (text.includes('<speak>')) {
    return parseSSML(text);
  }

  // Clean up problematic characters
  return text
    .replace(/[^\w\s\.,!?-]/g, '') // Remove special characters
```

```
    .replace(/\s+/g, ' ') // Normalize whitespace
    .trim();
}
```

Empty Text Handling:

```
function validateText(text) {
  if (!text || text.trim().length === 0) {
    throw new Error('Empty text provided');
  }

  if (!/[a-zA-Z]/.test(text)) {
    throw new Error('No readable text found');
  }

  return text.trim();
}
```

Step 4: Test Your Integration

The best way to test your TTS integration is by creating actual VAPI calls. Here's how:

Create a Test Call Through VAPI

Use VAPI's API to create a test call that will exercise your TTS system:

```
// Create a test call using VAPI's API
async function testTTSWithVAPICall() {
  const vapiApiKey = 'your-vapi-api-key';
  const assistantId = 'your-assistant-id'; // Assistant configured with
custom TTS

  const callData = {
    assistant: {
      id: assistantId,
    },
    phoneNumberId: 'your-phone-number-id',
    customer: {
      number: '+1234567890', // Your test phone number
    },
  };

  try {
    const response = await fetch('https://api.vapi.ai/call', {
      method: 'POST',
      headers: {
        Authorization: `Bearer ${vapiApiKey}`,
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(callData),
    });
```

```
    const call = await response.json();
    console.log('Test call created:', call.id);

    // Monitor the call
    return call;
  } catch (error) {
    console.error('Failed to create test call:', error);
  }
}
```

Monitor TTS Requests

Set up logging to see exactly what VAPI is sending to your endpoint:

```
app.use('/api/synthesize', (req, res, next) => {
  console.log('=== Incoming TTS Request ===');
  console.log('Headers:', req.headers);
  console.log('Body:', JSON.stringify(req.body, null, 2));
  console.log('============================');
  next();
});
```

Quick Endpoint Test

For initial testing, you can also test your endpoint directly:

```
async function quickEndpointTest() {
  const testPayload = {
    message: {
      type: 'voice-request',
      text: 'Hello, this is a test of the custom TTS system.',
      sampleRate: 24000,
      timestamp: Date.now(),
    },
  };

  try {
    const response = await fetch('http://localhost:3000/api/synthesize', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        'X-VAPI-SECRET': 'your-test-secret',
      },
      body: JSON.stringify(testPayload),
    });

    if (response.ok) {
      const audioBuffer = await response.buffer();
      console.log(`Audio generated: ${audioBuffer.length} bytes`);
      // Save for manual verification
```

```
      require('fs').writeFileSync('test-output.pcm', audioBuffer);
    } else {
      console.error('Test failed:', await response.text());
    }
  } catch (error) {
    console.error('Test error:', error);
  }
}
```

Understanding Request and Response Formats

What VAPI Sends You

Every TTS request from VAPI follows this structure:

```
{
  "message": {
    "type": "voice-request",
    "text": "Hello, world! How can I help you today?",
    "sampleRate": 24000,
    "timestamp": 1677123456789,
    "call": {
      "id": "call-123",
      "orgId": "org-456"
    },
    "assistant": {
      "id": "assistant-789",
      "name": "Customer Service Bot"
    },
    "customer": {
      "number": "+1234567890"
    }
  }
}
```

Required Fields

Field	Type	Description
`type`	string	Always `"voice-request"`
`text`	string	Text to synthesize
`sampleRate`	number	Target audio sample rate (typically 8000, 16000, 22050, or 24000 Hz)

Field	Type	Description
`timestamp`	number	Unix timestamp in milliseconds

Optional Context Information

Field	Type	Description
`call`	object	Call information including ID and organization
`assistant`	object	Assistant configuration and metadata
`customer`	object	Customer information
`artifact`	object	Additional call context data

What You Need to Return

Your endpoint must respond with:

1. **HTTP 200 status**
2. **Content-Type**: `application/octet-stream`
3. **Raw PCM audio data** in the response body

Correct Response Headers:

```
HTTP/1.1 200 OK
Content-Type: application/octet-stream
Transfer-Encoding: chunked
```

Audio Requirements

PCM Audio Specifications

Your TTS system must generate audio with these exact specifications:

- **Format**: Raw PCM (no headers or containers)
- **Channels**: 1 (mono only)
- **Bit Depth**: 16-bit signed integer
- **Byte Order**: Little-endian
- **Sample Rate**: Must exactly match the `sampleRate` in the request

Why These Requirements Matter

VAPI streams audio in real-time during phone calls. Any deviation from these specifications will cause audio distortion, playback failures, or call quality issues.

Common Issues and Troubleshooting

Request Timeouts

What you'll see: VAPI doesn't receive your audio response, calls may drop **Common causes**:

- Your TTS processing takes longer than the configured timeout
- Network connectivity problems between VAPI and your server
- Your server is overloaded or unresponsive

Audio Playback Problems

What you'll see: No audio during calls, or distorted/garbled sound **Common causes**:

- Wrong audio format (not raw PCM)
- Incorrect sample rate
- Sending stereo audio instead of mono
- Including audio file headers in your response

Authentication Failures

What you'll see: 401 Unauthorized responses in your server logs **Common causes**:

- Secret token mismatch between VAPI configuration and your server
- Missing `X-VAPI-SECRET` header validation
- Case sensitivity issues with header names

High Latency or Slow Response

What you'll see: Noticeable delays during conversations **Common causes**:

- TTS model loading time on first request
- Complex text processing before synthesis
- Network latency between services
- Inefficient audio generation

Why This Integration Works

This approach gives you complete control over voice synthesis while maintaining VAPI's real-time performance standards. The webhook-based design ensures:

1. **Flexibility**: Use any TTS system or model that works for your needs
2. **Scalability**: Handle the audio processing load on your own infrastructure
3. **Reliability**: Fallback options prevent call failures if your TTS is temporarily unavailable
4. **Performance**: Streaming audio delivery maintains conversation flow

By following this guide, you'll have a robust custom TTS integration that enhances your voice applications while providing the unique voice experience your users need.

Current Mode: VAPI Solution Engineer Mode - Active

Sources Used:

- VAPI Core Codebase Analysis
- Message Type Definitions
- Voice Type Specifications
- Authentication Implementation patterns
- Audio Processing requirements from core TTS implementations