Major Project Report

(23ONMCR-753)

of the program

Master of Computer Applications

Batch - Jul 2023

Fourth Semester.

CallGenius: Building Intelligent Voice AI Solutions with Vapi

Submitted by

Amandeep Singh

Master of Computer Applications

Batch - Jul 2023

UID-O23MCA110190

Synopsis

Project Title:

CallGenius: Building Intelligent Voice AI Solutions with Vapi

Introduction:

In an era where customer interaction plays a pivotal role in business growth, the need for intelligent, scalable, and human-like communication systems has become more critical than ever. This project focuses on developing an **AI-powered Call Assistant** that simulates human conversation using voice AI tools such as **Vapi AI** and **Make.com**. The assistant is capable of handling inbound and outbound calls, resolving queries, booking appointments, collecting leads, and integrating with workflow automation systems.

Objectives:

- To simulate natural, real-time human-like voice conversations using LLMs.
- To automate tasks like appointment booking, reservation handling, cancellations, and lead collection.
- To integrate with platforms such as Make.com for dynamic, real-time automation.
- To enhance user experience with features like emotion detection, background noise filtering, and low-latency responses.

Methodology:

The solution is built on a modular voice AI workflow that consists of:

1. **Listen**: Captures raw audio, applies endpointing, filters background noise, and transcribes speech.
2. **Think**: Processes transcription through LLMs and integrates emotion detection to personalize responses.
3. **Speak**: Converts generated text to natural-sounding voice responses using real-time speech synthesis with backchanneling and filler injection.

4. **Act**: Triggers automated workflows in Make.com for business operations like calendar updates, CRM entries, or form submissions.

Tools & Technologies Used:

- **Vapi AI**: Voice AI SDK for real-time speech processing
- **Make.com**: No-code workflow automation platform
- **Large Language Models (LLMs)**: For natural language understanding and response generation
- **Web & Mobile Interfaces**: For deployment and interaction

Scope:

The project addresses various real-world applications in:

- Customer service and call centers
- Appointment and reservation management
- Lead generation and qualification
- Healthcare, education, e-commerce, and hospitality sectors

Expected Outcome:

- The project delivers an AI Call Assistant capable of:
- Handling voice conversations intelligently in real-time
- Automatically performing actions based on user intent

Table of Content

Certificate

This is to certify that the Major project on titled **"**CallGenius: Building Intelligent Voice AI Solutions with Vapi**"** is a Project work done by **"Amandeep Singh"** submitted in the partial fulfillment of the requirement for the award of the degree of **"Master of Computer Applications"** from **"CHANDIGARH UNIVERSITY"** under my guidance and direction.

To the best of my knowledge and belief the data and information presented by her in the project has not been submitted earlier elsewhere.

Amandeep Singh

Declaration

I, **Amandeep Singh** solemnly declare that the project entitled **"CallGenius: Building Intelligent Voice
AI Solutions with Vapi"** submitted to **Chandigarh University** in partial fulfillment of the requirements
for the award of the degree of Master of Computer Applications, is an original work carried out by me.
I affirm that this work has not been submitted previously, in whole or in part, to any other university or
institution for any academic award.
I take full responsibility for the authenticity of the work presented herein.
Date: 30/05/2025
Place: Bangalore
Signature: *Amandeep Singh*
Name: Amandeep Singh

Acknowledgement

I would like to express my heartfelt gratitude to my project guide, expert guidance, and valuable insights throughout the development of this project. Their encouragement and technical mentorship played a crucial role in shaping this work into a practical and innovative solution.

I also extend my sincere thanks to the faculty and staff of **Chandigarh University** for providing the infrastructure, resources, and a conducive learning environment that enabled me to explore cutting-edge technologies in AI and automation.

Special appreciation goes to the development teams behind **Vapi AI** and **Make.com**, whose powerful platforms enabled seamless integration of voice AI capabilities and automation workflows in this project. Their tools were instrumental in building a fully functional AI Call Assistant capable of handling customer service, appointment scheduling, lead generation, and more.

Lastly, I am deeply thankful to my family and friends for their constant support, motivation, and patience throughout this journey. Their encouragement kept me focused and driven at every stage of this project.

Amandeep Singh

Abstract

In today's fast-paced digital landscape, businesses increasingly rely on automation to enhance customer engagement and operational efficiency. This project presents the development of an **AI Call Assistant** leveraging **Vapi AI** and **Make.com**, aimed at automating and streamlining voice-based customer interactions. The assistant is capable of handling a wide range of real-time tasks, including answering customer queries, booking and canceling appointments or reservations, collecting leads, and managing both inbound and outbound calls.

Vapi AI provides a robust low-latency voice interface powered by large language models (LLMs), simulating human-like conversations with high reliability across platforms such as web, mobile, and embedded systems. Make.com enables seamless automation of backend workflows triggered by voice inputs, thereby creating an end-to-end intelligent system. The project explores the integration of custom tools, default Vapi functionalities, and third-party automation to deliver a scalable, modular, and developer-friendly voice AI solution.

This AI Call Assistant has applications in customer service, appointment scheduling, marketing, and lead generation—making it a valuable asset for businesses aiming to optimize communication, reduce human dependency, and enhance user experience.

Introduction

In the era of digital transformation, businesses are increasingly seeking intelligent solutions to enhance operational efficiency, improve customer experience, and reduce manual intervention. Traditional call centers and customer service operations are often resource-intensive, error-prone, and unable to scale quickly with growing demand. The integration of artificial intelligence (AI) into voice-based communication offers a compelling alternative that is both cost-effective and scalable.

This project introduces an **AI Call Assistant** built using **Vapi AI** and **Make.com**, designed to simulate natural human conversation and automate a variety of tasks such as answering queries, booking appointments, canceling reservations, and collecting leads. The system is capable of handling both inbound and outbound calls across multiple platforms including web and mobile, delivering a seamless and responsive user experience.

Vapi AI is a developer-first voice AI platform that enables the creation of real-time, low-latency conversational agents powered by large language models (LLMs). It offers modular tools and integration capabilities, making it ideal for building everything from simple turn-based dialogues to complex, agent-like voice applications. **Make.com**, a powerful automation platform, is integrated to handle backend logic and workflows triggered by user inputs during calls — such as updating databases, sending confirmation emails, or triggering follow-up actions.

GitHub Repository

To facilitate version control, collaborative development, and open access to the source code, the entire project has been hosted on GitHub. This repository contains all essential files, including the source code, configuration files, documentation, and relevant resources used during the development process.

Repository Link

https://github.com/amandeepsinghh99/Ai_call_assist

Repository Contents

- **/src/** – Contains the main source code of the application.
- **/models/** – Includes any trained models or relevant AI/ML artifacts used.
- **/docs/** – Documentation files for setup, usage, and contribution.
- **README.md** – Overview of the project, features, setup instructions, and usage guide.

Software Development Life Cycle (SDLC) of the Project

The development of the AI Call Assistant followed a structured SDLC approach to ensure a robust, scalable, and natural-sounding conversational experience. The system leverages advanced voice models provided by **Vapi AI**, with real-time automation workflows powered by **Make.com**.

- **1. Requirement Analysis**
- In this phase, project requirements were gathered and documented through discussions with stakeholders. The goal was to develop an AI-powered voice assistant capable of:
- Handling inbound and outbound calls
- Understanding and responding to user queries
- Booking, canceling, or rescheduling appointments and reservations
- Performing lead generation and customer support tasks
- Integrating with automation workflows via Make.com
- **Technical Requirements Identified:**
- Voice input/output processing
- Integration with Vapi AI for real-time voice and LLM inference
- Automation workflows using Make.com
- Support for multi-platform deployment (web/mobile)

- ***2. System Design***

- The architecture of the system was designed based on a **modular voice AI pipeline**:
- **Client Layer (Voice Interface):**
    - Captures real-time raw audio from the user
    - Sends audio data to the server (or processes locally)

- **Processing Layer (Server or Cloud):**
    - **Step 1 – Listen**: Transcribes raw audio input to text
    - **Step 2 – Run LLM**: Feeds the transcript into a prompt, which is then processed using a Large Language Model (LLM) to generate intelligent responses
    - **Step 3 – Speak**: Converts the LLM output text into synthetic speech (Text-to-Speech), delivering a natural voice response
- **Automation Layer (Make.com):**
    - Executes triggered backend workflows based on user intent (e.g., create booking, cancel reservation, log a lead)
- The system supports both **cloud-based** and **client-side** processing of audio transcription and synthesis for flexibility and latency control.

- ***3. Implementation***

- The AI Call Assistant was implemented using the following stack:
- **Vapi AI**: For voice interface, real-time transcription, LLM integration, and text-to-speech (TTS) synthesis
- **Make.com**: For workflow automation triggered by call events or LLM output
- **LLM (GPT or similar)**: Integrated for generating intelligent responses based on user queries
- **Custom Tools**: Functions for CRM integration, calendar access, and lead management
- All modules were built as reusable and independently testable components.
-

- ***4. Testing***

- Testing was conducted at multiple levels to ensure a smooth conversational flow and functional accuracy:
- **Unit Testing**: Each module (transcription, LLM response, TTS, Make workflows) was tested in isolation
- **Integration Testing**: Ensured seamless data flow across Vapi AI and Make.com

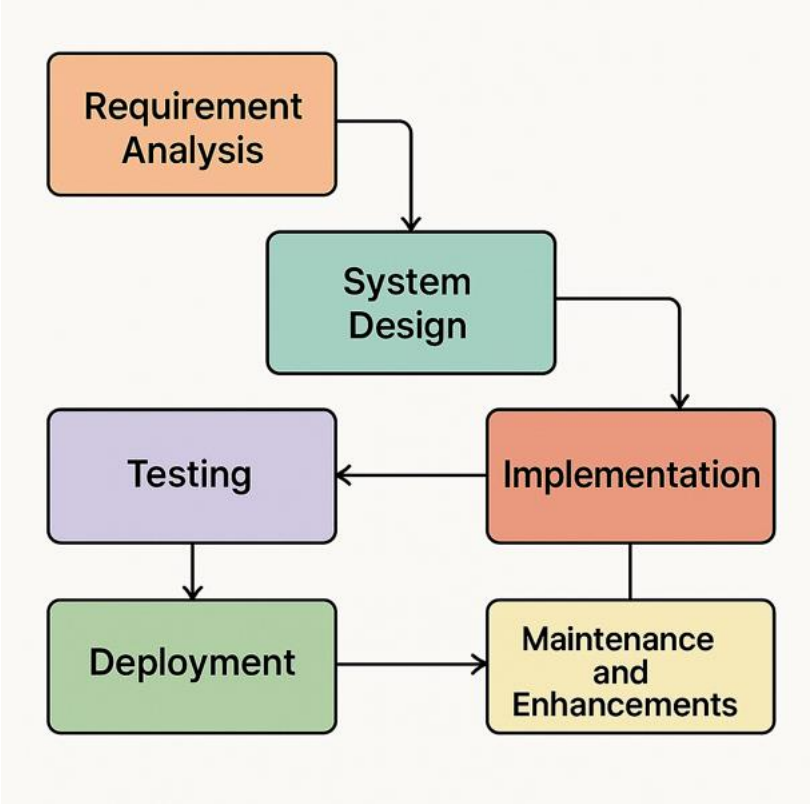
- **Performance Testing**: Measured voice response latency and ensured sub-second round-trip times
- **User Acceptance Testing (UAT)**: Simulated real-world call scenarios to validate the assistant's intelligence and reliability
-

- ***5. Deployment***

- The solution was deployed on cloud infrastructure, making the AI Call Assistant accessible through:
- Web browsers
- Mobile applications
- Embedded devices with network access
- Make.com workflows were also deployed and scheduled to monitor and handle lead submissions, calendar events, and other automated tasks.
-

- ***6. Maintenance and Enhancements***

- Post-deployment, the system is monitored continuously for:
- Latency, reliability, and uptime (particularly on Vapi AI)
- Workflow accuracy and execution on Make.com
- User feedback and conversation logs for improving intent recognition
- Regular updates to LLM prompts and voice scripts for improved responses.



System Design

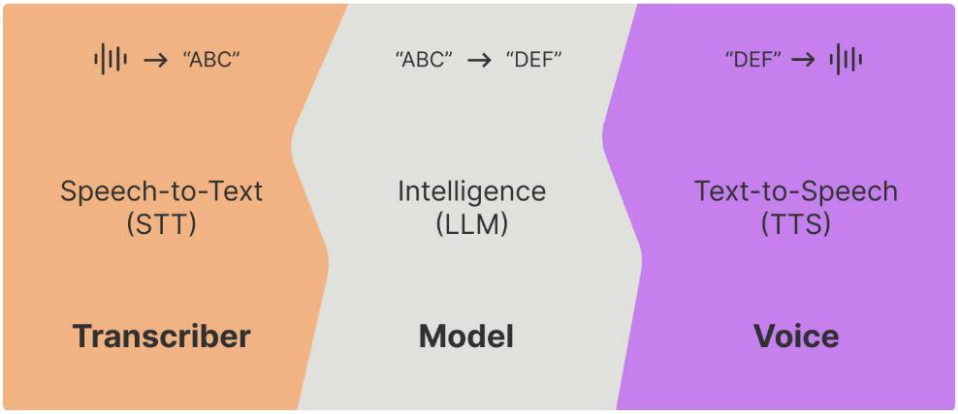
The system is designed in a **modular architecture** with the following layers:

A. Voice Interaction Pipeline (Powered by Vapi AI)

1. **Listen:**
    a. Captures raw audio input from the user in real time.
    b. Filters background noise and irrelevant speech using:
        i. **Background Noise Filtering**
        ii. **Background Voice Filtering**
    c. Transcribes user speech while maintaining speaker focus.
2. **Contextual Enhancement (Before LLM Inference):**
    a. Applies **Endpointing** to detect the natural end of user speech.
    b. Uses **Emotion Detection** to identify the user's emotional state.
    c. Handles **Interruptions (Barge-In)** to gracefully manage interjections.
3. **Think (LLM Inference):**
    a. Transcribed and enriched input is fed into a prompt for a large language model.
    b. The LLM generates an intelligent and context-aware response.

c. Adjusts response strategy based on detected emotion (e.g., polite, empathetic).

4. **Speak:**
    a. Converts LLM text output to speech using real-time **Text-to-Speech (TTS)**.
    b. Enhances speech with:
        i. **Filler Injection** to sound more human
        ii. **Backchanneling** to signal active listening (e.g., “got it”, “uh-huh”)



B. Automation Layer (Make.com)

- Executes triggered workflows like:
    - Booking or canceling a meeting/reservation
    - Logging and qualifying leads
    - Sending follow-up emails or SMS
    - Updating CRM or Google Calendar

C. User Interface Layer

- Supports voice interaction on:
    - Web
    - Mobile
    - Embedded devices (given network access)

3. Implementation

The system was implemented with a **cloud-first approach**, focusing on:

- Vapi AI SDKs for voice I/O, speech detection, and model integration
- Custom functions for CRM/calendar integrations via Make.com

- LLM prompts and intent handlers for user-specific tasks
- Real-time feedback and voice updates from Vapi's low-latency engine

Models Used:

Models Used:

Feature	Model Type	Purpose
Endpointing	Fusion Audio-Text Model	Detect user speech end, enable sub-second responses
Interruptions (Barge-in)	Custom Audio Model	Detect and handle user interjections
Noise Filtering	Proprietary Real-Time Model	Clean ambient sounds
Voice Filtering	Audio Focus Model	Ignore background speakers or echoes
Backchanneling	Audio-Text Fusion Model	Insert affirmations at the right moment
Emotion Detection	Real-Time Audio Emotion Model	Adapt LLM tone based on user emotion
Filler Injection	Streaming Text Modifier	Add "um", "like", etc. for human-like output

4. Testing

A comprehensive multi-stage testing process was used:

- **Unit Testing** – All Vapi components and Make workflows were individually tested
- **Model Behavior Testing** – Validated accuracy of backchanneling, interruptions, and emotion detection
- **Integration Testing** – Verified complete voice-to-automation flows
- **Latency Testing** – Ensured sub-second round-trip voice interactions
- **User Acceptance Testing (UAT)** – Real-world simulations tested with varied accents, emotions, and noise environments

5. Deployment

Deployment was done through:

- **Cloud-hosted services** for global accessibility
- **Web/mobile platforms** as client interfaces
- **Make.com** for real-time workflow automation
- Continuous monitoring of Vapi API usage and response times

6. Maintenance & Enhancements

Ongoing activities include:

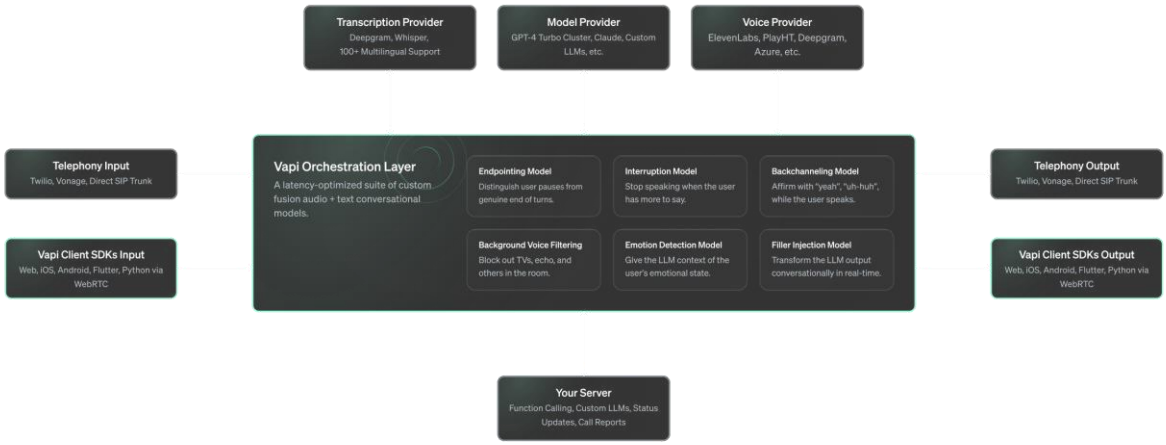
- Updating LLM prompts based on user feedback
- Improving voice model accuracy with new datasets
- Adding support for new domains (e.g., e-commerce, healthcare)
- Monitoring emotional tone recognition performance
- Expanding multilingual and regional language support

Orchestration Models

Learn about the real-time models Vapi runs on top of STT, LLM, and TTS.

On top of speech-to-text, a language model, and text-to-speech, we run a suite of real-time models that make conversations feel fast, fluid, and human.

These models are part of what we call the **orchestration layer**.



Overview

The following are the models we currently run, with many more coming soon:

Endpointing – Detects exactly when the user finishes speaking.

Interruptions – Lets users cut in and interrupt the assistant.

Background noise filtering – Cleans up ambient noise in real-time.

Background voice filtering – Ignores speech from TVs, echoes, or other people.

Backchanneling – Adds affirmations like “yeah” or “got it” at the right moments.

Emotion detection – Detects user tone and passes emotion to the LLM.

Filler injection – Adds “um”, “like”, “so” and other natural fillers to assistant responses.

Endpointing

Endpointing is a fancy word for knowing when the user is done speaking. Traditional methods use silence detection with a timeout. Unfortunately, if we want sub-second response-times, that's not going to work.
Vapi's uses a custom fusion audio-text model to know when a user has completed their turn. Based on both the user's tone and what they're saying, it decides how long to pause before hitting the LLM.
This is critical to make sure the user isn't interrupted mid-thought while still providing sub-second response times when they're done speaking.

Interruptions (Barge-in)

Interruptions (aka. barge-in in research circles) is the ability to detect when the user would like to interject and stop the assistant's speech.
Vapi uses a custom model to distinguish when there is a true interruption, like "stop", "hold up", "that's not what I mean, and when there isn't, like "yeah", "oh gotcha", "okay."
It also keeps track of where the assistant was cut off, so the LLM knows what it wasn't able to say.

Background Noise Filtering

Many of our models, including the transcriber, are audio-based. In the real world, things like music and car horns can interfere with model performance.
We use a proprietary real-time noise filtering model to ensure the audio is cleaned without sacrificing latency, before it reaches the inner models of the pipeline.

Background Voice Filtering

We rely quite heavily on the transcription model to know what's going on, for interruptions, endpointing, backchanneling, and for the user's statement passed to the LLM.
Transcription models are built to pick up everything that sounds like speech, so this can be a problem. As you can imagine, having a TV on in the background or echo coming back into the mic can severely impact the conversation ability of a system like Vapi.
Background noise cancellation is a well-researched problem. Background voice cancellation is not. To solve this, we built proprietary audio filtering model that's able to **focus in** on the primary speaker and block everything else out.

Backchanneling

Humans like to affirm each other while they speak with statements like “yeah”, “uh-huh”, “got it”, “oh no!”
They’re not considered interruptions, they’re just used to let the speaker know that their statement has been understood, and encourage the user to continue their statement.
A backchannel cue used at the wrong moment can derail a user’s statement. Vapi uses a proprietary fusion audio text model to determine the best moment to backchannel and to decide which backchannel cue is most appropriate to use.

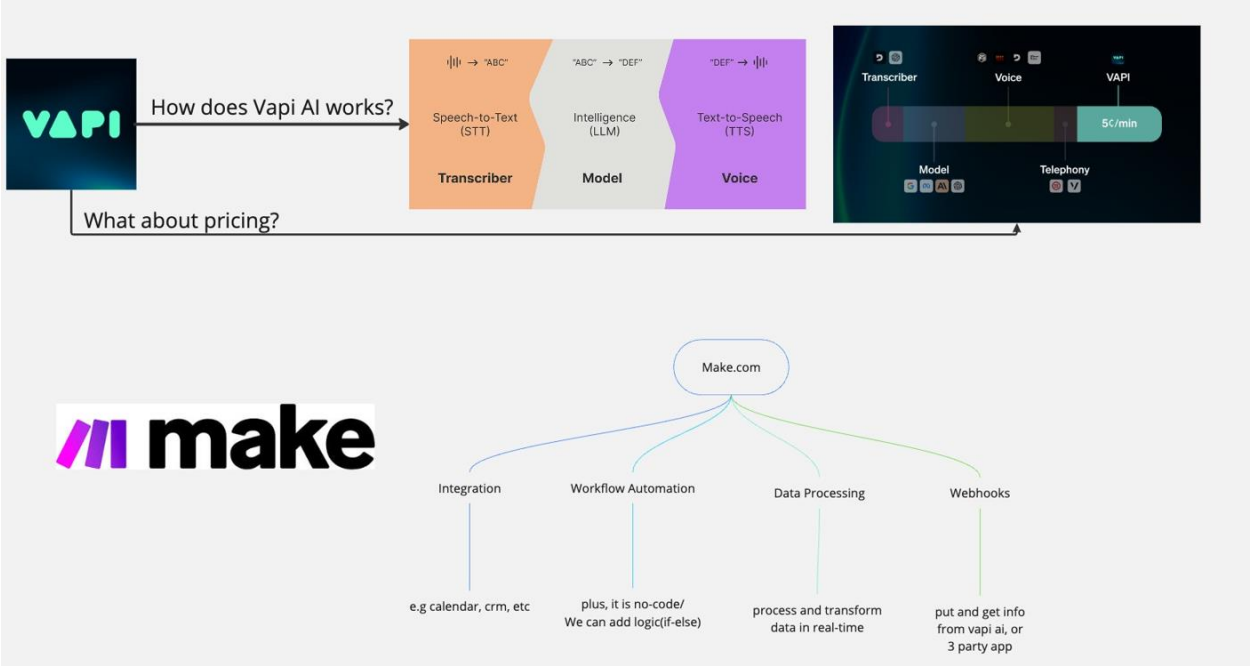
Emotion Detection

How a person says something is just as important as what they’re saying. So we’ve trained a real-time audio model to extract the emotional inflection of the user’s statement.
This emotional information is then fed into the LLM, so knows to behave differently if the user is angry, annoyed, or confused.

Filler Injection

The output of LLMs tends to be formal, and not conversational. People speak with phrases like “umm”, “ahh”, “i mean”, “like”, “so”, etc.
You can prompt the model to output like this, but we treat our user’s prompts as **sacred**.
Making a change like this to a prompt can change the behavior in unintended ways.

To ensure we don’t add additional latency transforming the output, we’ve built a custom model that’s able to convert streaming input and make it sound conversational in real-time.



Coding and Implementation

1. **Assistant customization**

title: Variables subtitle: Personalize assistant messages with dynamic and default variables slug: assistants/dynamic-variables

Overview

Use dynamic variables in the system prompt or any message in the dashboard with double curly braces (e.g., {{name}}).
To set values, make a phone call request through the API and set assistantOverrides. You cannot set variable values directly in the dashboard.
For example, set the assistant's first message to "Hello, {{name}}!" and assign name to John by passing assistantOverrides with variableValues:

```
{
 "variableValues": {
  "name": "John"
 }
}
```

Using dynamic variables in a phone call

Create a JSON payload with these key-value pairs:

- **assistantId**: Replace "your-assistant-id" with your assistant's actual ID.
- **assistantOverride**: Customize your assistant's behavior.
    - **variableValues**: Include dynamic variables in the format { "variableName": "variableValue" }. Example: { "name": "John" }.
- **customer**: Represent the call recipient.
    - **number**: Replace "+1xxxxxxxxxx" with the recipient's phone number (E.164 format).
- **phoneNumberId**: Replace "your-phone-id" with your registered phone number's ID. Find it on the Phone number page.

Send the JSON payload to the /call/phone endpoint using your preferred method (e.g., HTTP POST request).

```
{
 "assistantId": "your-assistant-id",
 "assistantOverrides": {
  "variableValues": {
   "name": "John"
  }
 },
 "customer": {
  "number": "+1xxxxxxxxxx"
 },
 "phoneNumberId": "your-phone-id"
}
```

Ensure `{{variableName}}` is included in all prompts where needed.

Default Variables

These variables are automatically filled based on the current (UTC) time, so you don't need to set them manually in variableValues:

Variable	Description	Example
{{now}}	Current date and time (UTC)	Jan 1, 2025 12:00 PM
{{date}}	Current date (UTC)	May 1, 2025
{{time}}	Current time (UTC)	12:00 PM
{{month}}	Current month (UTC)	May
{{day}}	Current day of month (UTC)	5
{{year}}	Current year (UTC)	2025
{{customer.number}}	Customer's phone number	+1xxxxxxxxxx
{{customer.X}}	Any other customer property	

Advanced date and time usage

You can use advanced date and time formatting in any prompt or message that supports dynamic variables in the dashboard or API. We use LiquidJS for formatting - see their docs for details.
Format a date or time using the LiquidJS date filter:
{{"now" | date: "%A, %B %d, %Y, %I:%M %p", "America/Los_Angeles"}}

Outputs: Thursday, May 01, 2025, 03:45 PM
Examples:

- 24-hour time:{{"now" | date: "%H:%M", "Europe/London"}}
  → 17:30
- Day of week:{{"now" | date: "%A"}}
  → Tuesday
- With customer number:Hello, your number is {{customer.number}} and the time is {{"now" | date: "%I:%M %p", "America/New_York"}}

Common formats:

Format String	Output	Description
%Y-%m-%d	2025-05-01	Year-Month-Day
%I:%M %p	03:45 PM	Hour:Minute AM/PM
%H:%M	15:45	24-hour time
%A	Monday	Day of week
%b %d, %Y	May 05, 2025	Abbrev. Month Day

Using dynamic variables in the dashboard

To use dynamic variables in the dashboard, include them in your prompts or messages using double curly braces. For example:

Hello, {{name}}!

When you start a call, you must provide a value for each variable (like `name`) in the call configuration or via the API/SDK.

<Note>
Always use double curly braces (`{{variableName}}`) to reference dynamic variables in your prompts and messages.
</Note>
</rewritten_file>

title: Multilingual subtitle: Set up multilingual support for your assistant slug: customization/multilingual

Overview

We support dozens of providers, giving you access to their available models for multilingual support.
Certain providers, like google and deepgram, have multilingual transcriber models that can transcribe audio in any language.

Transcribers (Speech-to-Text)

In the dashboard's assistant tab, click on "transcriber" to view all of the available providers, languages and models for each. Each model offers different language options.

Voice (Text-to-Speech)

Each provider includes a voice tag in the name of their voice. For example, Azure offers the es-ES-ElviraNeural voice for Spanish. Go to voice tab in the assistants page to see all of the available models.

Example: Setting Up a Spanish Voice Assistant

```
{
 "voice": {
  "provider": "azure",
  "voiceId": "es-ES-ElviraNeural"
 }
}
```

In this example, the voice es-ES-ElviraNeural from Azure supports Spanish. Replace es-ES-ElviraNeural with any other voice ID that supports your desired language.

title: Personalization with user information subtitle: Add customer-specific information to your voice assistant conversations slug: assistants/personalization

Overview

Personalization lets you include customer-specific information in your voice assistant conversations. When a customer calls, your server can provide data about that customer, which is then used to tailor the conversation in real time.
This approach is ideal for use cases like customer support, account management, or any scenario where the assistant should reference details unique to the caller.

How Personalization Works

When a call comes in, Vapi sends a request to your server instead of using a fixed assistant configuration. Your server receives the request, identifies the caller (for example, by phone

number), and fetches relevant customer data from your database or CRM. Your server responds to Vapi with either: - An existing assistant ID and a set of dynamic variables to personalize the conversation, or - A complete assistant configuration, with customer data embedded directly in the prompts or instructions. Vapi uses the personalized assistant configuration or variables to guide the conversation, referencing the customer's information as needed.

Prerequisites

- A Vapi phone number
- A created Vapi Assistant
- A server endpoint to receive Vapi's requests

Implementation

Use variable placeholders in your assistant's instructions or messages with the `{{variable_name}}` syntax. Example:
`"Hello {{customerName}}! I see you've been a {{accountType}} customer since {{joinDate}}."`
Update your phone number so that Vapi sends incoming call events to your server, rather than using a static assistant. ```json
PATCH /phone-number/{id}
{
  "assistantId": null,
  "squadId": null,
  "server": {
    "url": "https://your-server.com/api/assistant-selector"
  }
}
```

<Note>
  Your server must respond within 7.5 seconds, or the call will fail.
</Note>
Your server should handle POST requests from Vapi and return either: **Option 1: Use an Existing Assistant with Dynamic Variables**

```javascript

```
app.post("/api/assistant-selector", async (req, res) => {
  if (req.body.message?.type === "assistant-request") {
    const phoneNumber = req.body.call.from.phoneNumber;
    const customer = await crmAPI.getCustomerByPhone(phoneNumber);



    res.json({
      assistantId: "asst_customersupport",
      assistantOverrides: {
        variableValues: {
          customerName: customer.name,
          accountType: customer.tier,
          joinDate: customer.createdAt
        }
      }
    });
  }
});
```

Option 2: Return a Complete Assistant Configuration

```javascript
app.post("/api/assistant-selector", async (req, res) => {
  if (req.body.message?.type === "assistant-request") {
    const phoneNumber = req.body.call.from.phoneNumber;
    const customer = await crmAPI.getCustomerByPhone(phoneNumber);


    res.json({
      assistant: {
        name: "Dynamic Customer Support Assistant",
        model: {
          provider: "openai",
          model: "gpt-4",
          messages: [{
            role: "system",
```

```
          content: `You are helping ${customer.name}, a ${customer.tier} member since
${customer.createdAt}.`
        }]
      },
      voice: {
        provider: "11labs",
        voiceId: "shimmer"
      }
    }
  });
 }
});
```

Error Handling

If your server encounters an error or cannot find the customer, return a response like this to end the call with a spoken message:

```
{
 "error": "Unable to find customer record. Please try again later."
}
```

Voice formatting plan

title: Voice formatting plan subtitle: Format LLM output for natural-sounding speech slug: assistants/voice-formatting-plan

Overview

Voice formatting automatically transforms raw text from your language model (LLM) into a format that sounds natural when spoken by a text-to-speech (TTS) provider. This process—called **Voice Input Formatted**—is enabled by default for all assistants.

Formatting helps with things like:

- Expanding numbers and currency (e.g., $42.50 → "forty two dollars and fifty cents")
- Expanding abbreviations (e.g., ST → "STREET")

- Spacing out phone numbers (e.g., 123-456-7890 → "1 2 3 4 5 6 7 8 9 0")

You can turn off formatting if you want the TTS to read the raw LLM output.

How voice input formatting works

When enabled, the formatter runs a series of transformations on your text, each handled by a specific function. Here's the order and what each function does:

Step	Function Name	Description	Before	After	Default	Precedence
1	removeAngleBracketContent	Removes anything within <...>, except for <break>, <spell>, or double angle brackets << >>.	Hello <tag> world	Hello world	☑	-
2	removeMarkdownSymbols	Removes markdown symbols like _, `, and ~. Asterisks (*) are preserved in this step.	**Wanted** to say *hi*	**Wanted** to say *hi*	☑	0
3	removePhrasesInAsterisks	Removes text surrounded by single or double asterisks.	**Wanted** to say *hi*	to say	☒	0
4	replaceNewLinesWithPeriods	Converts new lines (\n) to periods for smoother speech.	Hello world\n to say\nWe have NASA	Hello world . to say . We have NASA	☑	0
5	replaceColonsWithPeriods	Replaces : with . for better phrasing.	price: $42.50	price. $42.50	☑	0
6	formatAcronyms	Converts known acronyms to lowercase (e.g., NASA → nasa) or spaces out	NASA and .NET	nasa and .net	☑	0

		unknown all-caps words unless they contain vowels.				
7	formatDollarAmounts	Converts currency amounts to spoken words.	$42.50	forty two dollars and fifty cents	☑	0
8	formatEmails	Replaces @ with "at" and . with "dot" in emails.	JOHN.DOE@example.COM	JOHN dot DOE at example dot COM	☑	0
9	formatDates	Converts date strings into spoken date format.	2023 05 10	Wednesday, May 10, 2023	☑	0
10	formatTimes	Expands or simplifies time expressions.	14:00	14	☑	0
11	formatDistances, formatUnits, formatPercentages, formatPhoneNumbers	Converts units, distances, percentages, and phone numbers into spoken words.	5km, 43 lb, 50%, 123-456-7890	5 kilometers, forty three pounds, 50 percent, 1 2 3 4 5 6 7 8 9 0	☑	0
12	formatNumbers	Formats general numbers: years read as digits, large numbers spelled out, negative and decimal numbers clarified.	-9, 2.5, 2023	minus nine, two point five, 2023	☑	0
13	removeAsterisks	Removes all asterisk characters from the text.	**Bold** and *italic*	Bold and italic	☑	1
14	Applying Replacements	Applies user-defined final replacements like expanding street abbreviations.	320 ST 21 RD	320 STREET 21 ROAD	☑	-

Customizing the formatting plan

You can control some aspects of formatting:

Enabled

Formatting is on by default. To disable, set:
voice.chunkPlan.formatPlan.enabled = false

Number-to-digits cutoff

Controls when numbers are read as digits instead of words.

- **Default:** 2025 (current year)
- Example: With a cutoff of 2025, numbers above this are read as digits.
- To spell out larger numbers, set the cutoff higher (e.g., 300000).

Replacements

Add exact or regex-based substitutions to customize output.

- **Example 1:** Replace hello with hi:{ type: 'exact', key: 'hello', value: 'hi' }

- **Example 2:** Replace words matching a pattern:{ type: 'regex', regex: '\b[a-zA-Z]{5}\b', value: 'hi' }

Currently, only replacements and the number-to-digits cutoff are customizable. Other options are not exposed.

Turning formatting off

To disable all formatting and use raw LLM output, set either of these to false:
voice.chunkPlan.enabled = false
// or
voice.chunkPlan.formatPlan.enabled = false

Summary

- Voice input formatting improves clarity and naturalness for TTS.
- Each transformation step targets a specific pattern for better speech output.
- You can customize or disable formatting as needed.

Background messages

title: Background messages subtitle: Silently update chat history with background messages slug: assistants/background-messages

Overview

Background messages let you add information to the chat history without interrupting or notifying the user. This is useful for logging actions, tracking background events, or updating conversation context silently.
For example, you might want to log when a user presses a button or when a background process updates the conversation. These messages help you keep a complete record of the conversation and system events, all without disrupting the user experience.

Add a button to your interface with an `onClick` event handler that will call a function to send the system message: ```html Log Action ``` When the button is clicked, the `logUserAction` function will silently insert a system message into the chat history: ```js function logUserAction() { // Function to log the user action vapi.send({ type: "add-message", message: { role: "system", content: "The user has pressed the button, say peanuts", }, }); } ``` - `vapi.send`: The primary function to interact with your assistant, handling various requests or commands. - `type: "add-message"`: Specifies the command to add a new message. - `message`: This is the actual message that you want to add to the message history. - `role`: "system" Designates the message origin as 'system', ensuring the addition is unobtrusive. Other possible values of role are 'user' | 'assistant' | 'tool' | 'function' - `content`: The actual message text to be added. - Silent logging of user activities. - Contextual updates in conversations triggered by background processes. - Non-intrusive user experience enhancements through additional information provision.

title: Assistant hooks subtitle: Automate actions on call events and interruptions slug: assistants/assistant-hooks

Overview

Assistant hooks let you automate actions when specific events occur during a call. Use hooks to transfer calls, run functions, or send messages in response to events like call ending or speech interruptions.
Supported events include:

- call.ending: When a call is ending
- assistant.speech.interrupted: When the assistant's speech is interrupted
- customer.speech.interrupted: When the customer's speech is interrupted

You can combine actions and add filters to control when hooks trigger.

How hooks work

Hooks are defined in the hooks array of your assistant configuration. Each hook includes:

- on: The event that triggers the hook
- do: The actions to perform (supports transfer, function, and say)
- filters: (Optional) Conditions that must be met for the hook to trigger

The `call.endedReason` filter can be set to any of the [call ended reasons](https://docs.vapi.ai/api-reference/calls/get#response.body.endedReason). The transfer destination type follows the [transfer call tool destinations](https://docs.vapi.ai/api-reference/tools/create#request.body.transferCall.destinations) schema.

Example: Transfer on pipeline error

Transfer a call to a fallback number if a pipeline error occurs:

```
{
 "hooks": [{
  "on": "call.ending",
  "filters": [{
   "type": "oneOf",
   "key": "call.endedReason",
   "oneOf": ["pipeline-error"]
  }],
  "do": [{
   "type": "transfer",
   "destination": {
    "type": "number",
    "number": "+1234567890",
    "callerId": "+1987654321"
   }
```

```
    }]
  }]
}
```

You can also transfer to a SIP destination:

```
{
  "hooks": [{
    "on": "call.ending",
    "filters": [{
      "type": "oneOf",
      "key": "call.endedReason",
      "oneOf": ["pipeline-error"]
    }],
    "do": [{
      "type": "transfer",
      "destination": {
        "type": "sip",
        "sipUri": "sip:user@domain.com"
      }
    }]
  }]
}
```

Example: Combine actions on pipeline error

Perform multiple actions—say a message, call a function, and transfer the call—when a pipeline error occurs:

```
{
  "hooks": [{
    "on": "call.ending",
    "filters": [{
      "type": "oneOf",
      "key": "call.endedReason",
      "oneOf": ["pipeline-error"]
    }],
    "do": [
      {
```

```
      "type": "say",
      "exact": "I apologize for the technical difficulty. Let me transfer you to our support
team."
    },
    {
      "type": "function",
      "function": {
        "name": "log_error",
        "parameters": {
          "type": "object",
          "properties": {
            "error_type": {
              "type": "string",
              "value": "pipeline_error"
            }
          }
        },
        "description": "Logs the error details for monitoring"
      },
      "async": true,
      "server": {
        "url": "https://your-server.com/api"
      }
    },
    {
      "type": "transfer",
      "destination": {
        "type": "number",
        "number": "+1234567890",
        "callerId": "+1987654321"
      }
    }
  ]
 }]
}
```

Example: Handle speech interruptions

Respond when the assistant's speech is interrupted by the customer:

```
{
 "hooks": [{
  "on": "assistant.speech.interrupted",
  "do": [{
   "type": "say",
   "exact": ["Sorry about that", "Go ahead", "Please continue"]
  }]
 }]
}
```

Handle customer speech interruptions in a similar way:

```
{
 "hooks": [{
  "on": "customer.speech.interrupted",
  "do": [{
   "type": "say",
   "exact": "I apologize for interrupting. Please continue."
  }]
 }]
}
```

Common use cases

- Transfer to a human agent on errors
- Route to a fallback system if the assistant fails
- Handle customer or assistant interruptions gracefully
- Log errors or events for monitoring
  Use `"oneOf": ["pipeline-error"]` as a catch-all filter for any pipeline-related error reason.
  Speech configuration

Overview

Speech configuration lets you control exactly when your assistant starts and stops speaking during a conversation. By tuning these settings, you can make your assistant feel more natural, avoid interrupting the customer, and reduce awkward pauses.
Speech speed can be controlled, but only PlayHT currently supports this feature with the `speed` field. Other providers do not currently support speed.
The two main components are:

- **Speaking Plan**: Controls when the assistant begins speaking after the customer finishes or pauses.
- **Stop Speaking Plan**: Controls when the assistant stops speaking if the customer starts talking.

Fine-tuning these plans helps you adapt the assistant's responsiveness to your use case—whether you want fast, snappy replies or a more patient, human-like conversation flow.
Currently, these configurations can only be set via API.
The rest of this page explains each setting and provides practical examples for different scenarios.

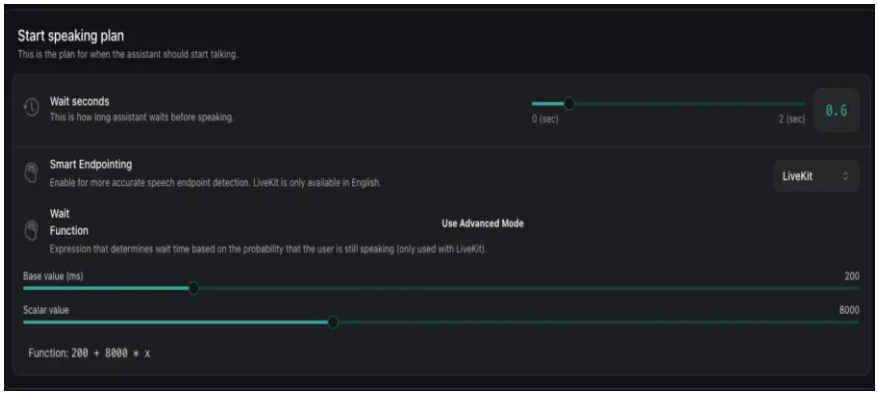
Start Speaking Plan

This plan defines the parameters for when the assistant begins speaking after the customer pauses or finishes.

- **Wait Time Before Speaking**: You can set how long the assistant waits before speaking after the customer finishes. The default is 0.4 seconds, but you can increase it if the assistant is speaking too soon, or decrease it if there's too much delay.

Example: For tech support calls, set waitSeconds for the assistant to more than 1.0 seconds to give customers time to complete their thoughts, even if they have some pauses in between.

- **Smart Endpointing Plan**: This feature uses advanced processing to detect when the customer has truly finished speaking, especially if they pause mid-thought. It can be configured in three ways:
    - **Off**: Disabled by default
    - **LiveKit**: Recommended for English conversations as it provides the most sophisticated solution for detecting natural speech patterns and pauses. LiveKit can be fine-tuned using the waitFunction parameter to adjust response timing based on the probability that the user is still speaking.
    - **Vapi**: Recommended for non-English conversations or as an alternative when LiveKit isn't suitable



LiveKit Smart Endpointing Configuration: When using LiveKit, you can customize the waitFunction parameter which determines how long the bot will wait to start speaking based on the likelihood that the user has finished speaking:
waitFunction: "200 + 8000 * x"

This function maps probabilities (0-1) to milliseconds of wait time. A probability of 0 means high confidence the caller has stopped speaking, while 1 means high confidence they're still speaking. The default function (200 + 8000 * x) creates a wait time between 200ms (when x=0) and 8200ms (when x=1). You can customize this with your own mathematical expression, such as 4000 * (1 - cos(pi * x)) for a different response curve.
Example: In insurance claims, smart endpointing helps avoid interruptions while customers think through complex responses. For instance, when the assistant asks "do you want a loan," the system can intelligently wait for the complete response rather than interrupting after the initial "yes" or "no." For responses requiring number sequences like "What's your account number?", the system can detect natural pauses between digits without prematurely ending the customer's turn to speak.

- **Transcription-Based Detection**: Customize how the assistant determines that the customer has stopped speaking based on what they're saying. This offers more control over the timing. **Example:** When a customer says, "My account number is 123456789, I want to transfer $500."
    - The system detects the number "123456789" and waits for 0.5 seconds (WaitSeconds) to ensure the customer isn't still speaking.
    - If the customer were to finish with an additional line, "I want to transfer $500.", the system uses onPunctuationSeconds to confirm the end of the speech and then proceed with the request processing.
    - In a scenario where the customer has been silent for a long and has already finished speaking but the transcriber is not confident to punctuate the transcription, onNoPunctuationSeconds is used for 1.5 seconds.

Stop Speaking Plan

The Stop Speaking Plan defines when the assistant stops talking after detecting customer speech.

- **Words to Stop Speaking**: Define how many words the customer needs to say before the assistant stops talking. If you want immediate reaction, set this to 0. Increase it to avoid interruptions by brief acknowledgments like "okay" or "right". **Example:** While setting an appointment with a clinic, set numWords to 2-3 words to allow customers to finish brief clarifications without triggering interruptions.
- **Voice Activity Detection**: Adjust how long the customer needs to be speaking before the assistant stops. The default is 0.2 seconds, but you can tweak this to balance responsiveness and avoid false triggers.

Example: For a banking call center, setting a higher voiceSeconds value ensures accuracy by reducing false positives. This avoids interruptions caused by background sounds, even if it slightly delays the detection of speech onset. This tradeoff is essential to ensure the assistant processes only correct and intended information.

- **Pause Before Resuming**: Control how long the assistant waits before starting to talk again after being interrupted. The default is 1 second, but you can adjust it depending on how quickly the assistant should resume.

Example: For quick queries (e.g., "What's the total order value in my cart?"), set backoffSeconds to 1 second.

Here's a code snippet for Stop Speaking Plan -

```
"stopSpeakingPlan": {
   "numWords": 0,
   "voiceSeconds": 0.2,
   "backoffSeconds": 1
 }
```

Considerations for Configuration

- **Customer Style**: Think about whether the customer pauses mid-thought or provides continuous speech. Adjust wait times and enable smart endpointing as needed.
- **Background Noise**: If there's a lot of background noise, you may need to tweak the settings to avoid false triggers. Default for phone calls is 'office' and default for web calls is 'off'.

```
"backgroundSound": "off",
```

- **Conversation Flow**: Aim for a balance where the assistant is responsive but not intrusive. Test different settings to find the best fit for your needs.

Voice Fallback Plan

Configure fallback voices that activate automatically if your primary voice fails.
Voice fallback plans can currently only be configured through the API. We are working on making this available through our dashboard.

Introduction

Voice fallback plans give you the ability to continue your call in the event that your primary voice fails. Your assistant will sequentially fallback to only the voices you configure within your plan, in the exact order you specify.
Without a fallback plan configured, your call will end with an error in the event that your chosen voice provider fails.

How It Works

When a voice failure occurs, Vapi will:

1. Detect the failure of the primary voice
2. If a custom fallback plan exists:

Switch to the first fallback voice in your plan
Continue through your specified list if subsequent failures occur
Terminate only if all voices in your plan have failed

Configuration

Add the **fallbackPlan** property to your assistant's voice configuration, and specify the fallback voices within the **voices** property.
Please note that fallback voices must be valid JSON configurations, and not strings.
The order matters. Vapi will choose fallback voices starting from the beginning of the list.

```
1  {
```

```
2    "voice": {
3    "provider": "openai",
4    "voiceId": "shimmer",
5    "fallbackPlan": {
6    "voices": [
7    {
8    "provider": "cartesia",
9    "voiceId": "248be419-c632-4f23-adf1-5324ed7dbf1d"
10   },
11   {
12   "provider": "playht",
13   "voiceId": "jennifer"
14   }
15   ]
16   }
17   }
18   }
```

Best practices

Use **different providers** for your fallback voices to protect against provider-wide outages. Select voices with **similar characteristics** (tone, accent, gender) to maintain consistency in the user experience.

Provider Keys

Have a custom model or voice with one of the providers? Or an enterprise account with volume pricing?

No problem! You can bring your own API keys to Vapi. You can add them in the Dashboard under the **Provider Keys** tab. Once your API key is validated, you won't be charged when using that provider through Vapi. Instead, you'll be charged directly by the provider.

Transcription Providers

Currently, the only available transcription provider is deepgram. To use a custom model, you can specify the deepgram model ID in the transcriber.model parameter of the Assistant.

Model Providers

We are currently have support for any OpenAI-compatible endpoint. This includes services like OpenRouter, AnyScale, Together AI, or your own server.
To use one of these providers, you can specify the provider and model in the model parameter of the Assistant.
You can find more details in the Custom LLMs section of the documentation.

Voice Providers

All voice providers are supported. Once you've validated your API through the Dashboard, any voice ID from your provider can be used in the voice.voiceId field of the Assistant.

Cloud Providers

Vapi stores recordings of conversations with assistants in the cloud. By default, Vapi stores these recordings in its own bucket in Cloudflare R2. You can configure Vapi to store recordings in your own bucket in AWS S3, GCP, or Cloudflare R2.
You can find more details on how to configure your Cloud Provider keys here:

- AWS S3
- GCP Cloud Storage
- Cloudflare R2

Custom transcriber

Overview

A custom transcriber lets you use your own transcription service with Vapi, instead of a built-in provider. This is useful if you need more control, want to use a specific provider like Deepgram, or have custom processing needs.

Why Use a Custom Transcriber?

- **Flexibility:** Integrate with your preferred transcription service.
- **Control:** Implement specialized processing that isn't available with built-in providers.
- **Cost Efficiency:** Leverage your existing transcription infrastructure while maintaining full control over the pipeline.

- **Customization:** Tailor the handling of audio data, transcript formatting, and buffering according to your specific needs.

How it works

Vapi connects to your custom transcriber endpoint (e.g. `/api/custom-transcriber`) via WebSocket. It sends an initial JSON message like this: ```json { "type": "start", "encoding": "linear16", "container": "raw", "sampleRate": 16000, "channels": 2 } ``` Vapi then streams binary PCM audio to your server. Your server forwards the audio to Deepgram (or your chosen transcriber) using its SDK. Deepgram processes the audio and returns transcript events that include a `channel_index` (e.g. `[0, ...]` for customer, `[1, ...]` for assistant). The service buffers the incoming data, processes the transcript events (with debouncing and channel detection), and emits a final transcript. The final transcript is sent back to Vapi as a JSON message: ```json { "type": "transcriber-response", "transcription": "The transcribed text", "channel": "customer" // or "assistant" } ```

Implementation steps

Create a new Node.js project and install the required dependencies: ```bash mkdir vapi-custom-transcriber cd vapi-custom-transcriber npm init -y npm install ws express dotenv @deepgram/sdk ``` Create a `.env` file with the following content: ```env DEEPGRAM_API_KEY=your_deepgram_api_key PORT=3001 ``` Add the following files to your project: **transcriptionService.js**

```js
const { createClient, LiveTranscriptionEvents } = require("@deepgram/sdk");
const EventEmitter = require("events");

const PUNCTUATION_TERMINATORS = [".", "!", "?"];
const MAX_RETRY_ATTEMPTS = 3;
const DEBOUNCE_DELAY_IN_SECS = 3;
const DEBOUNCE_DELAY = DEBOUNCE_DELAY_IN_SECS * 1000;
const DEEPGRAM_API_KEY = process.env["DEEPGRAM_API_KEY"] || "";

class TranscriptionService extends EventEmitter {
  constructor(config, logger) {
 super();
    this.config = config;
    this.logger = logger;
```

```
this.flowLogger = require("./fileLogger").createNamedLogger(
  "transcriber-flow.log"
);
if (!DEEPGRAM_API_KEY) {
  throw new Error("Missing Deepgram API Key");
}
this.deepgramClient = createClient(DEEPGRAM_API_KEY);
this.logger.logDetailed(
  "INFO",
  "Initializing Deepgram live connection",
  "TranscriptionService",
  {
    model: "nova-2",
    sample_rate: 16000,
    channels: 2,
  }
);
this.deepgramLive = this.deepgramClient.listen.live({
  encoding: "linear16",
  channels: 2,
  sample_rate: 16000,
  model: "nova-2",
  smart_format: true,
  interim_results: true,
  endpointing: 800,
  language: "en",
  multichannel: true,
});
this.finalResult = { customer: "", assistant: "" };
this.audioBuffer = [];
this.retryAttempts = 0;
this.lastTranscriptionTime = Date.now();
this.pcmBuffer = Buffer.alloc(0);

this.deepgramLive.addListener(LiveTranscriptionEvents.Open, () => {
  this.logger.logDetailed(
    "INFO",
```

```
      "Deepgram connection opened",
      "TranscriptionService"
    );
    this.deepgramLive.on(LiveTranscriptionEvents.Close, () => {
      this.logger.logDetailed(
        "INFO",
        "Deepgram connection closed",
        "TranscriptionService"
      );
      this.emitTranscription();
      this.audioBuffer = [];
    });
    this.deepgramLive.on(LiveTranscriptionEvents.Metadata, (data) => {
      this.logger.logDetailed(
        "DEBUG",
        "Deepgram metadata received",
        "TranscriptionService",
        data
      );
    });
    this.deepgramLive.on(LiveTranscriptionEvents.Transcript, (event) => {
      this.handleTranscript(event);
    });
    this.deepgramLive.on(LiveTranscriptionEvents.Error, (err) => {
      this.logger.logDetailed(
        "ERROR",
        "Deepgram error received",
        "TranscriptionService",
        { error: err }
      );
      this.emit("transcriptionerror", err);
    });
  });
}

send(payload) {
  if (payload instanceof Buffer) {
```

```
    this.pcmBuffer =
      this.pcmBuffer.length === 0
        ? payload
        : Buffer.concat([this.pcmBuffer, payload]);
   } else {
    this.logger.warn("TranscriptionService: Received non-Buffer data chunk.");
   }
   if (this.deepgramLive.getReadyState() === 1 && this.pcmBuffer.length > 0) {
    this.sendBufferedData(this.pcmBuffer);
    this.pcmBuffer = Buffer.alloc(0);
   }
  }

  sendBufferedData(bufferedData) {
   try {
    this.logger.logDetailed(
      "INFO",
      "Sending buffered data to Deepgram",
      "TranscriptionService",
      { bytes: bufferedData.length }
    );
    this.deepgramLive.send(bufferedData);
    this.audioBuffer = [];
    this.retryAttempts = 0;
   } catch (error) {
    this.logger.logDetailed(
      "ERROR",
      "Error sending buffered data",
      "TranscriptionService",
      { error }
    );
    this.retryAttempts++;
    if (this.retryAttempts <= MAX_RETRY_ATTEMPTS) {
     setTimeout(() => {
       this.sendBufferedData(bufferedData);
     }, 1000);
    } else {
```

```
      this.logger.logDetailed(
        "ERROR",
        "Max retry attempts reached, discarding data",
        "TranscriptionService"
      );
      this.audioBuffer = [];
      this.retryAttempts = 0;
    }
  }
}

handleTranscript(transcription) {
  if (!transcription.channel || !transcription.channel.alternatives?.[0]) {
    this.logger.logDetailed(
      "WARN",
      "Invalid transcript format",
      "TranscriptionService",
      { transcription }
    );
    return;
  }
  const text = transcription.channel.alternatives[0].transcript.trim();
  if (!text) return;
  const currentTime = Date.now();
  const channelIndex = transcription.channel_index
    ? transcription.channel_index[0]
    : 0;
  const channel = channelIndex === 0 ? "customer" : "assistant";
  this.logger.logDetailed(
    "INFO",
    "Received transcript",
    "TranscriptionService",
    { channel, text }
  );
  if (transcription.is_final || transcription.speech_final) {
    this.finalResult[channel] += ` ${text}`;
    this.emitTranscription();
```

```
    } else {
      this.finalResult[channel] += ` ${text}`;
      if (currentTime - this.lastTranscriptionTime >= DEBOUNCE_DELAY) {
        this.logger.logDetailed(
          "INFO",
          `Emitting transcript after ${DEBOUNCE_DELAY_IN_SECS}s inactivity`,
          "TranscriptionService"
        );
        this.emitTranscription();
      }
    }
    this.lastTranscriptionTime = currentTime;
  }

  emitTranscription() {
    for (const chan of ["customer", "assistant"]) {
      if (this.finalResult[chan].trim()) {
        const transcript = this.finalResult[chan].trim();
        this.logger.logDetailed(
          "INFO",
          "Emitting transcription",
          "TranscriptionService",
          { channel: chan, transcript }
        );
        this.emit("transcription", transcript, chan);
        this.finalResult[chan] = "";
      }
    }
  }
}

module.exports = TranscriptionService;
```

server.js
```js
const express = require("express");
```

```
const http = require("http");
const TranscriptionService = require("./transcriptionService");
const FileLogger = require("./fileLogger");
require("dotenv").config();

const app = express();
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

app.get("/", (req, res) => {
  res.send("Custom Transcriber Service is running");
});

const server = http.createServer(app);

const config = {
  DEEPGRAM_API_KEY: process.env.DEEPGRAM_API_KEY,
  PORT: process.env.PORT || 3001,
};

const logger = new FileLogger();
const transcriptionService = new TranscriptionService(config, logger);

transcriptionService.setupWebSocketServer = function (server) {
  const WebSocketServer = require("ws").Server;
  const wss = new WebSocketServer({ server, path: "/api/custom-transcriber" });
  wss.on("connection", (ws) => {
    logger.logDetailed(
      "INFO",
      "New WebSocket client connected on /api/custom-transcriber",
      "Server"
    );
    ws.on("message", (data, isBinary) => {
      if (!isBinary) {
        try {
          const msg = JSON.parse(data.toString());
```

```
      if (msg.type === "start") {
        logger.logDetailed(
          "INFO",
          "Received start message from client",
          "Server",
          { sampleRate: msg.sampleRate, channels: msg.channels }
        );
      }
    } catch (err) {
      logger.error("JSON parse error", err, "Server");
    }
  } else {
    transcriptionService.send(data);
  }
});
ws.on("close", () => {
  logger.logDetailed("INFO", "WebSocket client disconnected", "Server");
  if (
    transcriptionService.deepgramLive &&
    transcriptionService.deepgramLive.getReadyState() === 1
  ) {
    transcriptionService.deepgramLive.finish();
  }
});
ws.on("error", (error) => {
  logger.error("WebSocket error", error, "Server");
});
transcriptionService.on("transcription", (text, channel) => {
  const response = {
    type: "transcriber-response",
    transcription: text,
    channel,
  };
  ws.send(JSON.stringify(response));
  logger.logDetailed("INFO", "Sent transcription to client", "Server", {
    channel,
    text,
```

```
    });
  });
  transcriptionService.on("transcriptionerror", (err) => {
    ws.send(
      JSON.stringify({ type: "error", error: "Transcription service error" })
    );
    logger.error("Transcription service error", err, "Server");
  });
 });
};

transcriptionService.setupWebSocketServer(server);

server.listen(config.PORT, () => {
 console.log(`Server is running on http://localhost:${config.PORT}`);
});
```

1. **Deploy your server:** ```bash node server.js ``` 2. **Expose your server:** Use a tool like ngrok to expose your server via HTTPS/WSS. 3. **Initiate a call with Vapi:** Use the following CURL command (update the placeholders with your actual values): ```bash curl -X POST https://api.vapi.ai/call \ -H "Authorization: Bearer YOUR_API_KEY" \ -H "Content-Type: application/json" \ -d '{ "phoneNumberId": "YOUR_PHONE_NUMBER_ID", "customer": { "number": "CUSTOMER_PHONE_NUMBER" }, "assistant": { "transcriber": { "provider": "custom-transcriber", "server": { "url": "wss://your-server.ngrok.io/api/custom-transcriber" }, "secret": "your_optional_secret_value" }, "firstMessage": "Hello! I am using a custom transcriber with Deepgram." }, "name": "CustomTranscriberTest" }' ``` **Expected behavior:**

- Vapi connects via WebSocket to your custom transcriber at `/api/custom-transcriber`.
- The `"start"` message initializes the Deepgram session.
- PCM audio data is forwarded to Deepgram.
- Deepgram returns transcript events, which are processed with channel detection and debouncing.
- The final transcript is sent back as a JSON message:

```json
{
  "type": "transcriber-response",
  "transcription": "The transcribed text",
```

```
  "channel": "customer" // or "assistant"
}
```

Notes and limitations

- **Streaming support requirement:**

The custom transcriber must support streaming. Vapi sends continuous audio data over the WebSocket, and your server must handle this stream in real time.

- **Secret header:** The custom transcriber configuration accepts an optional field called **secret**. When set, Vapi will send this value with every request as an HTTP header named x-vapi-secret. This can also be configured via a headers field.
- **Buffering:**

The solution buffers PCM audio and performs simple validation (e.g. ensuring stereo PCM data length is a multiple of 4). If the audio data is malformed, it is trimmed to a valid length.

- **Channel detection:**

Transcript events from Deepgram include a channel_index array. The service uses the first element to determine whether the transcript is from the customer (0) or the assistant (1). Ensure Deepgram's response format remains consistent with this logic.

Custom LLMs

Fine-tuned OpenAI models

Use Another LLM or Your Own Server

Vapi supports using any OpenAI-compatible endpoint as the LLM. This includes services like OpenRouter, AnyScale, Together AI, or your own server.

Using an LLM provider

You'll first want to POST your API key via the **/credential** endpoint:

```
1  {
2  "provider": "openrouter",
3  "apiKey": "<YOUR OPENROUTER KEY>"
4  }
```

Then, you can create an assistant with the model provider:

```
1    {
```

```
2   "name": "My Assistant",
3   "model": {
4   "provider": "openrouter",
5   "model": "cognitivecomputations/dolphin-mixtral-8x7b",
6   "messages": [
7   {
8   "role": "system",
9   "content": "You are an assistant."
10  }
11  ],
12  "temperature": 0.7
13  }
14  }
```

Using Fine-Tuned OpenAI Models

To set up your OpenAI Fine-Tuned model, you need to follow these steps:

1. Set the custom llm URL to **https://api.openai.com/v1**.
2. Assign the custom llm key to the OpenAI key.
3. Update the model to their model.
4. Execute a PATCH request to the **/assistant** endpoint and ensure that **model.metadataSendMode** is set to off.

Using your server

To set up your server to act as the LLM, you'll need to create an endpoint that is compatible with the OpenAI Client. For best results, your endpoint should also support streaming completions.
If your server is making calls to an OpenAI compatble API, you can pipe the requests directly back in your response to Vapi.
If you'd like your OpenAI-compatible endpoint to be authenticated, you can POST your server's API key and URL via the **/credential** endpoint:

```
1   {
2   "provider": "custom-llm",
3   "apiKey": "<YOUR SERVER API KEY>"
```

```
4  }
```

If your server isn’t authenticated, you can skip this step.

Then, you can create an assistant with the **custom-llm** model provider:

```
{
"name": "My Assistant",
"model": {
"provider": "custom-llm",
"url": "<YOUR OPENAI COMPATIBLE ENDPOINT BASE URL>",
"model": "my-cool-model",
"messages": [
{
"role": "system",
"content": "You are an assistant."
}
],
"temperature": 0.7
}


}
```

title: Custom LLM Tool Calling Integration slug: customization/tool-calling-integration

What Is a Custom LLM and Why Use It?

A **Custom LLM** is more than just a text generator—it's a conversational assistant that can call external functions, trigger processes, and handle special logic, all while chatting with your users. Think of it as your smart helper that not only answers questions but also takes actions.

Why use a Custom LLM?

- **Enhanced Functionality:** It mixes natural language responses with actionable functions.
- **Flexibility:** You can combine built-in functions, attach external tools via Vapi, or even add custom endpoints.
- **Dynamic Interactions:** The assistant can return structured instructions—like transferring a call or running a custom process—when needed.
- **Seamless Integration:** Vapi lets you plug these custom endpoints into your assistant quickly and easily.

Setting Up Your Custom LLM for Response Generation

Before adding tool calls, let's start with the basics: setting up your Custom LLM to simply generate conversation responses. In this mode, your LLM receives conversation details, asks the model for a reply, and streams that text back.

How It Works

- **Request Reception:** Your endpoint (e.g., /chat/completions) gets a payload with the model, messages, temperature, and (optionally) tools.
- **Content Generation:** The code builds an OpenAI API request that includes the conversation context.
- **Response Streaming:** The generated reply is sent back as Server-Sent Events (SSE).

Sample Code Snippet

```
app.post("/chat/completions", async (req: Request, res: Response) => {
  // Log the incoming request.
  logEvent("Request received at /chat/completions", req.body);
  const payload = req.body;


  // Prepare the API request to OpenAI.
```

```
  const requestArgs: any = {
        model: payload.model,
        messages: payload.messages,
        temperature: payload.temperature ?? 1.0,
        stream: true,
        tools: payload.tools || [],
        tool_choice: "auto",
  };

  // Optionally merge in native tool definitions.
  const modelTools = payload.tools || [];
  requestArgs.tools = [...modelTools, ...ourTools];

  logEvent("Calling OpenAI API for content generation");
  const openAIResponse = await openai.chat.completions.create(requestArgs);
  logEvent("OpenAI API call successful. Streaming response.");

  // Set up streaming headers.
  res.setHeader("Content-Type", "text/event-stream");
  res.setHeader("Cache-Control", "no-cache");
  res.setHeader("Connection", "keep-alive");

  // Stream the response chunks back.
  for await (const chunk of openAIResponse as unknown as AsyncIterable<any>) {
        res.write(`data: ${JSON.stringify(chunk)}\n\n`);
  }
  res.write("data: [DONE]\n\n");
  res.end();
});
```

Attaching Custom LLM Without Tools to an Existing Assistant in Vapi

If you just want response generation (without tool calls), update your Vapi model with a PATCH request like this:

```
curl -X PATCH https://api.vapi.ai/assistant/insert-your-assistant-id-here \
        -H "Authorization: Bearer insert-your-private-key-here" \
        -H "Content-Type: application/json" \
        -d '{
```

```
  "model": {
        "provider": "custom-llm",
        "model": "gpt-4o",
        "url": "https://custom-llm-url/chat/completions",
        "messages": [
        {
        "role": "system",
        "content": "[TASK] Ask the user if they want to transfer the call; if not, continue the conversation."
        }
        ]
  },
  "transcriber": {
        "provider": "azure",
        "language": "en-CA"
  }
}'
```

Adding Tools Calling with Your Custom LLM

Now that you've got response generation working, let's expand your assistant's abilities. Your Custom LLM can trigger external actions in three different ways.

a. Native LLM Tools

These tools are built right into your LLM integration. For example, a native function like get_payment_link can return a payment URL.

How It Works:

1. **Detection:** The LLM's streaming response includes a tool call for get_payment_link.
2. **Execution:** The integration parses the arguments and calls the native function.
3. **Response:** The result is packaged into a follow-up API call and streamed back.

Code Snippet:

```
// Variables to accumulate tool call information.
let argumentsStr = "";
let toolCallInfo: { name?: string; id?: string } | null = null;


// Process streaming chunks.
for await (const chunk of openAIResponse as unknown as AsyncIterable<any>) {
```

```
const choice = chunk.choices && chunk.choices[0];
const delta = choice?.delta || {};
const toolCalls = delta.tool_calls;

if (toolCalls && toolCalls.length > 0) {
        for (const toolCall of toolCalls) {
        const func = toolCall.function;
        if (func && func.name) {
        toolCallInfo = { name: func.name, id: toolCall.id };
        }
        if (func && func.arguments) {
        argumentsStr += func.arguments;
        }
        }
}

const finishReason = choice?.finish_reason;
if (finishReason === "tool_calls" && toolCallInfo) {
        let parsedArgs = {};
        try {
        parsedArgs = JSON.parse(argumentsStr);
        } catch (err) {
        console.error("Failed to parse arguments:", err);
        }
        if (tool_functions[toolCallInfo.name!]) {
        const result = await tool_functions[toolCallInfo.name!](parsedArgs);
        const functionMessage = {
        role: "function",
        name: toolCallInfo.name,
        content: JSON.stringify(result)
        };

        const followUpResponse = await openai.chat.completions.create({
        model: requestArgs.model,
        messages: [...requestArgs.messages, functionMessage],
        temperature: requestArgs.temperature,
        stream: true,
```

```
        tools: requestArgs.tools,
        tool_choice: "auto"
        });

        for await (const followUpChunk of followUpResponse) {
        res.write(`data: ${JSON.stringify(followUpChunk)}\n\n`);
        }
        argumentsStr = "";
        toolCallInfo = null;
        continue;
        }
 }
 res.write(`data: ${JSON.stringify(chunk)}\n\n`);
}
```

b. Vapi-Attached Tools

These tools come pre-attached via your Vapi configuration. For example, the transferCall tool:

How It Works:

1. **Detection:** When a tool call for transferCall appears with a destination in the payload, the function isn't executed.
2. **Response:** The integration immediately sends a function call payload with the destination back to Vapi.

Code Snippet:

```
if (functionName === "transferCall" && payload.destination) {
 const functionCallPayload = {
        function_call: {
        name: "transferCall",
        arguments: {
        destination: payload.destination,
        },
        },
 };
 logEvent("Special handling for transferCall", { functionCallPayload });
 res.write(`data: ${JSON.stringify(functionCallPayload)}\n\n`);
 // Skip further processing for this chunk.
```

```
  continue;
}
```

c. Custom Tools

Custom tools are unique to your application and are handled by a dedicated endpoint. For example, a custom function named processOrder.

How It Works:

1. **Dedicated Endpoint:** Requests for custom tools go to /chat/completions/custom-tool.
2. **Detection:** The payload includes a tool call list. If the function name is "processOrder", a hardcoded result is returned.
3. **Response:** A JSON response is sent back with the result.

Code Snippet (Custom Endpoint):

```
app.post("/chat/completions/custom-tool", async (req: Request, res: Response) => {
  logEvent("Received request at /chat/completions/custom-tool", req.body);
  // Expect the payload to have a "message" with a "toolCallList" array.
  const vapiPayload = req.body.message;


  // Process tool call.
  for (const toolCall of vapiPayload.toolCallList) {
        if (toolCall.function?.name === "processOrder") {
        const hardcodedResult = "CustomTool processOrder With CustomLLM Always
Works";
        logEvent("Returning hardcoded result for 'processOrder'", { toolCallId: toolCall.id });
        return res.json({
        results: [
        {
        toolCallId: toolCall.id,
        result: hardcodedResult,
        },
        ],
        });
        }
  }
});
```

Testing Tool Calling with cURL

Once your endpoints are set up, try testing them with these cURL commands.

a. Native Tool Calling (get_payment_link)

```
curl -X POST https://custom-llm-url/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
        "model": "gpt-3.5-turbo",
        "messages": [
        {"role": "user", "content": "I need a payment link."}
        ],
        "temperature": 0.7,
        "tools": [
        {
        "type": "function",
        "function": {
        "name": "get_payment_link",
        "description": "Get a payment link",
        "parameters": {}
        }
        }
        ]
        }'
```

Expected Response:

Streaming chunks eventually include the result (e.g., a payment link) returned by the native tool function.

b. Vapi-Attached Tool Calling (transferCall)

```
curl -X POST https://custom-llm-url/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
        "model": "gpt-3.5-turbo",
        "messages": [
        {"role": "user", "content": "Please transfer my call."}
        ],
        "temperature": 0.7,
        "tools": [
        {
```

```
"type": "function",
"function": {
"name": "transferCall",
"description": "Transfer call to a specified destination",
"parameters": {}
}
}
],
"destination": "555-1234"
}'
```

Expected Response:

Immediately returns a function call payload that instructs Vapi to transfer the call to the specified destination.

c. Custom Tool Calling (processOrder)

```
curl -X POST https://custom-llm-url/chat/completions/custom-tool \
  -H "Content-Type: application/json" \
  -d '{
"message": {
"toolCallList": [
{
"id": "12345",
"function": {
"name": "processOrder",
"arguments": {
"param": "value"
}
}
}
]
}
}'
```

Expected Response:

```
{
  "results": [
{
```

```
        "toolCallId": "12345",
        "result": "CustomTools With CustomLLM Always Works"
        }
 ]
}
```

Integrating Tools with Vapi

After testing locally, integrate your Custom LLM with Vapi. Choose the configuration that fits your needs.

a. Without Tools (Response Generation Only)

```
curl -X PATCH https://api.vapi.ai/assistant/insert-your-assistant-id-here \
        -H "Authorization: Bearer insert-your-private-key-here" \
        -H "Content-Type: application/json" \
        -d '{
 "model": {
        "provider": "custom-llm",
        "model": "gpt-4o",
        "url": "https://custom-llm-url/chat/completions",
        "messages": [
        {
        "role": "system",
        "content": "[TASK] Ask the user if they want to transfer the call; if not, continue chatting."
        }
        ]
 },
 "transcriber": {
        "provider": "azure",
        "language": "en-CA"
 }
}'
```

b. With Tools (Including transferCall and processOrder)

```
curl -X PATCH https://api.vapi.ai/assistant/insert-your-assistant-id-here \
        -H "Authorization: Bearer insert-your-private-key-here" \
        -H "Content-Type: application/json" \
```

```
-d '{
  "model": {
    "provider": "custom-llm",
    "model": "gpt-4o",
    "url": "https://custom-llm-url/chat/completions",
    "messages": [
    {
    "role": "system",
    "content": "[TASK] Ask the user if they want to transfer the call; if they agree, trigger the transferCall tool; if not, continue the conversation. Also, if the user asks about the custom function processOrder, trigger that tool."
    }
    ],
    "tools": [
    {
    "type": "transferCall",
    "destinations": [
    {
    "type": "number",
    "number": "+xxxxxx",
    "numberE164CheckEnabled": false,
    "message": "Transferring Call To Customer Service Department"
    }
    ]
    },
    {
    "type": "function",
    "async": false,
    "function": {
    "name": "processOrder",
    "description": "it's a custom tool function named processOrder according to vapi.ai custom tools guide"
    },
    "server": {
    "url": "https://custom-llm-url/chat/completions/custom-tool"
    }
    }
```

```
        ]
  },
  "transcriber": {
        "provider": "azure",
"language": "en-CA"
  }
}'
```

Introduction to Workflows

title: Introduction to Workflows subtitle: >- Break down AI conversations into a visual workflow made up of discrete steps ("nodes") and branches between them ("edges"). slug: workflows

Workflows is now available to all Vapi users in Open Beta on [the dashboard here](https://dashboard.vapi.ai/workflows). Start building more reliable and structured conversational AI today.

Workflows is a new way to build conversational AI. It allows you to break down AI conversations into discrete steps, and then orchestrate those steps in a way that is easy to manage and modify.

Creating Your First Workflow

Begin by creating an assistant on the Assistants page and providing the required information, such as the assistant's name and capabilities. Once your assistant is set up, switch the model provider to Vapi and click "Create Workflow" when prompted. A modal will appear offering you the option to create a new workflow or attach to an existing one. Choose the appropriate option to proceed to the Workflow Builder.

Step 1: Create an Assistant Visit the Assistants page. Create a new assistant, give it a name, and select a voice and transcription model of your choice.

Step 2: Switch Provider to vapi

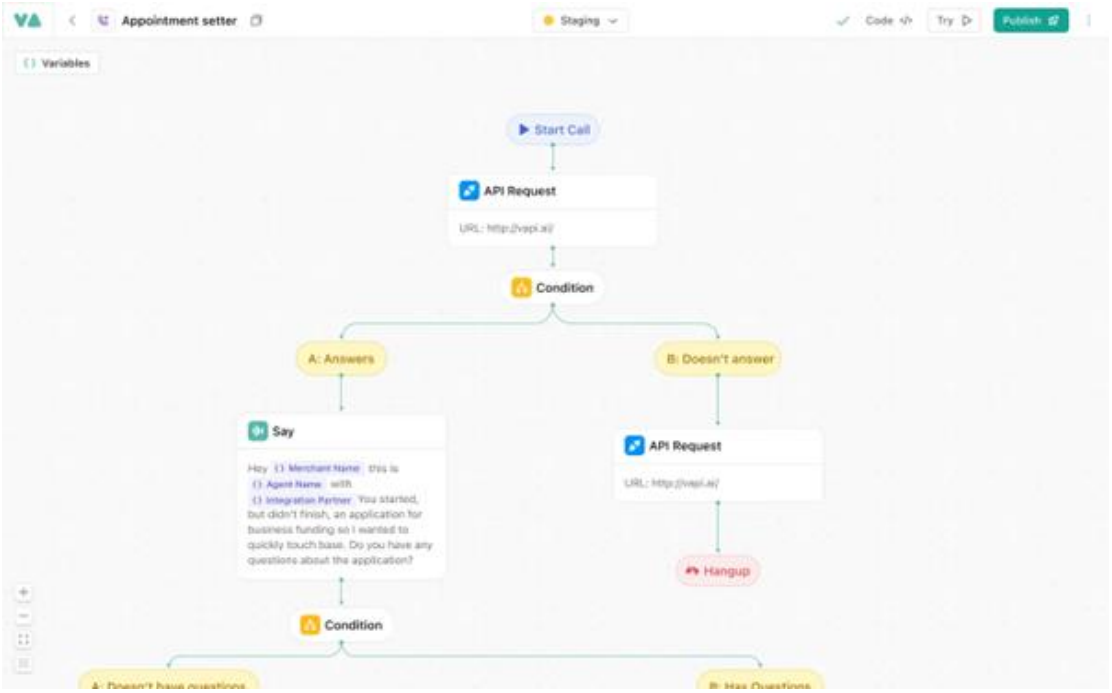
Under the "Model" section, switch the "Provider" field to `vapi`.

Step 3: Create a New Workflow or Attach an Existing One

Click the "Create Workflow" button. A prompt will appear asking you to create a new workflow by entering a unique title, or attach to an existing workflow.

Step 4: Build Your Workflow

In the Workflow Builder, you will see a "Start" call node. Click the + button at the bottom of this node to select your first **node**. Use the + button to add further steps as needed.



Step 5: Create Connections

To create new connections between nodes, drag a line from one step's top connection dot to another step's bottom dot, forming the logical flow of the conversation.

Tips for Building Workflows

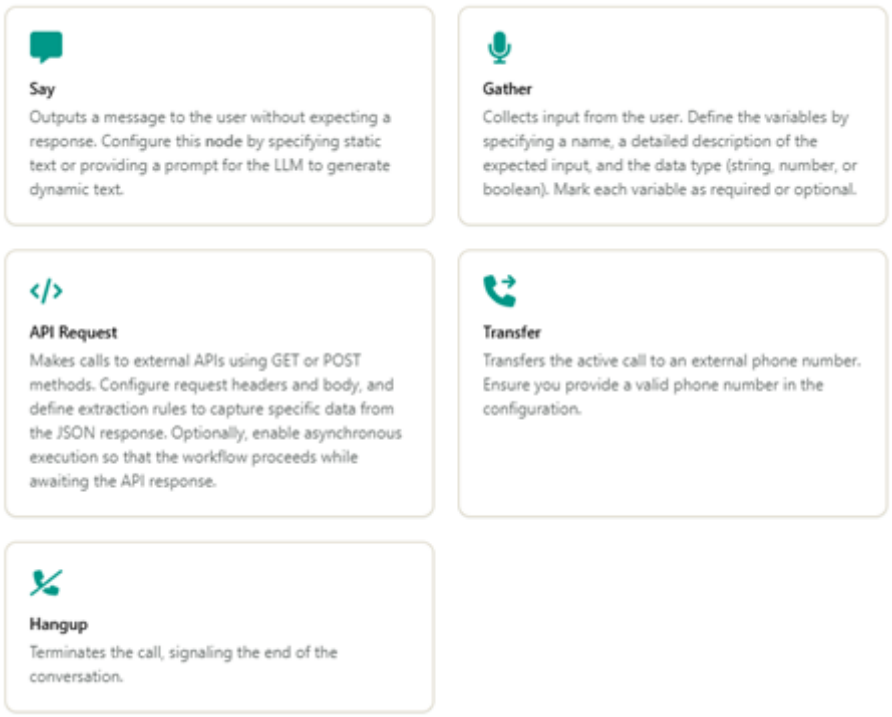
- **Deleting Nodes and Edges:** Click on any node or edge and press Backspace to delete it.
- **Attaching Nodes:** Attach a node to another by drawing a line from the top of one node to the bottom of another node.
- **Save Requirements:** A workflow cannot be saved until every node is connected and configured. The system will not allow saving with any dangling nodes.
- **Creating Conditionals:** To create conditionals, first add a condition node. Then, attach nodes for each branch by clicking the "Logic" tag on the connecting edges to set up the conditions.

Please let us know about any bugs you find by [submitting a bug report](https://roadmap.vapi.ai/bug-reports). We also welcome feature requests and suggestions - you can [submit those here](https://roadmap.vapi.ai/feature-requests). For

discussions about workflows and our product roadmap, please [join our Discord community](https://discord.com/invite/pUFNcf2WmH) to connect with our team.

Nodes

Workflows break down your AI voice agent's behavior into discrete, manageable nodes. Each node encapsulates a specific function within the conversation flow. Detailed configuration options let you tailor each step to your requirements. The available nodes are:
Outputs a message to the user without expecting a response. Configure this **node** by specifying static text or providing a prompt for the LLM to generate dynamic text. Collects input from the user. Define the variables by specifying a name, a detailed description of the expected input, and the data type (string, number, or boolean). Mark each variable as required or optional. Makes calls to external APIs using GET or POST methods. Configure request headers and body, and define extraction rules to capture specific data from the JSON response. Optionally, enable asynchronous execution so that the workflow proceeds while awaiting the API response. Transfers the active call to an external phone number. Ensure you provide a valid phone number in the configuration. Terminates the call, signaling the end of the conversation.



Say

Output a message to the user

Overview

The **Say** outputs a spoken message to the user without expecting a response. Use this task to provide instructions, notifications, or other information during a call.

Configuration

Exact Message: Specify the exact text that should be spoken.

Prompt for LLM Generated Message: Provide a prompt for the language model to generate the message dynamically.

Usage

Add **Say** when you need to deliver clear, concise information to your user. It works well as an initial greeting or when confirming actions.

Gather

Collect input from users

Overview

The **Gather** collects input from users during an interaction. It is used to capture variables that will be referenced later in your workflow.

Configuration

Define one or more variables to gather from the user with:

Name: A unique identifier.

Description: Details about the expected input to help the LLM get the right information from the caller.

Data Type: String, number, boolean, enum.

Required: Mark whether the variable is required or optional.

API Request

Interface with external APIs

Overview

The **API Request** enables your workflow to interact with external APIs. It supports both GET and POST methods, allowing the integration of external data and services.

Configuration

URL: Enter the endpoint to request.

Method: Specify the HTTP method (GET or POST).

Headers: Define each header with a key, value, and type.

Body Values: For POST requests, provide key, value, and type for each entry.

Output Values: Extract data from the API's JSON response:

Key: The key within the JSON payload to extract.
Target: The name of the output variable for the extracted value.
Type: The data type of the extracted value.
Mode: Toggle asynchronous execution with “run in the background” on or off.

Usage

Use the API Request to fetch information from an external API to use in your workflow, or to update information in your CRM or database.

1. **HTTP Configuration**: Enter the URL you want the workflow to call and select the HTTP method.
2. **API Metadata**: Specify key-value pairs for Headers and Body (for POST requests). This allows you to include authentication tokens and payload data with your API request.
3. **Define Output**: Set the expected output schema for the API response. This schema is used to extract variables that can be utilized later in your workflow.

For example, if the expected API response is

```
{
"name": "Jaden Dearsley",
"age": 25,
"isActive": true,
}
```

Define an output JSON schema for the API request with

```
{
...
"output": {
```

```
    "type": "object",
    "properties": {
      "name": {
        "type": "string",
        "description": "name of the user",
      },
      "age": {
        "type": "number",
        "description": "age of the user",
      },
      "isActive": {
        "type": "boolean",
        "description": "whether the user is active",
```

```
    }
  }
},
}
```

This will make **name**, **age**, and **isActive** available as variables for use throughout the rest of the workflow. To rename a variable, use the **target** option to specify a different variable name.

```
{
...
"output": {
"type": "object",
"properties": {
"name": {
"type": "string",
"description": "name of the user",
"target": "user_name" // renamed "name" to "user_name"
```

```
10  },
11  ...
12  }
13  },
14  }
```

Assistant

Speak to a configured assistant

Overview

The **Assistant** node enables a persistent conversation with one of your configured assistants.

Configuration

Select an Assistant Use the dropdown to select a pre-configured assistant.

Usage

Add **Assistant** nodes as leaf nodes to enable ongoing conversations with your configured assistants. Currently, Assistant nodes must be placed at the end of a workflow branch, as they don’t support transitioning to other nodes. This means the conversation with the assistant will continue until either the user ends the call or the assistant reaches a natural conclusion point.

The assistant will use its configured system prompt while inheriting the transcriber and voice settings from the global workflow assistant.

Transfer

Redirect calls to an external number

Overview

The **Transfer** transfers an active call to a designated external phone number. This enables routing calls to other departments or external contacts.

Configuration

Phone Number: Enter a valid destination number for the call transfer.

Usage

Use **Transfer** to escalate or redirect a call as needed. Ensure the provided phone number is formatted correctly for a smooth transfer experience.

Hangup

Terminate the call

Overview

The **Hangup** ends an active call, marking the conclusion of a conversation. It is typically used as the final step in your workflow.

Configuration

This task requires little to no configuration, as its purpose is solely to terminate the call.

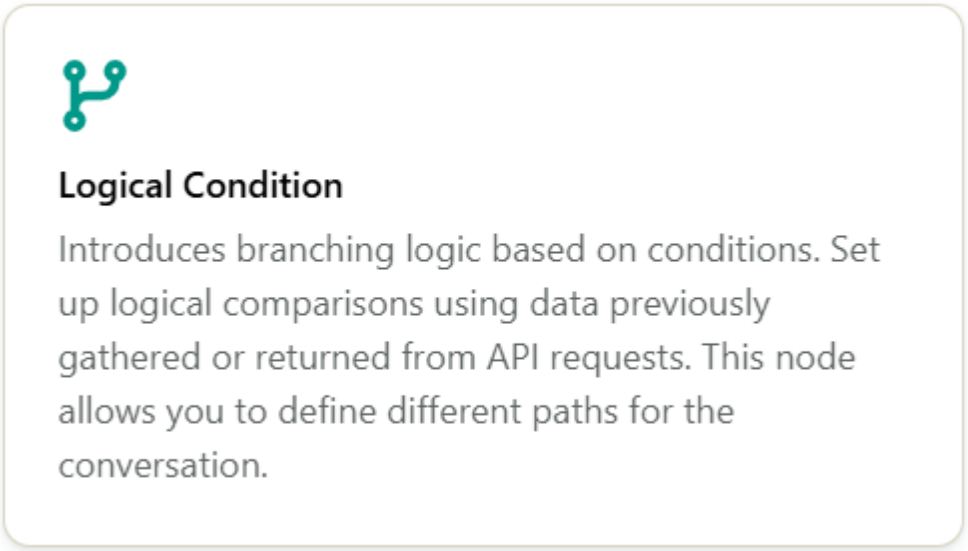
Usage

Place **Hangup** at the end of your workflow to close the conversation gracefully.

Edges

Edges allow you to create branching paths in your workflow based on different types of logic: Introduces branching logic based on conditions. Set up logical comparisons using data previously gathered or returned from API requests. This node allows you to define different paths for the conversation.

For detailed configuration instructions and advanced settings, please refer to our dedicated documentation pages for each task.



Logical Conditions

Branching logic for dynamic workflows

Overview

Logical Conditions enable you to create branching paths within your workflow. This feature allows your voice agent to decide the next steps based on data gathered earlier or retrieved via API calls.

Configuration

Condition Node: Start by inserting a condition node into your workflow.

Branch Setup: Attach one or more nodes to the condition node.

Logic Tag: Click the “Logic” tag on each connecting edge and select **Logic** from the **Condition Type** dropdown.

Condition Type: Choose between requiring ALL conditions to be met (AND logic) or ANY condition to be met (OR logic)

Logic Conditions Use the panel to define one or more rules or comparisons (e.g., equals, greater than) using variables collected from previous steps.

AI Conditions

Smart workflow branching powered by AI

Overview

AI Conditions use artificial intelligence to decide the next step in your workflow based on the conversation. Instead of using fixed rules, they can understand complex situations and make smart decisions in real-time.

How It Works

1. The AI looks at the conversation history and context
2. It makes a smart decision about which path to take, based on variables collected from Gather verbs and data returned from API requests.
3. Works alongside your existing rules for maximum flexibility

Configuration

Condition Node: Start by inserting a condition node into your workflow.

Branch Setup: Attach one or more nodes to the condition node.

AI Tag: Click on the connecting edge and choose **AI** from the **Condition Type** dropdown

AI Condition Use the input to define when the chosen branch should be taken.

Usage

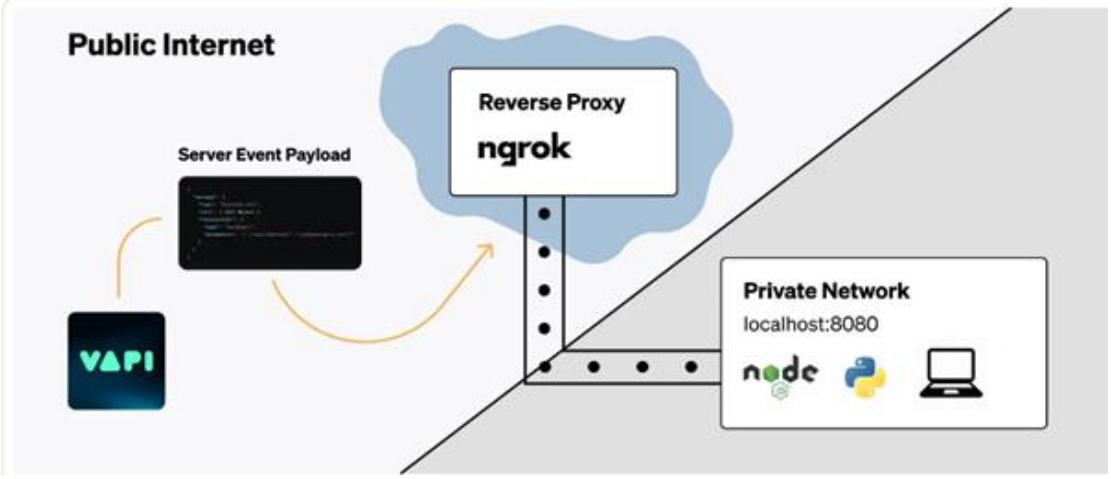
Use AI Conditions when you need:

To handle unclear or complex user responses

More flexibility than traditional rules can provide

More natural, human-like conversations

Developing Locally



The Problem

When Vapi dispatches events to a server, it must be able to reach the server via the open Internet.

If your API is already live in production, it will be accessible via a publicly known URL. But, during development, your server will often be running locally on your machine.

localhost is an alias for the IP address **127.0.0.1**. This address is called the "loopback" address and forwards the network request within the machine itself.

To receive server events locally, we will need a public address on the Internet that can receive traffic and forward it to our local machine.

Tunneling Traffic

We will be using a service called ngrok to create a secure tunnel to our local machine. The flow will look like the following:

1

Start Our API Locally

We will start our server locally so it is listening for http traffic. We will take note of the port our server is running on.

2

Start Ngrok Agent

We will use the **ngrok** command to start the ngrok agent on our machine. This will establish a connection from your local machine to ngrok's servers.

3

Copy Ngrok Forwarding URL

Ngrok will give us a public forwarding URL that can receive traffic. We will use this as a server URL during development.

4

Trigger Call Events

We will conduct normal calls on Vapi to trigger events. These events will go to the Ngrok URL & get tunnelled to our local machine.

We will see the event payloads come through locally & log them in our terminal.

Starting Our API Locally

First, ensure that your API is running locally. This could be a Node.js server, a Python server, or any other server that can receive HTTP requests.

Take note of the port that your server is running on. For example, if your server is running on port **8080**, you should be able to access it at **http://localhost:8080** in your browser.

Starting Ngrok Agent

Next we will install & run Ngrok agent to establish the forwarding pathway for Internet traffic:

1

Install Ngrok Agent CLI

Install the Ngrok agent by following Ngrok's quickstart guide. Once complete, we will have the **ngrok** command available in our terminal.

2

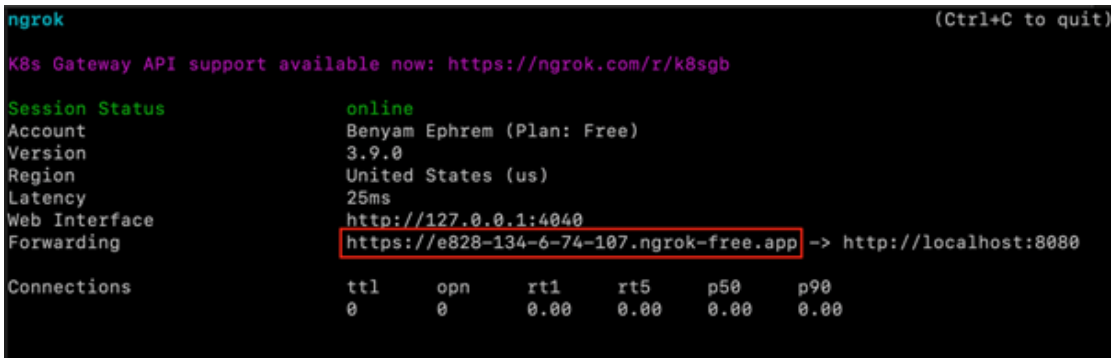
Start Ngrok Agent

Run the command **ngrok http 8080**, this will create the tunnel with Ngrok's servers.

Replace **8080** with the port your server is running on.

Copy Ngrok Forwarding URL

You will see an output from the Ngrok Agent CLI that looks like the following:



Terminal after running the 'ngrok' command forwarding to localhost:8080 — the 'Forwarding' URL is what we want.

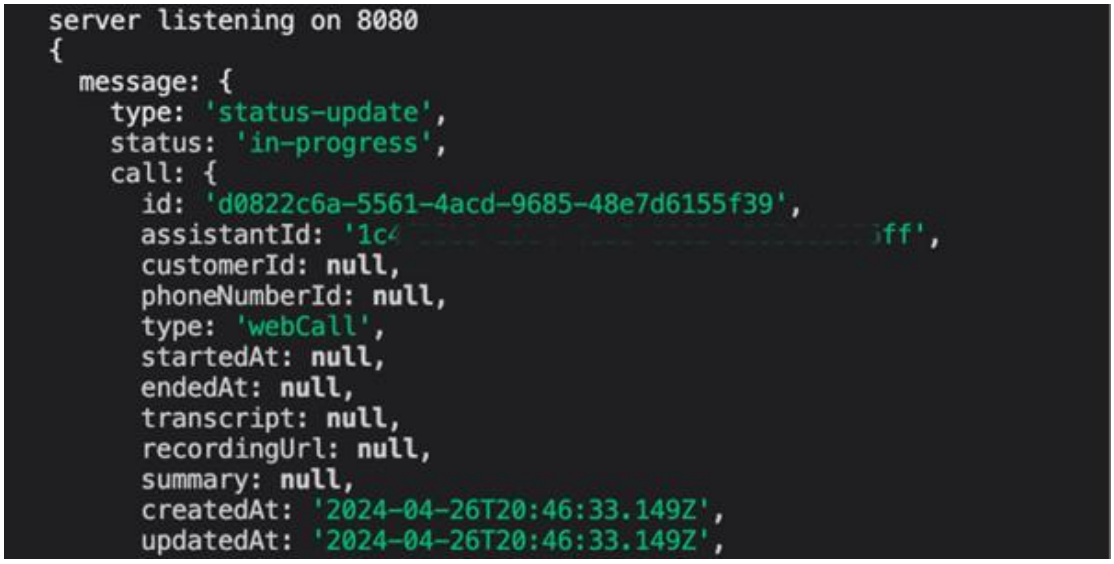
Copy this public URL that Ngrok provides. This URL will be accessible from the open Internet and will forward traffic to your local machine.

You can now use this as a server URL in the various places you can set server URLs in Vapi.

This URL will change every time that you run the **ngrok** command. If you'd like this URL to be the same every Ngrok session, look into static domains on Ngrok.

Trigger Call Events

We will now be able to see call events come through as **POST** requests, & can log payloads to our terminal.



Logging call events routed to our local environment.

Feel free to follow any of our quickstart guides to get started with building assistants & conducting calls.

Server Authentication

When configuring webhooks for your assistant, you can authenticate your server endpoints using either a secret token, custom headers, or OAuth2. This ensures that only authorized requests from Vapi are processed by your server.

Credential Configuration

Credentials can be configured at multiple levels:

1. **Tool Call Level**: Create individual credentials for each tool call
2. **Assistant Level**: Set credentials directly in the assistant configuration
3. **Phone Number Level**: Configure credentials for specific phone numbers
4. **Organization Level**: Manage credentials in the API Keys page

The order of precedence is:

1. Tool call-level credentials
2. Assistant-level credentials
3. Phone number-level credentials
4. Organization-level credentials from the API Keys page

Authentication Methods

Secret Token Authentication

The simplest way to authenticate webhook requests is using a secret token. Vapi will include this token in the **X-Vapi-Signature** header of each request.

Configuration

```
{
"server": {
"url": "https://your-server.com/webhook",
"secret": "your-secret-token"
}
}
```

Custom Headers Authentication

For more complex authentication scenarios, you can configure custom headers that Vapi will include with each webhook request.

This could include short lived JWTs/API Keys passed along via the Authorization header, or any other header that your server checks for.

Configuration

```
{
"server": {
"url": "https://your-server.com/webhook",
"headers": {
"Authorization": "Bearer your-api-key",
"Custom-Header": "custom-value"
```

```
}
}
}
```

OAuth2 Authentication

For OAuth2-protected webhook endpoints, you can configure OAuth2 credentials that Vapi will use to obtain and refresh access tokens.

Configuration

```
{
"server": {
"url": "https://your-server.com/webhook"
},
"credentials": {
"webhook": {
"type": "oauth2",
"clientId": "your-client-id",
"clientSecret": "your-client-secret",
"tokenUrl": "https://your-server.com/oauth/token",
```

```
    "scope": "optional, only needed to specify which scopes to request access for"
  }
}
```

OAuth2 Flow

1. Vapi makes a request to your token endpoint with client credentials
2. Your server validates the credentials and returns an access token
3. Vapi includes the access token in the Authorization header for webhook requests
4. Your server validates the access token before processing the webhook
5. When the token expires, Vapi automatically requests a new one

OAuth2 Token Response Format

Your server should return a JSON response with the following format:

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "token_type": "Bearer",
  "expires_in": 3600,
  "refresh_token": "tGzv3JOkF0XG5Qx2TlKWIA", // Optional
```

```
6   "scope": "read write" // Optional, only if scope was requested
7 }
```

Example error response:

```
1 {
2   "error": "invalid_client",
3   "error_description": "Invalid client credentials"
4 }
```

Security and privacy

GDPR Compliance

At Vapi, safeguarding your personal data is our top priority. In full alignment with the General Data Protection Regulation (GDPR), we maintain robust standards for data protection and privacy. This document provides an overview of our data processing practices, legal bases, data subject rights, and the security measures we employ—all designed to ensure that your data is managed with the utmost care.

Data Processing & Legal Bases

Our operations involve the secure processing of various types of personal data to enhance and deliver the Vapi service. We process information such as email addresses, names, phone numbers, physical addresses, usage statistics, and location data. The legal grounds underpinning this processing are:

Consent: Users voluntarily provide consent for non-essential data processing (e.g., location-based services and marketing communications). This consent can be withdrawn at any time.

Contractual Necessity: We process the data essential for fulfilling the services offered through Vapi, as detailed in our terms of service.

Legitimate Interests: Data is processed to improve service functionality, enhance security, and analyze usage patterns, provided that our legitimate interests do not override your rights.

Data Subject Rights

Vapi ensures that every user benefits from the robust rights granted by the GDPR. These rights include:

Right to Access: You can request and obtain a copy of your personal data.

Right to Rectification: If your data is inaccurate or incomplete, you can request corrections.

Right to Erasure (Right to be Forgotten): Under certain conditions, you can ask for your personal data to be deleted.

Right to Restrict Processing: You have the option to limit how your data is processed.

Right to Data Portability: You can obtain and transfer your data in a structured, commonly used format.

Right to Withdraw Consent: If your data processing is based on consent, you can withdraw it at any time.

Data Security Measures

We deploy a range of technical and organizational safeguards to protect your personal data from unauthorized access, alteration, disclosure, and destruction, including:

Encryption: Data is encrypted in transit and at rest.

Secure Server Configurations: Our infrastructure is optimized for enhanced security.

Access Controls: Strict controls ensure that only authorized personnel access sensitive data.

Regular Assessments: Security audits and penetration tests are routinely performed to identify and address vulnerabilities.

Third-Party Data Processors

To provide a best-in-class experience, Vapi partners with several reputable third-party providers, all of which comply with our GDPR standards. These include:

Analytics Tools:

Google Analytics

Cloudflare Analytics

Segment.io

Mixpanel (with opt-out options)

PostHog

CI/CD and Development Platforms:

GitHub

Payment Processors:

Stripe

Each of these providers is carefully selected and operates under strict data protection agreements to ensure that your data remains secure.

Transborder Data Transfers

In cases where personal data is transferred outside the European Union (primarily to the United States), we ensure that all transfers are governed by legally approved safeguards such as standard contractual clauses. These measures guarantee that your data receives the same level of protection, regardless of where it is processed.

Compliance Testing & Continuous Improvement

To reinforce our GDPR compliance, we conduct routine testing and audits including:

Penetration Testing: Confirming there are no critical vulnerabilities.

Compliance Audits: Verifying that our data processing practices adhere to GDPR standards.

Role-Based Access Control Tests: Ensuring that access to personal data is strictly limited to authorized personnel.

Data Breach Simulations: Evaluating the efficiency of our incident response plans.

User Consent Management Tests: Checking the ease and accuracy of obtaining or withdrawing user consent.

Data Recovery and Deletion Tests: Ensuring our backup systems and deletion protocols function as required.

These measures ensure that our data protection systems remain robust, up-to-date, and fully compliant with the ever-evolving data protection landscape.

Testing

Test Suites

End-to-end test automation for AI voice agents

Overview

Test Suite is an end-to-end feature that automates testing of your AI voice agents. Our platform simulates an AI tester that interacts with your voice agent by following a pre-defined script. After the interaction, the transcript is sent to a language model (LLM) along with your evaluation rubric. The LLM then determines if the interaction met the defined objectives.

Creating a Test Suite

Begin by creating a **Test Suite** that organizes and executes multiple test cases.

1

Step 1: Create a New Test Suite

Navigate to the **Test** tab in your dashboard and select **Test Suites**.

Click the **Create Test Suite** button.

2

Step 2: Define Test Suite Details

Enter a title for your **Test Suite**.

Select a phone number from your organization using the dropdown.

Make sure the phone number has an assistant assigned to it (if not, navigate to Phone Numbers tab to complete that action).

3

Step 3: Add Test Cases

Once your **Test Suite** is created, you will see a table where you can add test cases.

Click **Add Test** to add a new test case (up to 50 can be added).

4

Step 4: Configure Each Test Case

Script: Define how the testing agent should behave, including a detailed multi-step prompt to simulate how the customer should behave on the call.

Type: Set the type of the test. 'Chat' simulates a text conversation, which we recommend because it is faster. 'Voice' simulates a call so you can hear a voice recording of the two assistants talking to each other.

Rubric: List one or more questions that an LLM will use to evaluate if the interaction was successful.

Attempts: Choose the number of times (up to 5) the test case should be executed each time the **Test Suite** is run.

5

Step 5: Run and Review Tests

Click **Run Tests** to execute all test cases one by one.

While tests are running, you will see a loading state.

Upon completion, a table displays the outcomes with check marks (success) or x-marks (failure).

Click on a test row to view detailed results: a dropdown shows each attempt, the LLM's reasoning, the transcript of the call, the defined script, and the success rubric.

Test Execution and Evaluation

When you run a **Test Suite**, the following steps occur:

Simulation: An AI tester chats with or calls your voice agent, executing the pre-defined script.

Transcript Capture: The entire conversation is transcribed, capturing both the caller's behavior and your voice agent's responses.

Automated Evaluation: The transcript, along with your Success Criteria, is processed by an LLM to determine if the call was successful.

Results Display: Each test case outcome is shown with details. Clicking on a test case reveals:

The number of attempts made.

The LLM's reasoning for each attempt.

The complete transcript.

The configured script and rubric.

Example Test Case

Below are three example test cases to illustrate how you can configure detailed simulation scripts and evaluation rubrics.

Example : Billing Support

In this example, we will simulate a customer who is frustrated and calling about a billing discrepancy.

Script:

1. Express anger over an unexpected charge and the current bill appearing unusually high.
2. Try to get a detailed explanation, confirming whether an overcharge occurred, and understanding the steps for resolution.
3. End the call.

Rubric:

The voice agent acknowledges the billing discrepancy respectfully without dismissing the concern.

Voice Testing

Automated voice call testing for AI voice agents

Overview

Voice Test Suites enable you to test your AI voice agents through simulated phone conversations. Our platform connects two AI agents - your voice agent and our testing agent - on a real phone call, following your predefined scripts to evaluate performance under various scenarios.

How Voice Testing Works

1. **Simulation:** Our AI tester calls your voice agent and follows a script that simulates real customer behavior.
2. **Conversation:** Both AIs engage in a natural voice conversation, with the tester following your script guidelines.
3. **Recording:** The entire call is recorded and transcribed for evaluation.
4. **Assessment:** After the call, the transcript is evaluated against your rubric by a language model (LLM).

Benefits of Voice Testing

Natural Interaction: Test your voice agent in the most realistic scenario - actual phone calls.

Audio Quality Assessment: Evaluate not just responses but also voice clarity, tone, and cadence.

End-to-End Verification: Confirm that your entire voice pipeline works correctly from telephony to response.

Creating Voice Tests

You can create voice tests as part of a Test Suite:

1. Navigate to the **Test** tab and select **Test Suites**.
2. Create a new Test Suite or edit an existing one.
3. When adding tests, select **Voice** as the test type.
4. Define your script and success criteria as detailed in the Test Suites documentation.

Voice Test Limitations

Voice tests require more time to execute compared to chat tests.

Each test consumes calling minutes from your account.

Maximum call duration is limited to 15 minutes per test.

Applications of AI Call Assistant in the Real World

The AI Call Assistant developed using **Vapi AI** and **Make.com** has numerous impactful applications across various sectors. It simulates natural human conversation with real-time voice interactions, emotion awareness, and workflow automation — transforming how businesses handle voice communication.

1. Customer Service Automation

- 24/7 voice-based customer support without human agents
- Handles common queries, FAQs, and complaint resolution
- Emotion detection helps adapt tone (e.g., calm for angry customers)

2. Appointment Booking and Scheduling

- Automatically books, reschedules, or cancels appointments
- Integrates with Google Calendar, CRMs, and booking platforms via Make.com
- Sends follow-up confirmations via email/SMS

3. Lead Generation & Qualification

- Inbound and outbound AI calls to qualify leads
- Engages customers with personalized questions
- Captures lead data and pushes it to CRM tools automatically

4. Order Confirmation and Delivery Tracking

- Provides order status updates and delivery notifications
- Allows customers to modify or cancel orders via voice
- Real-time integration with e-commerce platforms

5. Hospitality & Reservations

- Handles restaurant, hotel, or event reservations
- Supports multilingual voice conversations for diverse customers
- Automates reminders and updates

6. Healthcare Applications

- Schedules doctor appointments or lab visits
- Sends voice-based medication reminders
- Collects patient feedback after appointments

7. Educational Institutes

- Assists with admission inquiries and interview scheduling
- Provides voice-based support to prospective students
- Collects feedback on courses and sessions

8. Call Center Optimization

- Reduces load on human agents by managing routine calls
- Hands off to humans only when needed
- Ensures consistent quality with emotion-aware responses

9. Surveys and Feedback Collection

- Conducts post-service surveys using natural voice conversations
- Adjusts tone and phrasing based on emotional cues
- Integrates feedback into analytics platforms

10. Banking and Financial Services

- Provides account balance, loan status, or transaction history over calls
- Verifies user identity through voice prompts

Conclusion

The AI Call Assistant developed using Vapi AI and Make.com represents a significant step forward in the automation of human-like voice interactions. By integrating cutting-edge technologies such as real-time speech recognition, LLM-driven natural language understanding, emotion detection, and dynamic voice synthesis, the system is capable of simulating human conversation in both inbound and outbound call scenarios.

This project has demonstrated how businesses can streamline processes such as appointment scheduling, lead generation, customer support, and feedback collection — all through AI-driven, context-aware, and emotion-sensitive voice interfaces. The low-latency performance and seamless integration with platforms like Make.com allow for real-time workflow automation, making the assistant not only conversational but also actionable.

Overall, the AI Call Assistant showcases the transformative potential of voice AI in enhancing customer experience, reducing operational costs, and enabling scalable, round-the-clock support across multiple domains.

Bibliography

Here is a sample APA-style bibliography with references relevant to your project:

1. Brownlee, J. (2020). *Machine Learning Mastery with Python: Understand Your Data, Create Accurate Models, and Work Projects End-To-End*. Machine Learning Mastery.
2. Vapi AI. (2024). *Developer Documentation*. Retrieved from https://docs.vapi.ai
3. OpenAI. (2023). *ChatGPT: Optimizing Language Models for Dialogue*. Retrieved from https://openai.com/chatgpt
4. Make.com. (2024). *Automation Platform Documentation*. Retrieved from https://www.make.com/en/help
5. Zhang, Q., Yang, L. T., Chen, Z., & Li, P. (2018). A survey on deep learning for big data. *Information Fusion*, 42, 146–157. https://doi.org/10.1016/j.inffus.2017.10.006
6. Jurafsky, D., & Martin, J. H. (2023). *Speech and Language Processing* (3rd ed.). Stanford University. Retrieved from https://web.stanford.edu/~jurafsky/slp3/
7. Chollet, F. (2021). *Deep Learning with Python* (2nd ed.). Manning Publications.
8. Panayotov, V., Chen, G., Povey, D., & Khudanpur, S. (2015). Librispeech: An ASR corpus based on public domain audio books. *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 5206–5210.