AI Claims Support Agent (Insurance)

Overview

This project implements a voice enabled AI Claims Support Assistant for Observe Insurance with the goal of automating routine inbound support interactions and improving customer experience. The purpose of this project is to demonstrate the ability to design, build, and integrate a conversational voice agent that balances natural dialogue, workflow control, backend data access, and reliability.

Use Cases

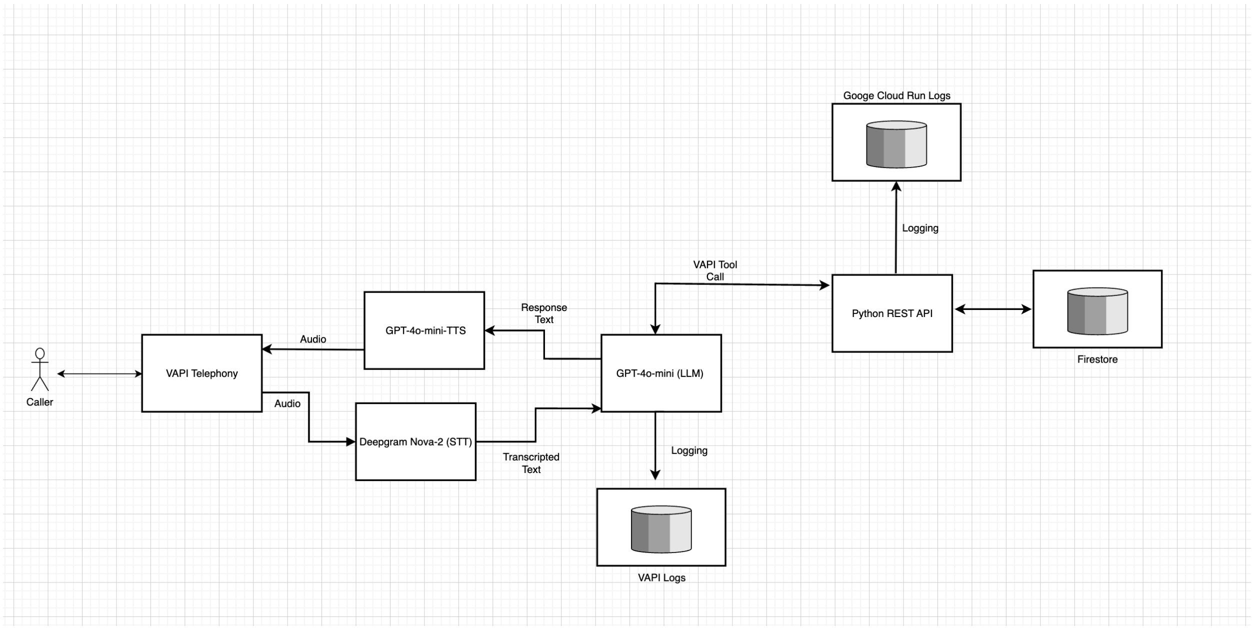
Use Case	Description	Outcome
Handle Incoming Support Call	The assistant answers the call, introduces itself, and explains what it can help with.	Caller understands purpose and begins interaction.
Authenticate Caller	The assistant requests and processes the caller's phone number, then verifies identity using backend claim data.	Caller identity is confirmed or a fallback/escalation path is triggered.
Provide Claim Status Information	If authenticated, the assistant retrieves and communicates the status of the caller's insurance claim.	Caller receives accurate claim information.
Support Common Questions	The assistant responds to general claim related FAQs, such as office hours or how to submit additional information.	Caller receives relevant information without needing a human agent.
Escalate When Needed	If the caller requests a representative or cannot be verified, the assistant transitions the call appropriately and ends with next steps.	Caller is informed of escalation and the process continues outside the automated flow.
Log Interaction Summary	After the call, the assistant writes a structured record including caller details (if authenticated), sentiment, summary, and timestamp.	Interaction data is captured for reporting, analysis, and future optimization.

Architecture Overview

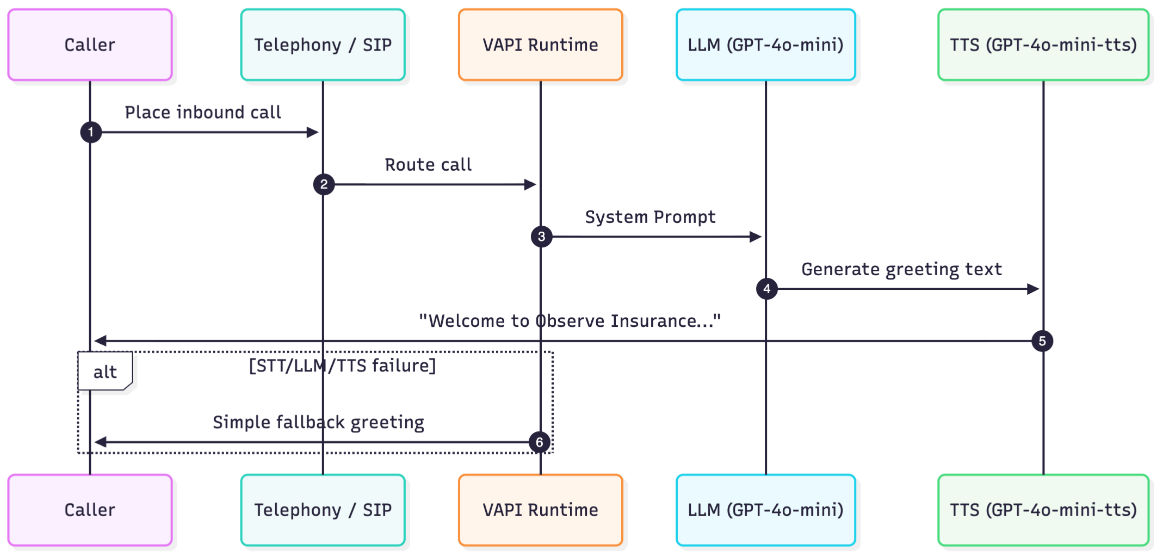
The architecture for this solution is centered around **VAPI** as the primary conversation and voice orchestration framework. Within VAPI the AI assistant uses a combination of models including **Deepgram Nova-2** for speech to text, **OpenAI GPT-4o-mini** for reasoning and tool calling, and **GPT-4o-mini-TTS** for text to speech output. These models work together inside VAPI's runtime to provide a responsive call experience. To support dynamic data retrieval and post call processing, I implemented a custom backend API written in Python using **FastAPI** as a framework. This service is deployed on **Google Cloud Run**. For the database, I integrated with **Firestore**, which stores both claim records and logged interaction history. This architecture separates conversational intelligence from business logic and data access, allowing for modularity and future scalability.

Diagrams

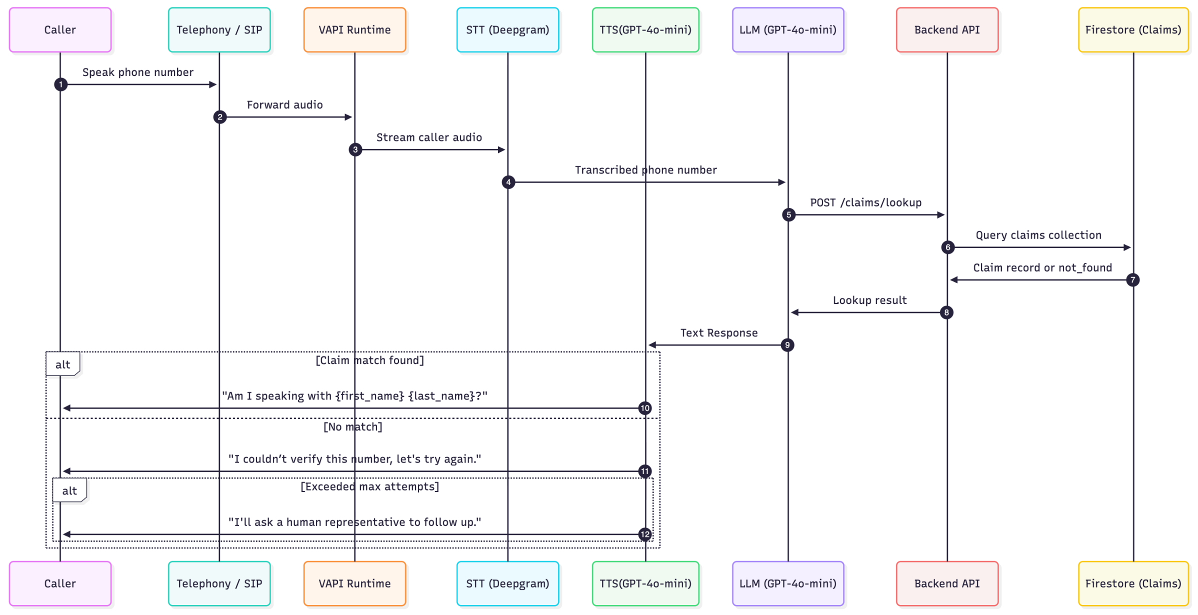
Architecture Diagram

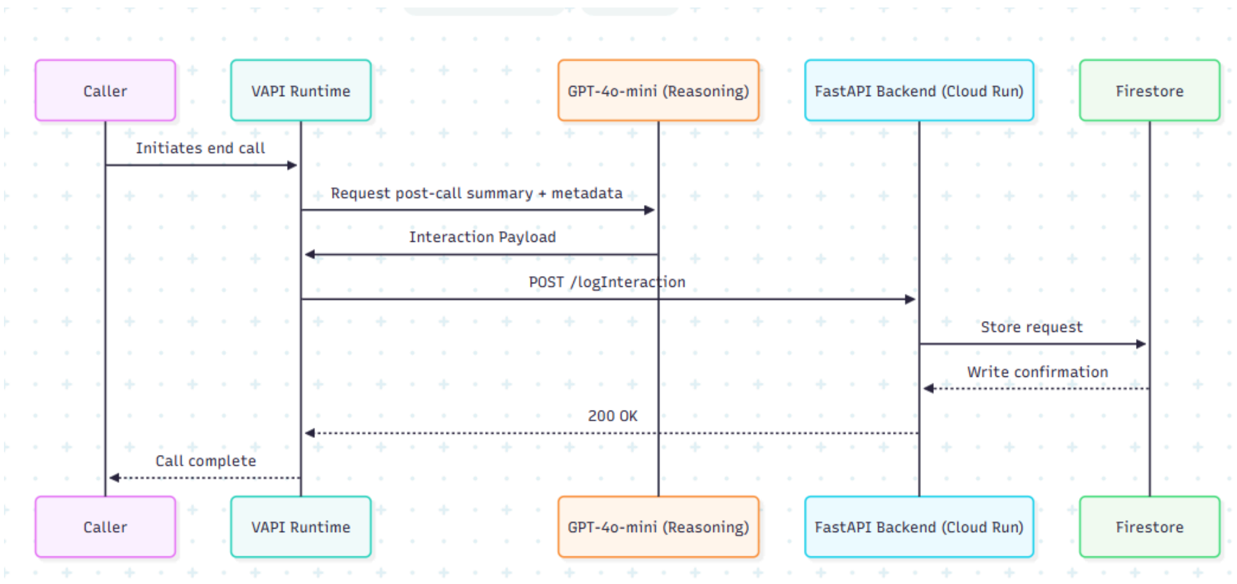


Greeting Flow



Phone Number Lookup


Caller
Telephony / SIP
VAPI Runtime
STT (Deepgram)
TTS(GPT-4o-mini)
LLM (GPT-4o-mini)
Backend API
Firestore (Claims)
1 Speak phone number
2 Forward audio
3 Stream caller audio
4 Transcribed phone number
5 POST /claims/lookup
6 Query claims collection
7 Claim record or not_found
8 Lookup result
9 Text Response
alt [Claim match found]
10 "Am I speaking with {first_name} {last_name}?"
[No match]
11 "I couldn't verify this number, let's try again."
alt [Exceeded max attempts]
12 "I'll ask a human representative to follow up."



Framework

VAPI Overview

Resource	Link
Phone calls	Phone calls | Vapi
Core Models	Core Models | Vapi
Orchestration Models	Orchestration Models | Vapi

VAPI is a real time voice AI platform designed to orchestrate live conversational agents without requiring custom telephony infrastructure or audio streaming logic. VAPI provides the telephony connection, model setup, and phone number generation right out of the box. VAPI also provides a unified runtime that manages the entire voice interaction loop, capturing audio from a caller, converting speech to text, invoking an LLM for reasoning and tool calls, generating spoken responses with text to speech, and then streaming that audio back to the caller.

VAPI Features

- **Telephony Integration:**
    - VAPI provides out of the box support for both inbound and outbound calls.

    - VAPI offers support to set up a phone number within their framework. This phone number comes set up with SIP routing and audio streaming capabilities out of the box.
- **LLM:**
    - VAPI also supports LLM orchestration right out of the box. Within VAPI's UI you have the ability to select which LLM provider and model you wish to use.
    - VAPI also gives you the ability to directly pass an opening message and system prompt.
- **STT/TTS:**
    - Similar to the LLMs, VAPI also supports STT/TTS orchestration out of the box. You have the ability to select your STT/TTS providers and models within their UI.
    - VAPI also handles the audio routing and streaming out of the box.
- **VAPI Tools:**
    - VAPI tools can be configured as functions your LLM can utilize upon receiving a prompt.
    - Using VAPI tools I was able to give my LLM model access to my API endpoints for searching customers and writing the post call interaction.
- **Logging**
    - VAPI also captures user transcripts and provides access to them within the dashboard.
    - VAPI also stores logs detailing the processes occurring during the interaction (including function calls and responses).

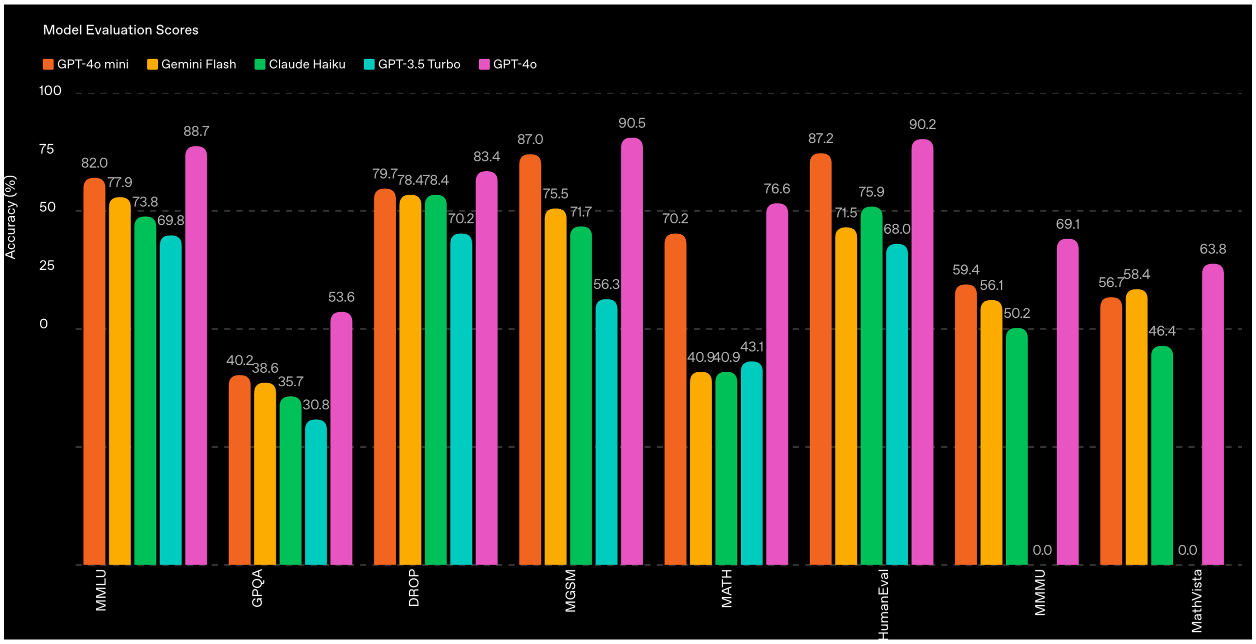
LLM Model

GPT-4o mini: advancing cost-efficient intelligence

Overview

The LLM I chose for this project was OpenAI's GPT-4o-mini. This model plays the role of understanding the user's intent, determining backend tools to utilize, and outputting a response to be spoken to the customer.

Below is a diagram showcasing GPT-40-mini's preformance:



Role

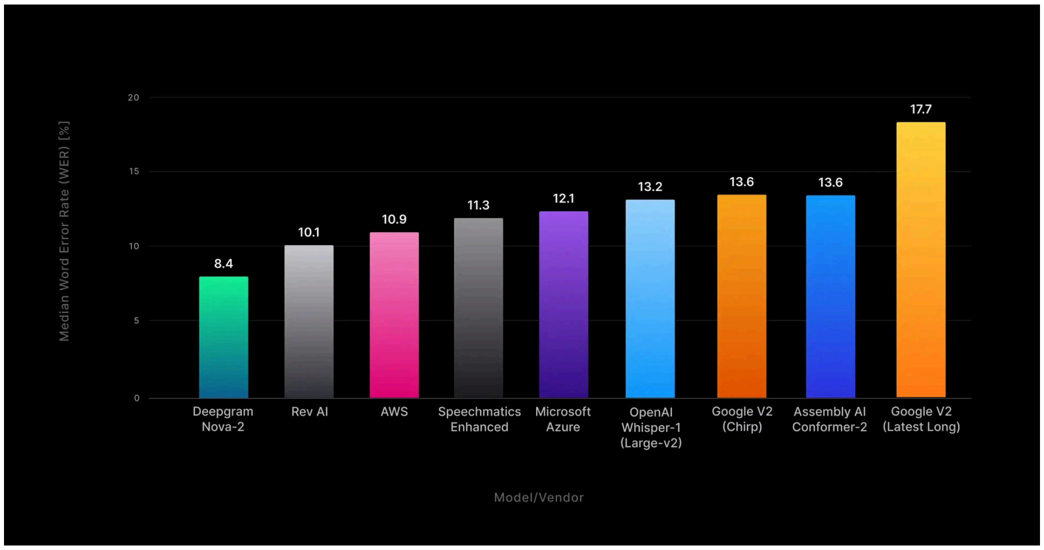
- In my project GPT-4o-mini has the ability to utilize two VAPI functions I configured (lookupClaim and logInteraction).
- The model handles when to trigger these tools along with conversational logic like FAQ responses and escalation.

STT Model

Introducing Nova-2: The Fastest, Most Accurate Speech-to-Text API

Overview

The STT model I chose to use for this project was Deepgram Nova-2. This model plays the role of converting the live caller's audio into text for GPT-4o-mini to process.



Role

- In my project Deepgram Nova-2 utilizes a websocket connection to continuously stream the caller's audio.
- This stream of audio is then transformed into text and is passed to GPT-4o-mini.

TTS Model

Overview

OpenAI Platform

GPT-4o-Mini-TTS: Steerable, Low-Cost Speech via Simple APIs

For my TTS model I chose to go with OpenAI's model GPT-4o-mini-TTS. This model is utilized by taking output from my LLM model and transforming it into speech.

Role

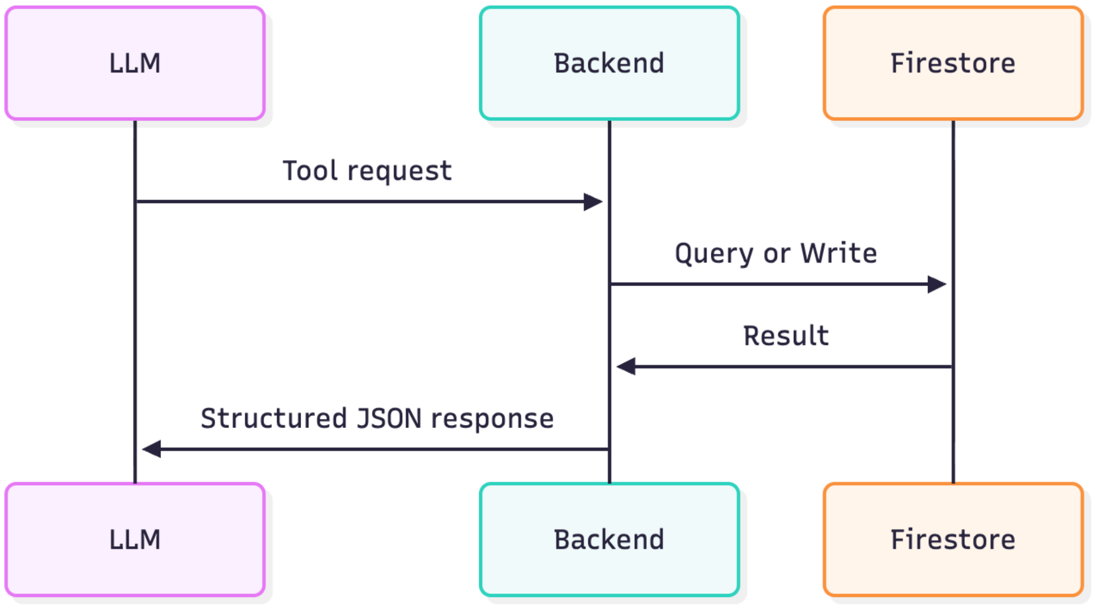
- GPT-4o-mini-TTS also utilizes a websocket connection like Deepgram to send audio to the caller.
- The model also provides a consistent tone and flow of pace with the user.

Backend Infrastructure

FastAPI

Overview

For the backend implementation I chose to implement a REST API using Python and FastAPI. For the data storage I used Firebase's Firestore NoSQL document DB. I also deployed a containerized version of my REST API using Google Cloud Run.



Role

- Claim Lookup
    - Receives a phone number from the AI agent via a VAPI tool call.
    - Queries Firestore for a matching claim record.

    - Returns the customer’s name and claim status information.
- Interaction Logging
    - Accepts structured call metadata: customer name (if authenticated), summary, sentiment, and timestamp.
    - Stores the interaction in the Firestore interactions collection for reporting and monitoring.

Justification

The technologies I selected for this solution were chosen based on performance, latency, modularity, scalability, and cost efficiency. Because this project simulates a real inbound claims support line, every component needed to support real time behavior, reliable function calls, and consistent conversational responses.

My decision to use **VAPI** as the conversational orchestration layer was driven by its ability to seamlessly integrate telephony, LLM reasoning, STT, TTS, and external tool execution within a single managed runtime. Rather than building or maintaining SIP connections, audio pipelines, and bidirectional model streaming manually, VAPI provides these capabilities out of the box while still allowing control of model selection. This modularity also ensures future extensibility (e.g., swapping STT models, or adding Twilio as a carrier). VAPI's ad hoc cost model ($0.05/minute) also made it an affordable choice.

For reasoning and function calling I selected **OpenAI GPT-4o-mini** due to its performance. GPT-4o-mini provides strong accuracy in structured task execution, reliable tool call behavior, and low latency suitable for real time bidirectional conversation. Its pricing model (approx. $0.15/million input tokens, $0.60/million output) makes it cost effective at scale while still delivering high reasoning performance.

For STT, I chose **Deepgram Nova-2** because of its low word error rate which I felt was very important for this use case, especially since we need to process input like phone numbers. A missed digit will lead to authentication failure, so accuracy mattered more than cost alone.

For TTS, I chose **OpenAI GPT-4o-mini-TTS** due to its low latency and natural sounding dialog flow. The low latency helps minimize pauses and helped maintain a human like conversational experience.

Finally, for the backend I decided to implement a **Python REST API using FastAPI as a framework.** This service was deployed on **Google Cloud Run. Firestore** was the database I selected to handle lookups and post call logging storage. This architecture is lightweight, and horizontally scalable, supporting different call volume without requiring manual infrastructure management. Firestore's flexible NoSQL schema also allows claims records and interaction logs to evolve without rigid migrations.

Collectively, these decisions ensured the solution was modular, reliable, scalable, and aligned with realistic cost constraints for an early stage voice automation use case, while still being adaptable to enterprise grade growth and future expansion.

How would this scale for Production

This architecture is designed so the prototype can evolve into a production system without requiring major rework. Each component can independently scale based on demand while maintaining loose coupling and predictable behavior. VAPI handles concurrency at the voice orchestration layer, meaning additional inbound call volume does not require any conversational or telephony reconfiguration.

Google Cloud Run scales automatically based on request traffic, ensuring that the backend remains responsive even under varying or unpredictable call loads. Since Firestore is fully managed and optimized for low latency document lookup at scale, it supports growth in both stored claim records and interaction logs without requiring schema migrations or performance tuning.

Additional enhancements could be introducing an API gateway for things like authentication using Google IAM.

Problem Solving/Debugging

Technical Challenge

One of the biggest technical challenges I encountered during this project was integrating VAPI’s function calling system with my backend API. Initially, calls from the assistant to my backend service would return errors like "No result returned" and "Missing arguments field." The issue stemmed from a mismatch between the expected request format from VAPI and the request schema my FastAPI endpoint was expecting. VAPI sends tool call arguments as a JSON string, while my API was expecting a structured JSON payload. This resulted in invalid payload errors, even though the tool name and invocation timing were correct.

Resolution

To resolve this, I added logs that would post to my Cloud Run instance. This allowed me to see the exact payload VAPI was sending to my backend service. Once I identified the formatting mismatch, I then implemented a serializer for incoming requests and outgoing responses. This allowed me to ensure the data coming into my system was valid and matched what my underlying functions were expecting.

Optimization Priorities

If I had one more week to work on the project I would prioritize improving the infrastructure of the application. One of the first things I would implement is an API gateway to work as an authentication and traffic routing layer for the backend API. This prevents direct exposure of the backend and creates a controlled boundary for external integrations.

A second area of improvement would be **model workflow optimization**. Currently, a single agent handles the entire conversation lifecycle. With more time, I would explore splitting that logic into dedicated task agents. For example one agent could focus solely on authentication, one on claim lookup logic, and one for post call summarization. This separation can reduce reasoning load, minimize hallucination risk, and improve reliability. Another addition could be implementing an MCP server to centrally manage backend capabilities for AI agents to utilize.

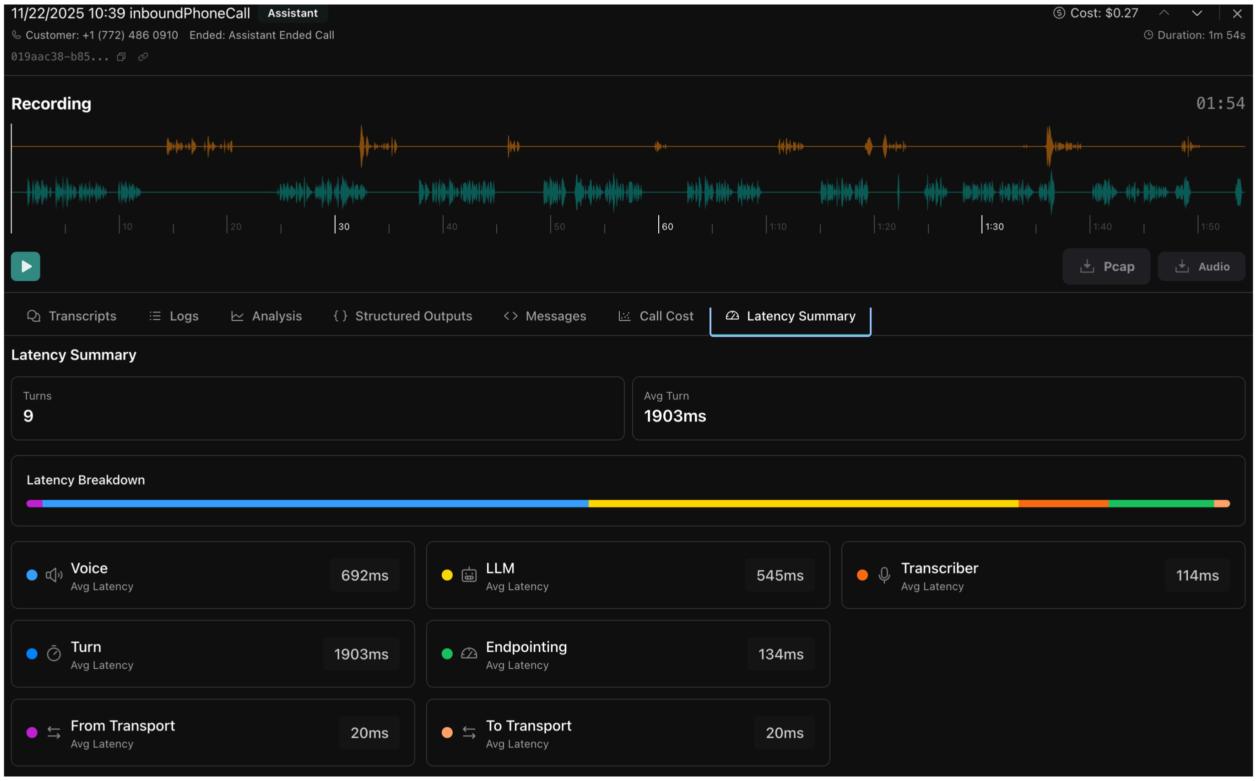
One final optimization I would make would be to have a dedicated store for FAQ information. Currently the content is embedded directly in the LLM system prompt which could be difficult to scale. Having the FAQ stored in a separate DB would make updates and versioning much more streamlined.

Data Metrics & Evaluation

Performance Measures

Category	Metric	Purpose
Customer Experience	Sentiment Score	Measures how callers feel during the interaction.
	Interrupt Frequency	Indicates whether responses are too long or unclear.
Operational Performance	Average Handle Time	Determines efficiency and helps benchmark progress toward faster resolution.
	Containment Rate	Primary ROI driver and measures automation success.
Technical Metrics	Word Error Rate	Tracks speech recognition quality.
	Response Latency	Can be used to track performance and detect possible bottlenecks.
	Tool Success / Failure Rate	Confirms backend integrations behave as intended.

Below are some latency metrics VAPI calculates:


11/22/2025 10:39 inboundPhoneCall Assistant

Cost: $0.27

Customer: +1 (772) 486 0910 Ended: Assistant Ended Call

Duration: 1m 54s

019aac38-b85...

Recording 01:54

Latency Summary

Turns	Avg Turn
9	1903ms

Latency Breakdown

Voice, Avg Latency	692ms	●
LLM, Avg Latency	545ms	●
Transcriber, Avg Latency	114ms	●
Turn, Avg Latency	1903ms	●
Endpointing, Avg Latency	134ms	●
From Transport, Avg Latency	20ms	●
To Transport, Avg Latency	20ms	●

Prompt Tuning

Using these metrics, I would be able to identify friction points and bottlenecks within the system and I would be able to utilize them to improve the performance and ROI. The goal would be to create a feedback loop where real call data directly informs prompt refinement, model configuration, and backend logic.

For example:

- Containment Rate:
  - If containment rate were to drop, I would review transcripts to identify where callers disengage or request human help. If a specific FAQ or claim status response repeatedly leads to escalation, I would try to refine that version of the prompt, or even look to FAQ data for content accuracy.
- Average Handle Time:
  - If the average handle time increased, I would evaluate whether responses were too verbose, whether the assistant requested unnecessary

    - clarification, or whether tool calls added latency. Improvements could include shortening phrasing, improving confirmation logic, or even diving into backend code implementation (query logic, caching).
- Word Error Rate:
    - If the word error rate were to increase, I would focus on possibly switching to a more optimized model for this industry. Possibly models that are more optimized for noisy environments. And if I have access to the model, maybe even looking to add bias for digits.

Containment Drop

To diagnose the issue, I would first filter transcripts for escalated calls and review what led to the handoff or extended handle time. By comparing these moments against expected behavior, I can identify whether the problem stems from unclear responses, misinterpreted audio, repeated verification loops, etc. This would allow me to isolate the issue and allow me to further target where to make the correction.