# AutoCally

**Titolo del progetto:** AutoCally
**Alunno/a:** Alexandru Ciobanu
**Classe:** Info 4BB
**Anno scolastico:** 2024/2025
**Docente responsabile:** Fabio Piccioni

# 1 Introduzione

## 1.1 Informazioni sul progetto

- Titolo Progetto: AutoCally
- Allievi: Alexandru Ciobanu I4BB,
- Docente responsabile: Fabio Piccioni, Fabio Trevito
- Scuola: Arti e Mestieri Trevano, sezione Informatica
- Data di inizio e fine: 27.01.2025 – 4.04.2025
- Data di presentazione: 14.04.2025 - 17.04.2025

**Abstract**

*As the demand for efficient customer support solutions grows, new methods must be developed to create intelligent voice agents quickly and effectively. For this project, a new platform called the Voice AI Agent Builder is created. This platform allows users to design virtual assistants that can answer phone calls, manage appointments, and store conversation logs.*
*With this innovative tool, businesses can build voice agents capable of natural interactions, from simple booking tasks to more complex customer support conversations. Traditionally, managing customer calls would require dedicated staff and significant time. With AutoCally, a single agent can handle numerous interactions seamlessly, significantly reducing workload and improving efficiency.*

## 1.2 Scopo

Lo scopo di questo progetto è sviluppare una piattaforma per la creazione di agenti vocali intelligenti dedicati al supporto clienti. Gli utenti potranno costruire agenti vocali personalizzati, in grado di rispondere alle chiamate, organizzare appuntamenti e salvare i log delle conversazioni in modo automatico.

La piattaforma offrirà strumenti intuitivi per configurare il comportamento dell'agente, scegliere la voce e stimare i costi delle interazioni. Gli agenti potranno inoltre integrare funzioni aggiuntive, come la gestione dei calendari e l’uso di strumenti esterni per migliorare l’efficienza delle risposte.

Inoltre, il progetto permetterà di esplorare l’utilizzo di modelli linguistici avanzati e tecnologie di sintesi vocale, offrendo un’esperienza pratica nella creazione e gestione di soluzioni vocali automatizzate

# 2 Analisi

## 2.1 Analisi del dominio

Attualmente, molte aziende si affidano a personale umano o a sistemi di risposta automatica molto semplici per gestire le chiamate di supporto e organizzare appuntamenti. Questi sistemi, tuttavia, sono spesso lenti, richiedono un significativo impegno umano e non sempre offrono un'esperienza soddisfacente per il cliente. Inoltre, molte aziende non dispongono di strumenti che consentano di raccogliere e analizzare automaticamente le conversazioni, o di gestire in modo integrato l'interazione con calendari e sistemi di gestione. Esistono dei prodotti simili a questo progetto già esistenti come Vapi.
Gli utenti principali di questo prodotto sono le piccole e medie imprese, come studi professionali, saloni, bar e altre attività che necessitano di un sistema di supporto clienti semplice ed economico
La piattaforma è pensata per essere intuitiva, quindi non è necessario un background tecnico per utilizzare gli strumenti di creazione degli agenti vocali.

## 2.2 Analisi e specifica dei requisiti

| **ID: REQ-001** | |
|---|---|
| **Nome** | Login, registrazione |
| **Priorità** | 0.7 |
| **Versione** | 1.0 |
| **Note** | Un utente per poter utilizzare il sito deve effettuare il login |

| **ID: REQ-002** | |
|---|---|
| **Nome** | Costruttore agenti vocali |
| **Priorità** | 0.6 |
| **Versione** | 1.0 |
| **Note** | Un agente si costruisce mediante un prompt, delle istruzioni per il suo comportamento, oltre a ciò ci saranno delle impostazioni tecniche per il LLM utilizzato. |
| **Sotto requisiti** | |
| **001** | Scelta voce |
| **002** | Dashboard costo al minuto |
| **003** | Dashboard latenza |

| ID: REQ-003 | |
|---|---|
| **Nome** | Chiamate |
| **Priorità** | 1 |
| **Versione** | 1.0 |
| **Note** | L’agente vocale risponde alle chiamate. |
| **Sotto requisiti** | |
| **001** | Salvataggio conversazione mediante trascrizione |
| **002** | Salvataggio audio |
| **003** | Log azioni dell’agente |
| **004** | Chiamate ricevute |
| **005** | Chiamate all’esterno |

| ID: REQ-004 | |
|---|---|
| **Nome** | Importazione numero Twilio |
| **Priorità** | 1 |
| **Versione** | 1.0 |
| **Note** | Per poter utilizzare le chiamate telefoniche verrà utilizzato Twilio |

| ID: REQ-005 | |
|---|---|
| **Nome** | Knowledge base |
| **Priorità** | 1 |
| **Versione** | 1.0 |
| **Note** | Sarà possibile memorizzare il contenuto di siti web, di testo e di documenti. Queste conoscenze serviranno agli agenti vocali. |

| ID: REQ-006 | |
|---|---|
| **Nome** | Tools |
| **Priorità** | 0.8 |
| **Versione** | 1 |
| **Note** | Durante la chiamata l'agente vocale potrà utilizzare degli strumenti aggiuntivi |
| **Sotto requisiti** | |
| **001** | Calendario |
| **002** | Notepad |
| **003** | Chiamate REST a servizi esterni |

| ID: REQ-007 | |
|---|---|
| **Nome** | Implementazione pagamenti |
| **Priorità** | 0.5 |
| **Versione** | 1 |
| **Note** | Sistema di pagamento con Stripe per gestione abbonamenti mensili |

## 2.3 Use case

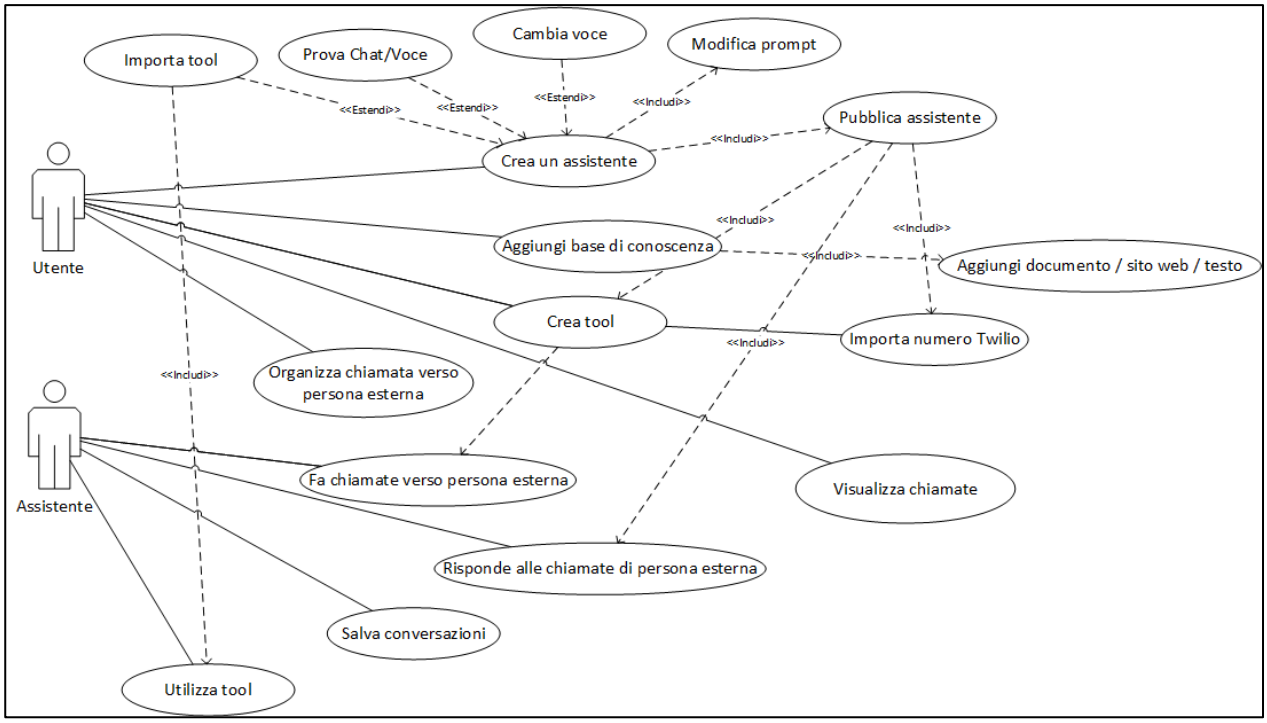



Figura 1 - Use Case

Come si può notare da questo schema, l'utente ha la possibilità di creare e personalizzare un assistente virtuale eseguendo diverse operazioni. Tra queste, può creare un assistente, modificarne la voce, testare le funzionalità di chat o voce e personalizzarne i prompt. Inoltre, può pubblicare l'assistente e arricchirlo con una base di conoscenza, aggiungendo documenti, siti web o testi, e associando un numero Twilio per gestire le chiamate.

L'utente ha anche la possibilità di creare e importare tool per espandere le funzionalità dell'assistente.

L'assistente, una volta configurato, può effettuare e organizzare chiamate verso persone esterne, rispondere alle chiamate in arrivo e salvare le conversazioni. Inoltre, può visualizzare lo storico delle chiamate e utilizzare i tool messi a disposizione dall'utente.

## 2.4 Pianificazione

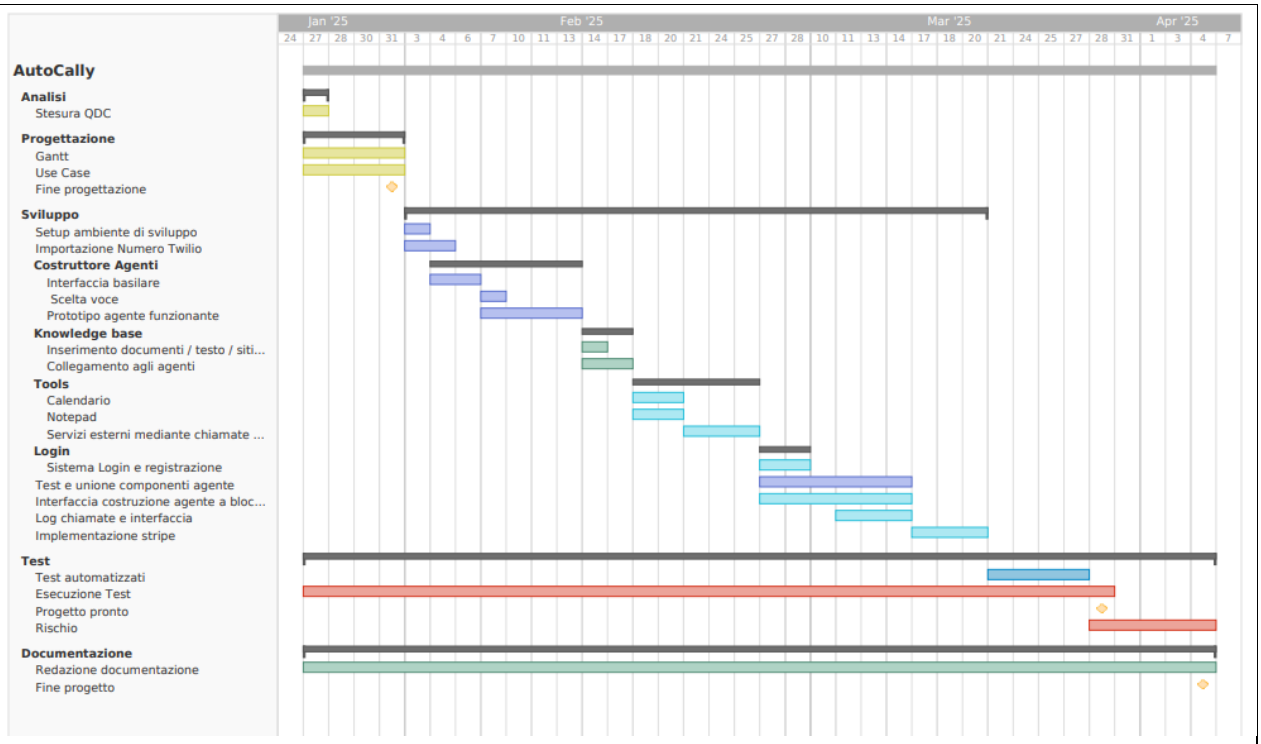



Figura 2 - Gantt.

## 2.5 Analisi dei mezzi

Per questo progetto si ha a disposizione un PC scolastico, su cui viene eseguita una macchina virtuale Linux Mint, su di essa girano dei container Docker.

### 2.5.1 Software

- Cursor Editor 0.44.11, come editor di testo
- Mozilla Firefox 135.0.1
- Gitlab, come Repository online
- Microsoft Visio, per fare lo Use Case
- Microsoft Teams, per le comunicazioni
- Microsoft Word, per redigere la documentazione, il QdC e il diario
- Trello.com, per la gestione dei task

### 2.5.2 Hardware

Computer utilizzato come host delle macchine virtuali:

- Processore: Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz
- RAM: 32GB
- Scheda Video: NVIDIA GeForce RTX 2060
- SSD: 512GB
- 2x Display 1920x1080 60H

## 3 Progettazione

### 3.1 Design dell'architettura del sistema

Si è progettato questo sistema per poter offrire un'esperienza completa agli utenti.
Gli utenti accedono all'applicativo tramite un'interfaccia grafica intuitiva e moderna tramite browser.
Si collegano al frontend tramite Nginx, che fa da web server ed agisce da reverse proxy verso il backend in Python.

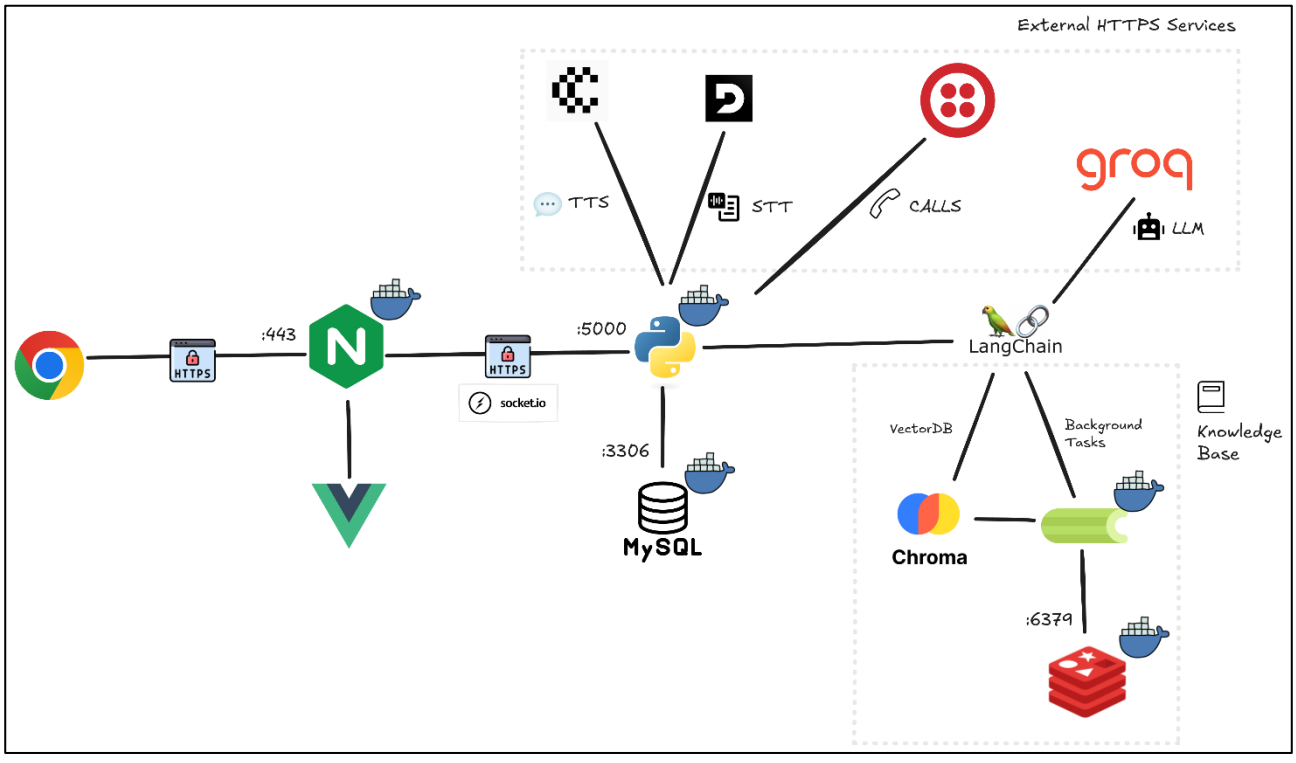



Figura 3 - Architettura

## 3.2 Design dei dati e database

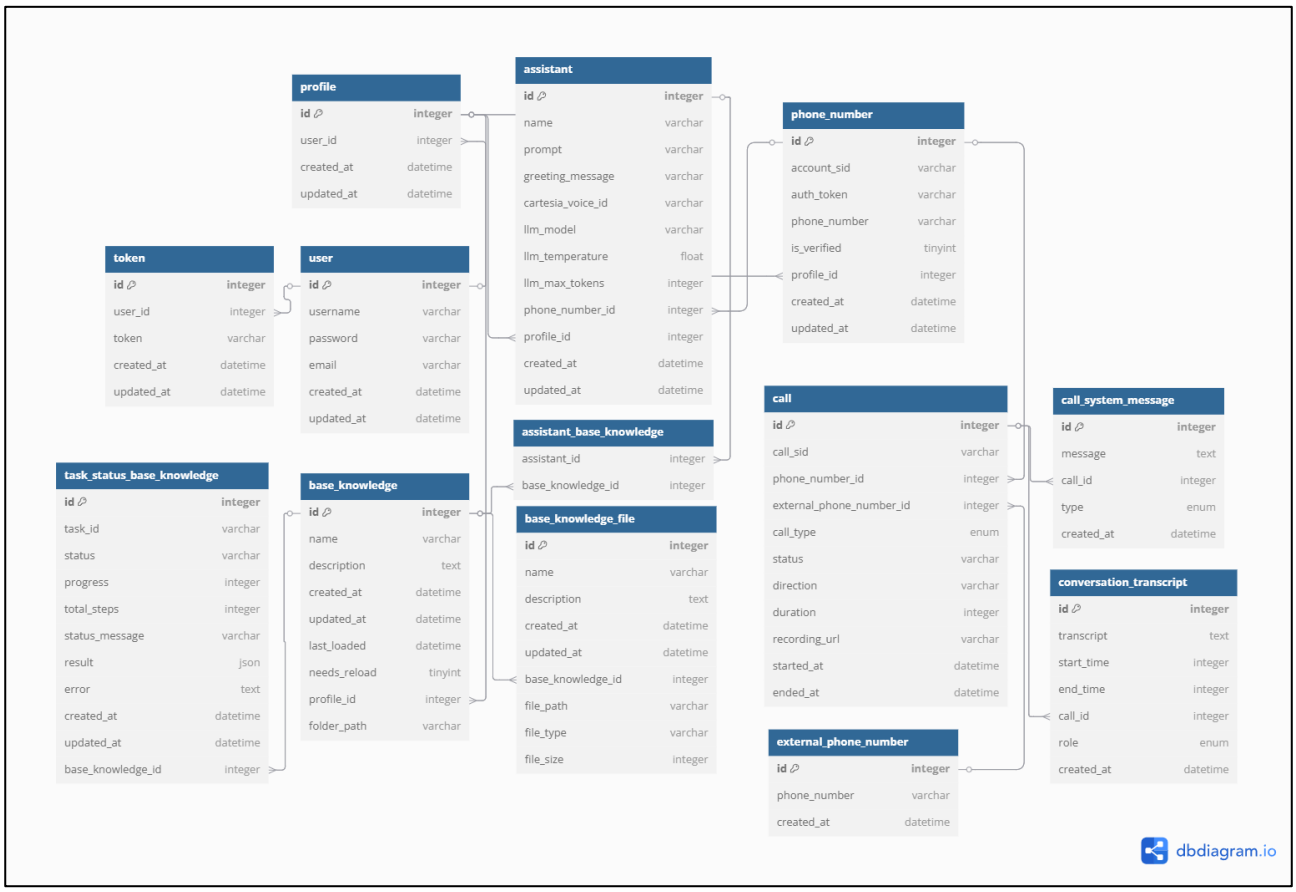



Figura 4 - Schema ER

**1. Tabella user**
Contiene le informazioni di base sugli utenti, come l'ID, nome utente, password, email e timestamp di creazione e aggiornamento. È la tabella principale che rappresenta gli utenti del sistema.

**2. Tabella profile**
Ogni utente può avere un profilo, che è collegato tramite user_id. Contiene dati di profilo come l'ID del profilo e i timestamp di creazione e aggiornamento. Un profilo rappresenta una sezione estesa dell'utente, che potrebbe includere informazioni personalizzate.

**3. Tabella phone_number**
Gestisce i numeri di telefono legati ai profili. Ogni numero è associato a un profile_id e può essere verificato tramite il campo is_verified. Contiene anche le credenziali per l'autenticazione (come account_sid e auth_token), insieme a timestamp di creazione e aggiornamento.

**4. Tabella assistant**
Definisce gli assistenti virtuali che possono essere associati a profili e numeri di telefono. Ogni assistente ha un nome, un messaggio di saluto, un prompt e parametri specifici per il modello di linguaggio (come llm_model, llm_temperature, ecc.). Un assistente è collegato a un numero di telefono tramite phone_number_id e a un profilo tramite profile_id.

**5. Tabella base_knowledge**

Contiene la conoscenza di base associata ai profili, che può includere descrizioni, file e altre risorse. La tabella include campi per gestire il percorso della cartella (folder_path), la necessità di ricaricare la conoscenza (needs_reload) e altre informazioni di stato.

**6. Tabella assistant_base_knowledge**
Associa gli assistenti con la conoscenza di base. Ogni assistente può avere una o più conoscenze di base caricate.

**7. Tabella base_knowledge_file**
Gestisce i file di conoscenza associati a un base_knowledge_id. Ogni file ha un nome, descrizione, percorso del file (file_path), tipo di file e dimensione. Può contenere risorse legate alla conoscenza di base.

**8. Tabella call**
Gestisce le chiamate telefoniche effettuate dal sistema, inclusi i numeri di telefono associati (phone_number_id e external_phone_number_id), tipo di chiamata, stato, durata e registrazione. La tabella contiene anche i timestamp di inizio e fine della chiamata.

**9. Tabella call_system_message**
Contiene i messaggi di sistema associati alle chiamate. Ogni messaggio è collegato a una chiamata tramite call_id e ha un tipo (ad esempio, errore, avviso) e un testo del messaggio.

**10. Tabella conversation_transcript**
Gestisce la trascrizione delle conversazioni telefoniche. Ogni trascrizione è collegata a una chiamata e include il testo trascritto, i tempi di inizio e fine, il ruolo della persona che parla e il timestamp di creazione.

**11. Tabella external_phone_number**
Contiene numeri di telefono esterni non associati direttamente ai profili o agli assistenti, ma collegati alle chiamate tramite call.external_phone_number_id.

**12. Tabella task_status_base_knowledge**
Monitora lo stato dei task associati alla conoscenza di base. Ogni task è identificato tramite task_id e ha un stato, progresso e messaggio di stato, oltre ai risultati o errori associati.

**13. Tabella token**
Gestisce i token di accesso per gli utenti. Ogni token è associato a un user_id e ha un timestamp di creazione e aggiornamento.

**14. Tabella tool**
Contiene informazioni su strumenti personalizzati che possono essere utilizzati nel sistema. Ogni strumento ha una definizione della sua funzione in formato JSON e timestamp di creazione e aggiornamento.

**Relazioni principali:**

- **user e profile**: Un utente può avere uno o più profili.
- **profile e phone_number**: Ogni profilo può avere numeri di telefono associati.
- **profile e assistant**: Gli assistenti sono associati ai profili.
- **assistant e base_knowledge**: Gli assistenti possono avere conoscenze di base associate tramite la tabella assistant_base_knowledge.
- **call e phone_number**: Le chiamate sono collegate ai numeri di telefono e possono anche coinvolgere numeri esterni.
- **call_system_message e call**: I messaggi di sistema sono collegati alle chiamate.
- **conversation_transcript e call**: Le trascrizioni delle conversazioni sono collegate alle chiamate.
- **task_status_base_knowledge e base_knowledge**: Monitora lo stato dei task legati alla conoscenza di base.

### 3.3 Design delle interfacce

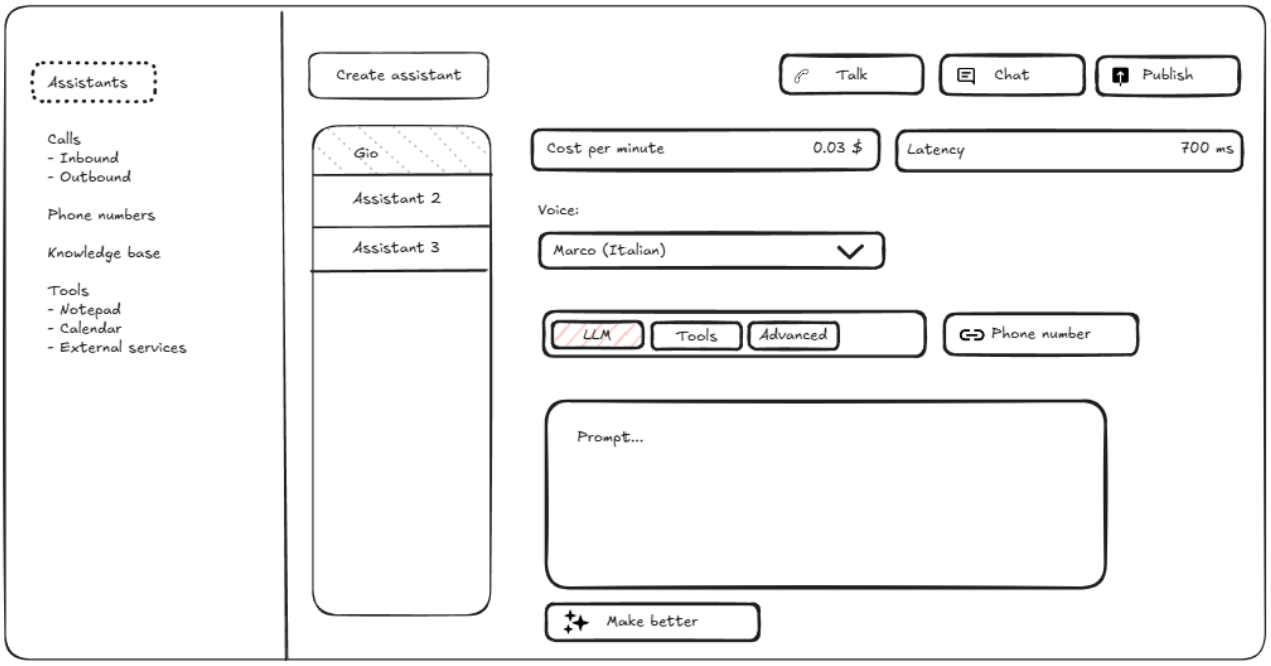



Figura 5 - Design 1

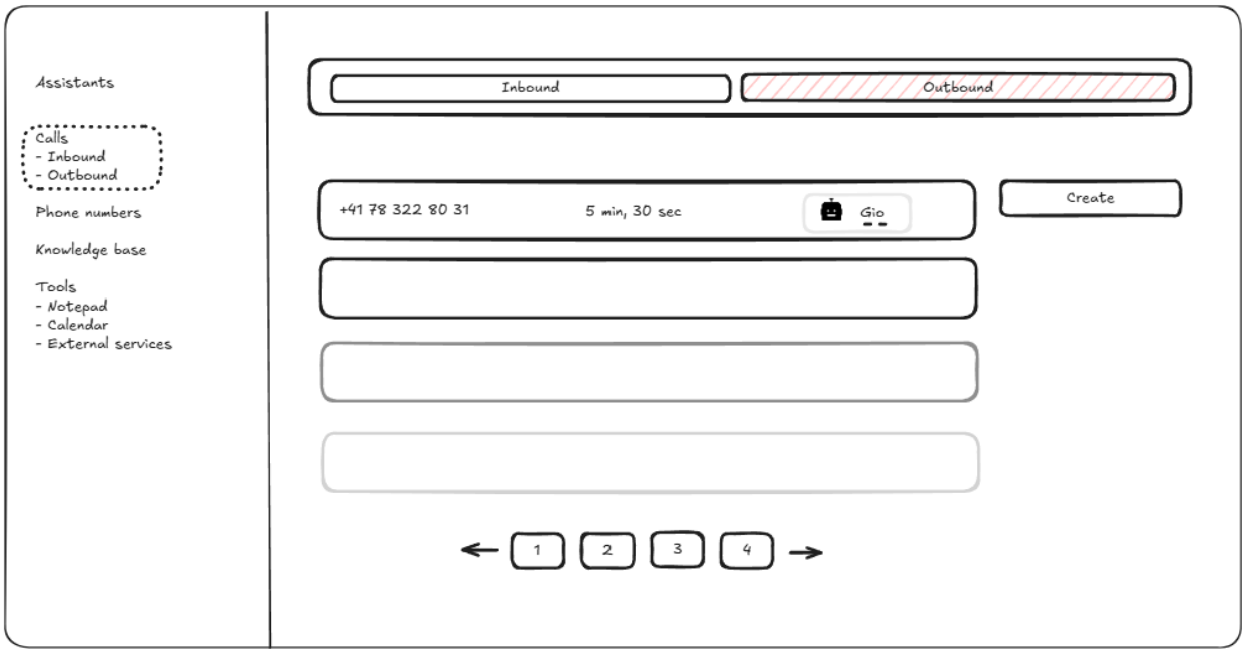



Figura 6 - Design 2

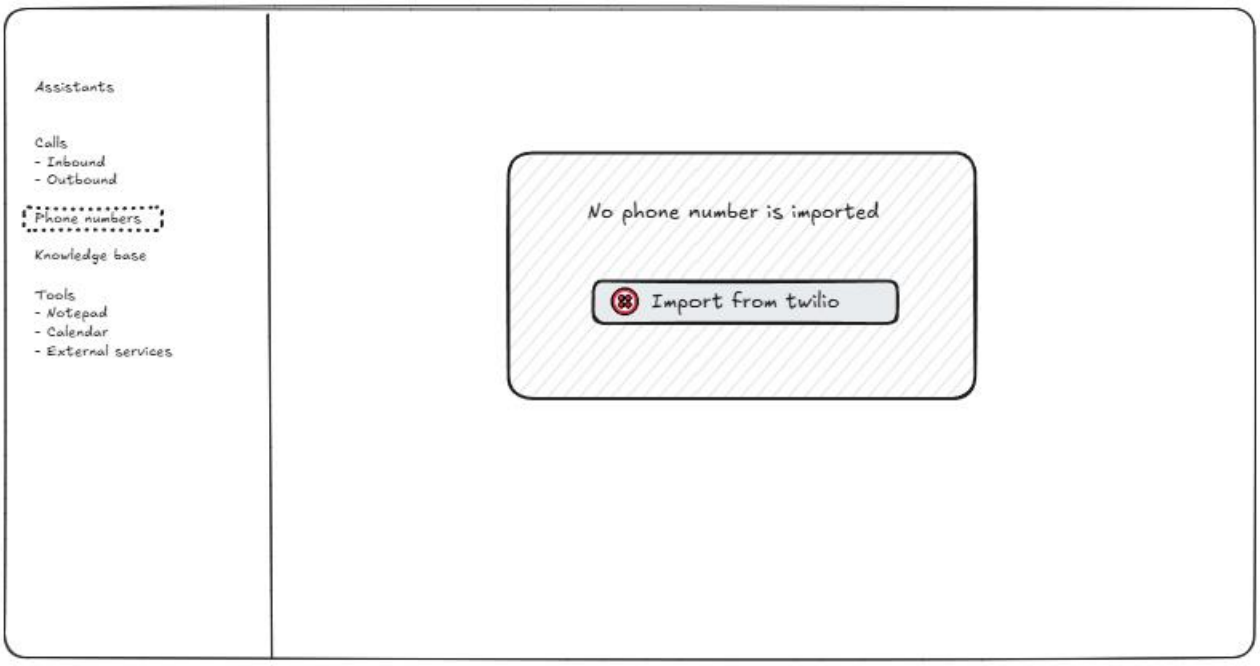



Figura 7 - Design 3


Figura 8 - Design 4

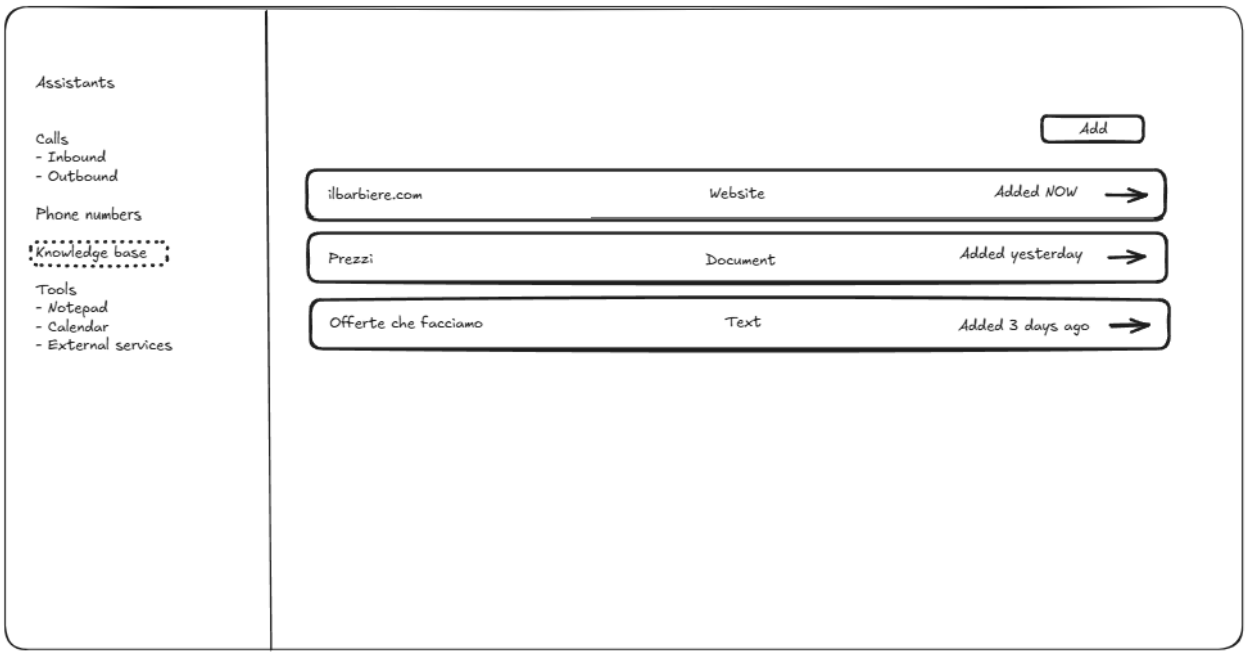



Figura 9 - Design 5

# 4 Implementazione

## 4.1 Struttura delle cartelle

**Root del progetto**
Nella root del progetto sono presenti i file di configurazione principali, tra cui **docker-compose.yml** e **nginx.conf**. Il file **docker-compose.yml** è utilizzato per definire e gestire i servizi del progetto tramite Docker, facilitando la creazione, l'esecuzione e la configurazione dei container necessari per l'ambiente di sviluppo e produzione. Il file **nginx.conf**, invece, contiene le configurazioni relative al server web Nginx, come la gestione delle rotte, la sicurezza e le impostazioni di performance. Entrambi i file sono essenziali per la gestione dell'infrastruttura e per il corretto funzionamento dell'applicazione in ambienti containerizzati.

**Struttura del Backend**
La struttura del backend è suddivisa in componenti principali e directory di risorse, ognuna delle quali ha una funzione specifica.

Componente principali dell'applicazione:
- **/app/**: Contiene la logica core dell'applicazione.
- **/assistants/**: Gestisce tutte le funzionalità relative agli assistenti virtuali e all'AI.
- **/base_knowledge/**: Gestisce e organizza la base di conoscenza dell'applicazione.
- **/calls/**: Si occupa della gestione delle chiamate.
- **/phone_numbers/**: Gestisce l'archiviazione e la gestione dei numeri di telefono.
- **/security/**: Contiene i moduli per la sicurezza e l'autenticazione degli utenti.
- **Directory delle risorse:**
- **/audio_logs/**: Archiviazione delle registrazioni delle chiamate.
- **/certs/**: Contiene i certificati SSL per garantire la sicurezza delle comunicazioni.
- **/files/**: Directory per l'archiviazione di file, inclusi i vari database di conoscenza.
- **/libs/**: Librerie personalizzate utilizzate all'interno dell'applicazione.
- **/migrations/**: File di migrazione per la gestione delle modifiche al database.
- **ChromDB**: Directory contenente diversi database di conoscenza basati su *chroma_db* (database vettoriale per il recupero delle informazioni).

**Struttura del Frontend**
Il frontend è organizzato in modo da separare i componenti riutilizzabili, la gestione dello stato e i servizi API. La struttura include anche le risorse statiche e le definizioni per la build e la pubblicazione con Nginx.

Codice Sorgente (/src/):
- **/components/**: Componente riutilizzabili sviluppati con Vue.js.
- **/icons/**: Componenti dedicati alle icone utilizzate nell'interfaccia.
- **/tests/**: Directory contenente i test dei componenti.
- **/views/**: Componente per la gestione delle diverse pagine dell'applicazione.
- **/router/**: Definizioni delle rotte dell'applicazione.
- **/stores/**: Gestione dello stato dell'applicazione.
- **/services/**: Servizi API per la comunicazione tra il frontend e il backend.
- **/types/**: Definizioni dei tipi TypeScript per garantire la corretta tipizzazione del codice.
- **/assets/**: Risorse statiche come immagini, fogli di stile, e altri file non dinamici.
- **Build e Pubblicazione:**
- **/dist/**: Output della build, contiene i file pronti per la produzione.
- **/public/**: File pubblici statici che possono essere accessibili dal client.
- **/certs/**: Certificati SSL per il frontend.

### 4.2 Docker Compose

Il progetto utilizza Docker Compose per orchestrare i seguenti servizi:

- Backend (Flask): API REST e logica applicativa
- Frontend (Vue.js): Serve a compilare il codice Vue.js in HTML, CSS e JavaScript
- Celery Worker: Gestione asincrona delle operazioni lunghe
- Redis: Sistema di caching e message broker, serve come broker per le task di Celery
- MySQL: Database relazionale principale
- Nginx: Reverse proxy che gestisce il routing delle richieste HTTP/HTTPS e la terminazione SSL, fungendo da punto di ingresso sicuro per l'applicazione

Questa architettura a microservizi permette una maggiore scalabilità e manutenibilità del sistema.

#### 4.2.1 Container Backend

Il backend è implementato in Python usando Flask. Gestisce tutte le richieste API, l'autenticazione e la logica di business.

Caratteristiche principali:

- Espone API REST su HTTPS
- Espone WebSocket per la comunicazione in tempo reale utilizzando SocketIO
- Integrazione con MySQL per la persistenza dei dati
- Comunica con Celery per operazioni asincrone in background¨

```
backend:
  build: ./autocally-backend
  ports:
    - "5001:5000"  # Espone la porta 5000 del container sulla 5001 dell'host
  volumes:
    - ./autocally-backend:/app/src  # Monta il codice sorgente
    - ./server.cert:/app/src/certs/server.cert:ro  # Monta i certificati SSL
  environment:
    - REDIS_PORT=6380  # Configura la porta Redis
```

Il backend viene configurato per essere costruito dalla directory ./autocally-backend, con il codice sorgente montato come volume. I certificati SSL sono montati in sola lettura.

### 4.2.2 Container Celery

Celery è un sistema di gestione delle code distribuito progettato per gestire operazioni asincrone e task in background.
Permette di eseguire attività in modo parallelo, migliorando l'efficienza e la reattività delle applicazioni.
Utilizza eventlet come pool di workers, che è un insieme di processi o thread che eseguono le operazioni in modo concorrente.
Le operazioni I/O bound (Input/Output bound) sono quelle operazioni in cui il tempo di esecuzione è principalmente limitato dalla velocità di input e output (I/O), anziché dalla capacità di elaborazione del processore (CPU).

Funzionalità principali:

- Gestione aggiornamento asincrono delle Knowledge Base

```
celery_worker:
  build: ./autocally-backend
  command: celery -A celery_worker.celery worker --pool=eventlet  # Avvia worker con eventlet
  volumes:
    - ./autocally-backend:/app/src  # Condivide lo stesso codice del backend
  environment:
    - REDIS_PORT=6380  # Usa la stessa porta Redis
```

Questo worker Celery usa la stessa base del backend e condivide il codice sorgente. Il comando specifica l'uso di eventlet come pool per gestire le operazioni.

### 4.2.3 Container Redis

Redis viene utilizzato sia come sistema di caching che come message broker per Celery.

```
redis:
  image: "redis:alpine"  # Usa l'immagine Alpine per dimensioni ridotte
  volumes:
    - redis_data:/data  # Volume per persistenza dati
  ports:
    - "6380:6379"  # Porta Redis esposta
```

Usa un'immagine Alpine per minimizzare le dimensioni. I dati sono persistenti grazie al volume dedicato.

### 4.2.4 Container MySQL

Database che gestisce tutti i dati persistenti dell'applicazione.
I dati sono persistenti grazie al volume mysql_data.

### 4.2.5 Container Nginx

Nginx funge da reverse proxy, gestendo SSL termination e servendo i file statici del frontend.

Funzionalità principali:

- Reverse Proxy
- Terminazione SSL/TLS
- Serving di contenuti statici

```
nginx:
  image: nginx:alpine  # Versione leggera di Nginx
  volumes:
    - ./nginx.conf:/etc/nginx/nginx.conf:ro  # Configurazione Nginx
    - ./server.cert:/etc/nginx/ssl/server.cert:ro  # Certificato SSL
    - ./server.key:/etc/nginx/ssl/server.key:ro  # Chiave SSL
  ports:
    - "443:443"  # Porta HTTPS
```

Monta i certificati e la configurazione Nginx in nginx.conf, in seguito le parti importanti della configurazione.

```
server {
    listen 443 ssl;
    server_name localhost;

    # Certificati SSL
    ssl_certificate     /etc/nginx/ssl/server.cert;
    ssl_certificate_key /etc/nginx/ssl/server.key;

    # Proxy per le API del backend
    location /api {
        proxy_pass https://172.20.0.11:5000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    # Gestione WebSocket per SocketIO
    location /socket.io {
        proxy_pass https://172.20.0.11:5000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";
    }

    # Frontend Vue.js
    location / {
        root   /usr/share/nginx/html;
        index  index.html;
        try_files $uri $uri/ /index.html;
    }
}
```

Questa configurazione gestisce tre aspetti principali:

- HTTPS: Configurazione SSL sulla porta 443
- API Backend: Reverse proxy per le chiamate API su /api
- WebSocket: Supporto per comunicazione real-time con SocketIO
- Frontend: Serving dell'applicazione Vue.js con gestione del routing

### 4.2.6 Rete Container

La rete Docker è configurata come bridge network isolata, permettendo comunicazione sicura tra i servizi.

### 4.2.7 Volumi Container

I volumi Docker garantiscono persistenza dei dati e performance ottimali.

```
volumes:
  redis_data:  # Volume per Redis
  mysql_data:  # Volume per MySQL
  node_modules:  # Volume per dipendenze Node.js
```

### 4.2.8 Certificati SSL

La sicurezza è garantita da certificati SSL condivisi tra i servizi.

Implementazione:

- Certificati montati in sola lettura
- Condivisi tra backend, frontend e nginx

## 4.3 Configurazione Backend, Gunicorn, Eventlet, Socketio

L'architettura del backend integra tutti questi componenti per creare un sistema robusto e performante. Gunicorn gestisce le richieste HTTP, Eventlet fornisce la concorrenza asincrona, e SocketIO abilita la comunicazione real-time. Questa combinazione permette di gestire efficacemente sia il traffico web tradizionale che le comunicazioni in tempo reale, mantenendo un'ottima scalabilità e un uso efficiente delle risorse.

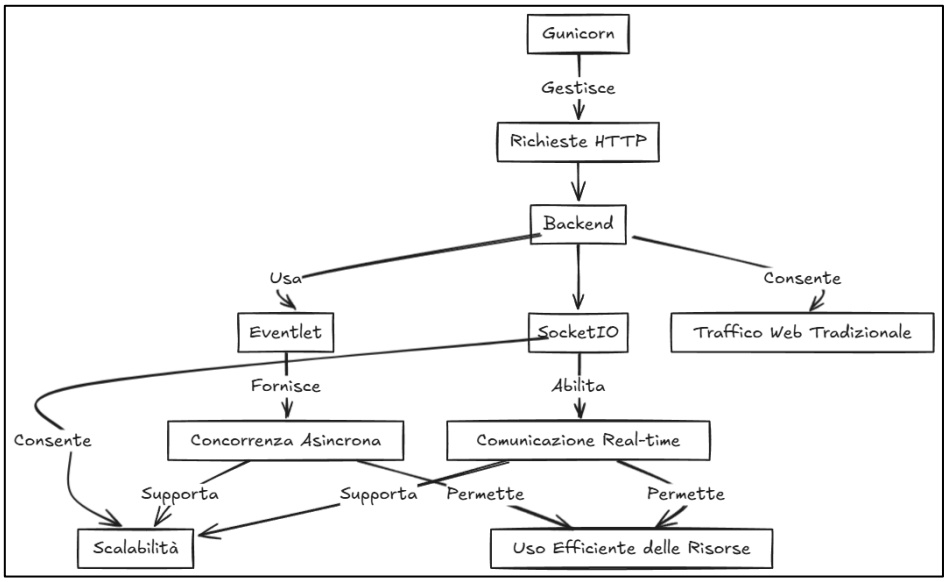



Figura 10 - Backend, Gunicorn, Eventlet, Socketio

### 4.3.1 Dockerfile

Il Dockerfile del backend è configurato per creare un ambiente Python con tutte le dipendenze necessarie per l'applicazione Flask.

```
FROM python:3.9-slim

WORKDIR /app/src

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY . .

CMD ["gunicorn", "--worker-class", "eventlet", "-w", "1", "--bind", "0.0.0.0:5000", "app:app"]
```

Questo Dockerfile crea un'immagine leggera basata su Python 3.9, installa le dipendenze necessarie e configura Gunicorn con Eventlet come worker class per gestire le connessioni WebSocket.

### 4.3.2 Gunicorn

Gunicorn (Green Unicorn) è un server WSGI HTTP per Python progettato per l'uso in ambienti di produzione, in quanto permette di gestire richieste multiple in modo efficiente e scalabile.

Gunicorn riceve le richieste da Nginx e le inoltra all'applicazione Flask. Il worker Eventlet gestisce efficientemente le connessioni multiple, incluse quelle WebSocket, mantenendo un basso consumo di risorse.

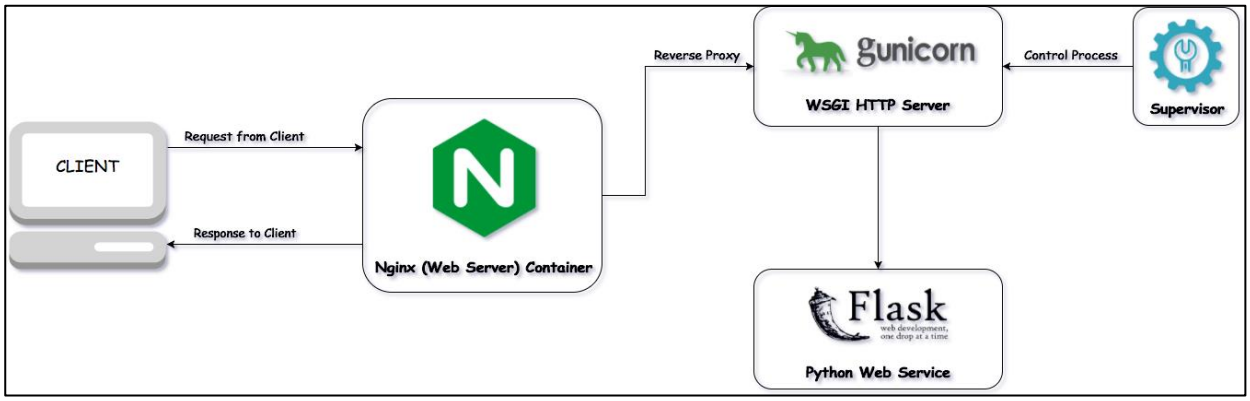



Figura 11 - Nginx, Gunicorn

Il Supervisor è un programma di gestione dei processi che viene utilizzato per monitorare e gestire i processi di sistema in modo che possano essere avviati, fermati o riavviati automaticamente in caso di errori.
Nel contesto di Gunicorn, Supervisor è utilizzato per garantire che l'applicazione web (ad esempio, una web app in Python) venga eseguita continuamente senza interruzioni, anche in caso di crash o riavvio del server.

Scelto per l'ambiente di produzione, Gunicorn è ideale per gestire in modo sicuro le connessioni WebSocket.

La configurazione base di Gunicorn include:

```
# Configurazione Gunicorn
bind = "0.0.0.0:5000"
worker_class = "eventlet"
workers = 1
timeout = 120
```

Imposta Gunicorn per ascoltare su tutte le interfacce sulla porta 5000, utilizzando Eventlet come worker class. Un singolo worker è sufficiente grazie all'efficienza di Eventlet nella gestione delle connessioni concorrenti.

### 4.3.3 Eventlet

Eventlet è una libreria di networking che permette la programmazione asincrona.

Eventlet utilizza il green threading per gestire la concorrenza.
Quando un'operazione di I/O blocca un thread, Eventlet passa automaticamente ad un altro task, ottimizzando l'utilizzo delle risorse del sistema.
Il green threading è una tecnica di programmazione che consente di gestire più task in modo concorrente all'interno di un singolo thread di sistema

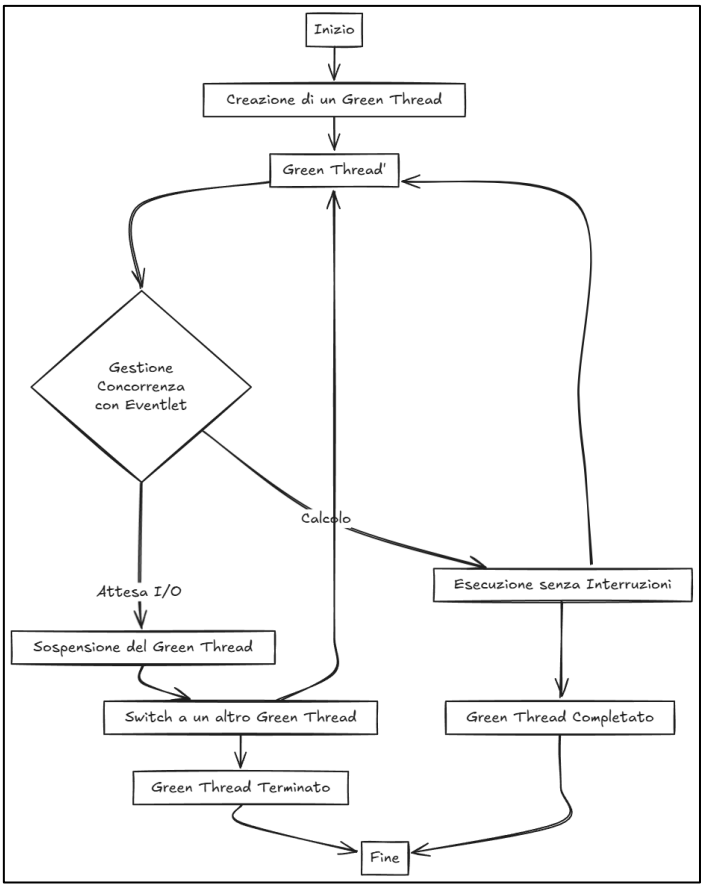



Figura 12 - Eventlet

```
import eventlet
eventlet.monkey_patch()

# Configurazione base
eventlet_socket = eventlet.listen(('', 5000))
eventlet.wsgi.server(eventlet_socket, app)
```

Inizializza Eventlet e applica il monkey patching necessario per rendere asincrone le operazioni di I/O. Il socket viene configurato per ascoltare sulla porta 5000 e servire l'applicazione Flask.

### 4.3.4 SocketIO

Flask-SocketIO permette la comunicazione real-time tra client e server con dei websocket.

Il server mantiene connessioni WebSocket persistenti con i client, permettendo lo scambio di messaggi in tempo reale. Il sistema gestisce automaticamente la riconnessione in caso di interruzioni e fornisce fallback al polling HTTP, che invia richieste periodiche al server per nuovi dati quando i WebSocket non sono disponibili, ma è meno efficiente e introduce latenza.

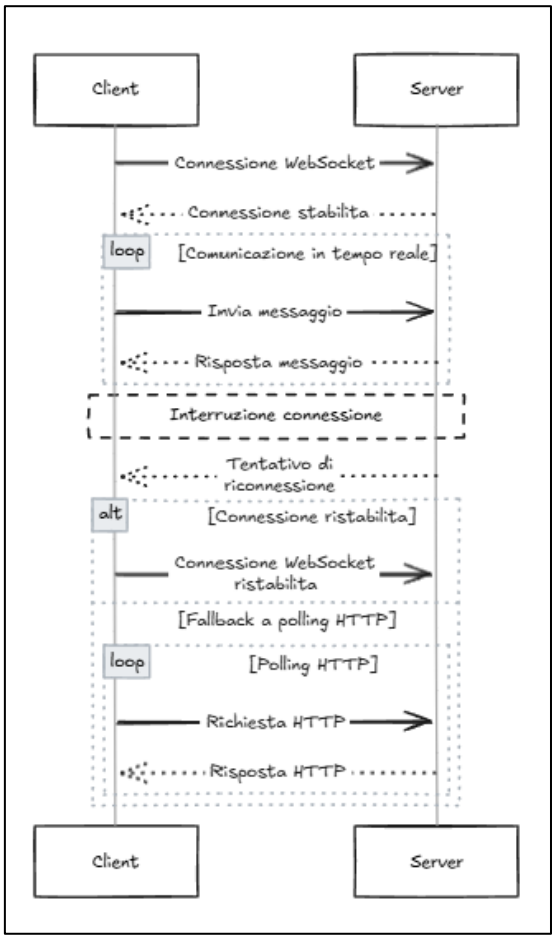



Figura 13 - Socketio

## 4.4 Login, Registrazione, Sicurezza

Il sistema di autenticazione utilizza token JWT per gestire le sessioni utente. L'architettura è composta da:

- Backend (Flask): Gestisce autenticazione e autorizzazione
- Frontend (Vue.js): Interfaccia di login e gestione token
- Database: Memorizza utenti, profili e token

Il processo di autenticazione inizia quando l'utente inserisce le proprie credenziali nel form di login. Il backend procede alla verifica delle credenziali e, se valide, genera un nuovo token JWT.

Questo token viene memorizzato nel frontend attraverso lo store di autenticazione, dopodiché l'utente viene reindirizzato alla dashboard.

Per ogni richiesta successiva che richiede autenticazione, il frontend include automaticamente il token nell'header Authorization. Il backend verifica la validità del token e, in base all'esito della verifica, consente o nega l'accesso alle risorse richieste.

Durante il logout, il frontend invia una richiesta dedicata al backend che provvede ad invalidare il token attivo. Lo store di autenticazione viene quindi ripulito dal token e l'utente viene reindirizzato alla pagina di login.

La sicurezza del sistema è garantita attraverso diverse misure: le password vengono hashate utilizzando l'algoritmo pbkdf2:sha256, mentre i token sono generati in modo sicuro tramite secrets.token_hex. Tutte le comunicazioni avvengono obbligatoriamente su HTTPS per garantire la cifratura dei dati in transito. Il sistema implementa una configurazione CORS che limita l'accesso ai soli domini autorizzati ed include protezioni contro attacchi XSS (Cross-Site Scripting) e CSRF (Cross-Site Request Forgery).

### 4.4.1 Backend

Il backend utilizza HTTPTokenAuth di Flask per l'autenticazione basata su token, implementa un sistema di autenticazione basato su token utilizzando la libreria Flask-HTTPAuth.
Viene creato un oggetto HTTPTokenAuth con lo schema Bearer, che è uno standard comune per l'autenticazione tramite token.
La funzione verify_token è decorata con @auth.verify_token, il che significa che verrà chiamata automaticamente per verificare la validità del token fornito nelle richieste.
All'interno della funzione, viene cercato un oggetto Token nel database corrispondente al token fornito.

Se il token è valido, la funzione restituisce l'utente associato a quel token; in caso contrario, restituisce None.
Questo meccanismo consente di autenticare gli utenti in modo sicuro e gestire le sessioni utente in modo efficace.

```
from flask_httpauth import HTTPTokenAuth
auth = HTTPTokenAuth(scheme='Bearer')

@auth.verify_token
def verify_token(token):
    token_obj = Token.query.filter_by(token=token).first()
    if token_obj:
        # Recupera l'utente associato al token
        return User.query.get(token_obj.user_id)
    return None
```

Il processo di login genera un token univoco per sessione:

```
@security.route('/login', methods=['POST'])
def login_view():
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter(
            (User.username == form.username.data) |
            (User.email == form.username.data)
        ).first()

        if user and check_password_hash(user.password, form.password.data):
            token = secrets.token_hex(16)
```

```
            new_token = Token(user_id=user.id, token=token)
            db.session.add(new_token)
            db.session.commit()
            return jsonify({'token': token, 'user': user.username})
```

Il logout invalida tutti i token dell'utente:

```
@security.route('/logout', methods=['POST'])
@auth.login_required
def logout_view():
    user = auth.current_user()
    if user:
        tokens = Token.query.filter_by(user_id=user.id).all()
        for token in tokens:
            db.session.delete(token)
        db.session.commit()
```

### 4.4.2 Frontend

Il frontend utilizza Axios per le chiamate API con intercettori per la gestione dei token.

```
const axiosInstance = axios.create({
    baseURL,
    headers: {
      'Content-Type': 'application/json',
    },
    withCredentials: true
  })

  const getHeaders = () => {
    const authStore = useAuthStore()
    return {
      headers: {
        'Content-Type': 'application/json',
        Authorization: `Bearer ${authStore.token}`
      }
    }
  }
```

```
const handleSubmit = async () => {
    try {
      const formData = new FormData()
      formData.append('username', email.value)
      formData.append('password', password.value)

      const response = await axiosInstance.post('/security/login', formData)
      const { token, user } = response.data

      // Store the token and user info
      authStore.setAuth(token, user)
      router.push('/')
    } catch (err) {
      error.value = err instanceof Error ? err.message : 'An error occurred'
    }
  }
```

Questa funzione gestisce l'invio del modulo di login.
Crea un oggetto FormData per raccogliere le credenziali dell'utente, invia una richiesta POST all'endpoint di login e, se la risposta è positiva, memorizza il token e le informazioni dell'utente nello store di autenticazione, reindirizzando infine l'utente alla dashboard.

### 4.5 Store Frontend

Lo store del frontend è gestito tramite Pinia, un gestore di stato per Vue.js che permette di mantenere e condividere dati tra i vari componenti dell'applicazione.

Lo store di autenticazione (auth.ts) gestisce lo stato dell'autenticazione dell'utente nell'applicazione. Utilizza il localStorage del browser per persistere i dati di autenticazione anche dopo il riavvio dell'applicazione.

Lo store mantiene tre stati principali:
- Il token JWT di autenticazione
- Il nome utente dell'utente autenticato
- Un flag che indica se l'utente è autenticato

Lo store espone due funzioni principali:

setAuth: Viene chiamata dopo un login riuscito e si occupa di:
- Memorizzare il nuovo token e username sia nello store che nel localStorage
- Impostare il flag di autenticazione
- Stabilire una connessione WebSocket autenticata

clearAuth: Viene chiamata durante il logout e si occupa di:
- Rimuovere token e username sia dallo store che dal localStorage
- Resettare il flag di autenticazione
- Disconnettere il WebSocket

Lo store è integrato con il sistema di WebSocket dell'applicazione, gestendo automaticamente la connessione e disconnessione del socket quando cambia lo stato di autenticazione.

Questo approccio centralizzato alla gestione dell'autenticazione garantisce consistenza in tutta l'applicazione e semplifica la gestione delle sessioni utente.

### 4.6 Phone Numbers

Il sistema permette agli utenti di importare e gestire numeri di telefono Twilio all'interno dell'applicazione. Questa funzionalità è essenziale per consentire agli assistenti virtuali di effettuare e ricevere chiamate telefoniche.

#### 4.6.1 Backend

Il modello PhoneNumber memorizza le informazioni essenziali dei numeri telefonici:

```
class PhoneNumber(db.Model):
```

```
    id = db.Column(db.Integer, primary_key=True)
    account_sid = db.Column(db.String(64), nullable=False)
    auth_token = db.Column(db.String(64), nullable=False)
    phone_number = db.Column(db.String(15), nullable=False)
    is_verified = db.Column(db.Boolean, default=False)
    profile_id = db.Column(db.Integer, db.ForeignKey('profile.id'), nullable=False)
    profile = db.relationship('Profile', backref='phone_numbers')
    created_at = db.Column(db.DateTime, server_default=db.func.now())
    updated_at = db.Column(db.DateTime, server_default=db.func.now(), onupdate=db.func.now())
```

Questo modello memorizza le credenziali Twilio (account_sid e auth_token) insieme al numero di telefono. Ogni numero è collegato a un profilo utente tramite una relazione che permette di recuperare facilmente tutti i numeri associati a un profilo. Il campo is_verified tiene traccia dello stato di verifica del numero, però non è stata implementata la verifica.

**add_phone_number()**
Questo endpoint gestisce l'aggiunta di un nuovo numero di telefono. Valida i dati in ingresso, controlla che tutti i campi necessari siano presenti e crea un nuovo record nel database. Utilizza gestione delle eccezioni per garantire l'integrità dei dati e fornire messaggi di errore appropriati.

**get_phone_numbers()**
Recupera tutti i numeri di telefono associati al profilo dell'utente corrente. Questa chiamata viene utilizzata per popolare l'interfaccia utente al caricamento della pagina. I dati vengono serializzati in JSON, includendo tutti i dettagli rilevanti per ciascun numero.

### 4.6.2 Frontend

La parte più interessante del frontend è la gestione degli stati di visualizzazione. A seconda della situazione, vengono mostrati componenti diversi:

```
<template>
    <template v-if="isLoading">
      <div class="h-full flex items-center justify-center">
        <div class="animate-spin rounded-full h-8 w-8 border-b-2 border-[#4285F4]"></div>
      </div>
    </template>
    <template v-else-if="error">
      <div class="h-full flex items-center justify-center">
        <div class="text-red-500">{{ error }}</div>
      </div>
    </template>
    <template v-else-if="showConfigForm">
      <TwilioConfigForm :onSave="handleSave" :onCancel="handleCancel" />
    </template>
    <template v-else-if="numbers.length === 0">
      <PhoneNumberEmptyState :onImport="handleImport" :isImporting="isImporting" />
    </template>
    <template v-else>
      <PhoneNumberList :numbers="numbers" @addNumber="handleImport" />
    </template>
</template>
```

La logica condizionale determina quale componente visualizzare:

- Durante il caricamento: uno spinner
- In caso di errore: messaggio di errore
- Durante l'importazione: form di configurazione Twilio
- Quando non ci sono numeri: stato vuoto personalizzato
- Quando ci sono numeri: lista dei numeri

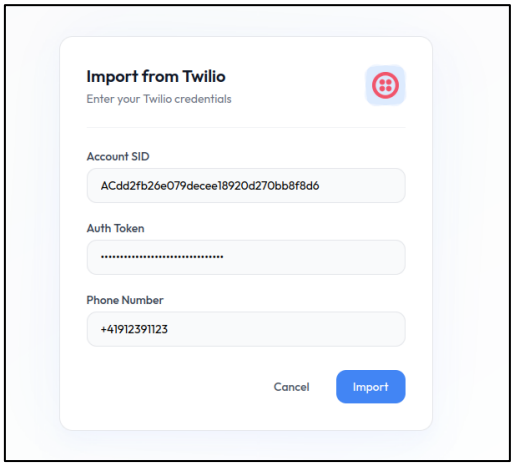



Figura 14 - Importazione numero Twilio

Un'altra caratteristica interessante è la gestione dell'espansione delle righe nella lista:

```
const expandedNumber = ref<string | null>(null)

const toggleExpand = (number: string) => {
  expandedNumber.value = expandedNumber.value === number ? null : number
}

// Nel template
<div v-if="expandedNumber === number.number" class="mt-4 pt-4 border-t border-gray-100">
  <div class="grid grid-cols-2 gap-4">
    <!-- Dettagli estesi mostrati solo quando la riga è espansa -->
    <div class="space-y-1">
      <p class="text-sm text-gray-500">Last call</p>
      <p class="text-sm font-medium text-gray-900">{{ number.lastCall }}</p>
    </div>
    <div class="space-y-1">
      <p class="text-sm text-gray-500">Total calls</p>
      <p class="text-sm font-medium text-gray-900">{{ number.totalCalls }}</p>
    </div>
    <!-- Altri dettagli... -->
  </div>
</div>
```

Questa implementazione permette all'utente di espandere una riga alla volta per visualizzare le statistiche dettagliate del numero di telefono selezionato. La funzione toggleExpand gestisce lo stato di espansione, consentendo di espandere una riga e comprimere quella precedentemente espansa.

```
<template>
    <div class="h-full w-full flex items-start justify-center bg-white pt-40">
      <div class="relative w-full max-w-5xl mx-auto px-6">
        <div class="bg-gradient-to-br from-gray-50 to-white py-20 rounded-2xl border border-gray-100
shadow-sm">
          <div class="flex flex-col items-center justify-center space-y-8">
            <div class="relative w-24 h-24 mb-4">
              <!-- Elementi decorativi -->
            </div>

            <div class="text-center space-y-4">
              <h1 class="text-2xl font-semibold text-gray-900">No phone numbers imported</h1>
              <p class="text-gray-500 max-w-md mx-auto">
                Import your Twilio phone numbers to start making calls and managing your
communication
              </p>
            </div>

            <button
              @click="onImport"
              :disabled="isImporting"
              class="group relative inline-flex items-center gap-2 px-6 py-3 bg-white border border-
gray-200 rounded-xl text-sm font-medium text-gray-700 hover:border-gray-300 hover:shadow-lg
hover:shadow-blue-50/50 transition-all duration-300 disabled:opacity-50"
            >
              <div class="relative w-5 h-5">
                <img src="@/assets/img/twilio.png" alt="" class="w-5 h-5 opacity-80 group-
hover:opacity-100 transition-opacity" />
              </div>
              <span class="relative">{{ isImporting ? 'Importing...' : 'Import from Twilio'
}}</span>
            </button>
          </div>
        </div>
```

```
      </div>
    </div>
  </template>
```

Invece di mostrare una semplice lista vuota, l'applicazione fornisce un'esperienza utente guidata con un messaggio informativo e un pulsante di azione chiaro. Questo componente è specificamente progettato per guidare l'utente nel processo di importazione del suo primo numero di telefono.

```
export const phoneNumbersApi = {
    importFromTwilio: async (config: {
      accountSid: string
      authToken: string
      phoneNumber: string
    }) => {
      const response = await axiosInstance.post('/phone-numbers/', {
        account_sid: config.accountSid,
        auth_token: config.authToken,
        phone_number: config.phoneNumber
      }, getHeaders())
      return response.data
    },

    getAll: async () => {
      const response = await axiosInstance.get('/phone-numbers/', getHeaders())
      return response.data
    }
  }
```

Questo modulo incapsula tutte le chiamate API relative ai numeri di telefono, semplificando l'interazione con il backend. Fornisce metodi per importare numeri da Twilio e recuperare tutti i numeri esistenti, gestendo automaticamente headers e formattazione dei dati.

### 4.6.3 Flusso utilizzo

- L'utente accede alla pagina Phone Numbers
- Se non ci sono numeri di telefono registrati, viene visualizzato PhoneNumberEmptyState che mostra un messaggio informativo e un pulsante per importare un numero.
- Quando l'utente preme il pulsante "Import from Twilio", viene mostrato il TwilioConfigForm dove può inserire le credenziali Twilio e il numero di telefono.
- Dopo aver salvato, il numero viene aggiunto al database e la vista passa automaticamente a PhoneNumberList che mostra tutti i numeri di telefono dell'utente.
- In PhoneNumberList, l'utente può vedere i dettagli dei numeri, espandere ogni voce per vedere statistiche dettagliate e aggiungere nuovi numeri.

L'applicazione include uno stato vuoto personalizzato che, in assenza di numeri di telefono, mostra un componente dedicato (PhoneNumberEmptyState) per guidare l’utente nel processo di importazione. I numeri di telefono sono presentati in una visualizzazione espandibile, che consente di accedere a statistiche dettagliate come il numero totale di chiamate, la durata media e il tasso di successo. Inoltre, per ogni numero viene fornita una chiara indicazione dell’utilizzo, specificando se è assegnato a un assistente o se non è in uso. L’app si integra con l’API Twilio, permettendo la gestione dei numeri per le chiamate in entrata e in uscita. Infine, l’interfaccia è reattiva e fornisce feedback visivi durante il caricamento e l’importazione per tenere l’utente sempre informato sullo stato delle operazioni.

### 4.7 Assistants

Gli “Assistants” sono una componente centrale dell'applicazione che permette agli utenti di creare e gestire assistenti virtuali in grado di effettuare e ricevere chiamate telefoniche. Ogni assistente può essere personalizzato con un prompt specifico, un messaggio di saluto, una voce personalizzata e può essere collegato a numeri di telefono per la comunicazione.

#### 4.7.1 Backend

Il backend utilizza un modello di dati che memorizza tutte le informazioni relative agli assistenti nel database:

```
class Assistant(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), nullable=False)
    prompt = db.Column(db.String(10000), nullable=True)
    greeting_message = db.Column(db.String(500), nullable=True)
    cartesia_voice_id = db.Column(db.String(64), nullable=True)
    llm_model = db.Column(db.String(64), default="llama-3.3-70b-versatile")
    llm_temperature = db.Column(db.Float, default=0.0)
    llm_max_tokens = db.Column(db.Integer, default=100)

    phone_number_id = db.Column(db.Integer, db.ForeignKey('phone_number.id'), nullable=True)
    phone_number = db.relationship('PhoneNumber', backref='assistants', lazy='joined')

    profile_id = db.Column(db.Integer, db.ForeignKey('profile.id'), nullable=False)
    profile = db.relationship('Profile', backref='assistants', foreign_keys=[profile_id])

    created_at = db.Column(db.DateTime, server_default=db.func.now())
    updated_at = db.Column(db.DateTime, server_default=db.func.now(), onupdate=db.func.now())
```

Il modello include proprietà fondamentali come il nome dell'assistente, il prompt di istruzione, il messaggio di benvenuto e l'ID della voce utilizzata. I parametri LLM (llm_model, llm_temperature, llm_max_tokens) controllano il comportamento del modello di linguaggio utilizzato dall'assistente. Ogni assistente può essere associato a un numero di telefono e appartiene a un profilo utente specifico.

Il backend fornisce diversi endpoint per la gestione completa degli assistenti.

```
@assistants.route('/create', methods=['POST'])
@auth.login_required
def create_assistant():
    data = request.get_json()
    current_user = auth.current_user()
    current_profile = current_user.profile

    if not current_profile:
        return jsonify({'error': 'No profile found for user'}), 400

    name = data.get('name')
    if not name:
        name = 'New Assistant ' + str(uuid.uuid4())

    new_assistant = Assistant(
        name=name,
        prompt=data.get('prompt'),
```

```
        greeting_message=data.get('greeting_message'),
        cartesia_voice_id=data.get('cartesia_voice_id'),
        phone_number_id=data.get('phone_number_id'),
        profile_id=current_profile.id,
        llm_model=data.get('llm_model', "llama-3.3-70b-versatile"),
        llm_temperature=data.get('llm_temperature', 0.0),
        llm_max_tokens=data.get('llm_max_tokens', 100)
    )

    db.session.add(new_assistant)
    db.session.commit()

    return jsonify({
        'message': 'Assistant created successfully',
        'assistant': {
            'id': new_assistant.id,
            'name': new_assistant.name,
            'prompt': new_assistant.prompt,
            'greeting_message': new_assistant.greeting_message,
            'cartesia_voice_id': new_assistant.cartesia_voice_id,
            'phone_number_id': new_assistant.phone_number_id,
            'llm_model': new_assistant.llm_model,
            'llm_temperature': new_assistant.llm_temperature,
            'llm_max_tokens': new_assistant.llm_max_tokens
        }
    }), 201
```

Questo endpoint gestisce la creazione di un nuovo assistente. Se il nome non viene fornito, ne genera uno automaticamente. Imposta valori predefiniti per i parametri del modello di linguaggio e associa l'assistente al profilo dell'utente corrente.

```
@assistants.route('/', methods=['GET'])
@auth.login_required
def get_assistants():
    try:
        current_user = auth.current_user()
        current_profile = current_user.profile

        if not current_profile:
            return jsonify({'error': 'No profile found for user'}), 400

        all_assistants = Assistant.query.filter_by(profile_id=current_profile.id).all()
        result = assistants_schema.dump(all_assistants)
        return jsonify(result), 200
    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Questo endpoint recupera tutti gli assistenti associati al profilo dell'utente corrente. Utilizza un serializzatore (assistants_schema) per convertire gli oggetti del modello in JSON.

```
@assistants.route('/update/<int:assistant_id>', methods=['PUT'])
@auth.login_required
def update_assistant(assistant_id):
    current_profile = auth.current_user().profile
    assistant = Assistant.query.get_or_404(assistant_id)

    if assistant.profile_id != current_profile.id:
        return jsonify({'error': 'Unauthorized access'}), 403
```

```
    data = request.get_json()

    if 'name' in data:
        assistant.name = data['name']
    if 'prompt' in data:
        assistant.prompt = data['prompt']
    if 'greeting_message' in data:
        assistant.greeting_message = data['greeting_message']
    if 'cartesia_voice_id' in data:
        assistant.cartesia_voice_id = data['cartesia_voice_id']
    if 'phone_number_id' in data:
        assistant.phone_number_id = data['phone_number_id']
    if 'llm_model' in data:
        assistant.llm_model = data['llm_model']
    if 'llm_temperature' in data:
        assistant.llm_temperature = data['llm_temperature']
    if 'llm_max_tokens' in data:
        assistant.llm_max_tokens = data['llm_max_tokens']

    db.session.commit()

    return jsonify({
        'message': 'Assistant updated successfully',
        'assistant': {
            'id': assistant.id,
            'name': assistant.name,
            'prompt': assistant.prompt,
            'greeting_message': assistant.greeting_message,
            'cartesia_voice_id': assistant.cartesia_voice_id,
            'phone_number_id': new_assistant.phone_number_id,
            'llm_model': assistant.llm_model,
            'llm_temperature': assistant.llm_temperature,
            'llm_max_tokens': assistant.llm_max_tokens
        }
    })
```

Questo endpoint permette di aggiornare un assistente esistente. Verifica che l'utente sia autorizzato ad accedere all'assistente e aggiorna solo i campi forniti nella richiesta.

Gli assistenti possono essere associati a Knowledge Base per migliorare le loro capacità di risposta, con testi e documenti dati dall’utente.

```
assistant_base_knowledge = db.Table('assistant_base_knowledge',
    db.Column('assistant_id', db.Integer, db.ForeignKey('assistant.id'), primary_key=True),
    db.Column('base_knowledge_id', db.Integer, db.ForeignKey('base_knowledge.id'), primary_key=True)
)
```

Questa tabella di relazione molti-a-molti consente di associare più basi di conoscenza a un assistente e viceversa, fornendo grande flessibilità nella configurazione delle conoscenze disponibili per ciascun assistente.

### 4.7.2 Frontend

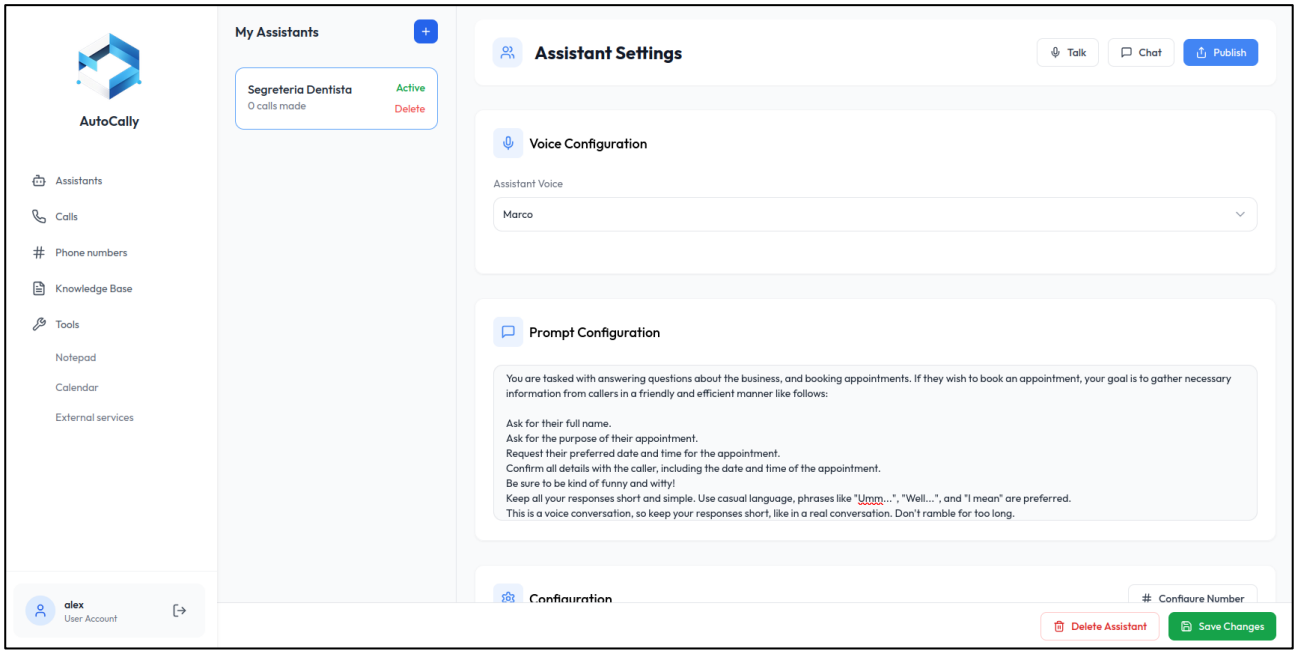



Figura 15 - Interfaccia assistenti

```
export const assistantsApi = {
  create: async (data: {
    name?: string
    prompt?: string
    greeting_message?: string
    cartesia_voice_id?: string
    phone_number_id?: string
  }) => {
    const response = await axiosInstance.post('/assistants/create', data, getHeaders())
    return response.data
  },

  getAll: async () => {
    const response = await axiosInstance.get('/assistants/', getHeaders())
    return response.data
  },

  getById: async (id: string | number) => {
    const response = await axiosInstance.get(`/assistants/${id}`, getHeaders())
    return response.data
  },

  delete: async (id: string | number) => {
    const response = await axiosInstance.delete(`/assistants/delete/${id}`, getHeaders())
    return response.data
  },

  update: async (id: number, data: {
    name?: string;
```

```
    prompt?: string;
    greeting_message?: string;
    cartesia_voice_id?: string;
    phone_number_id?: string;
    llm_model?: string;
    llm_temperature?: number;
    llm_max_tokens?: number;
  }) => {
    const response = await axiosInstance.put(`/assistants/update/${id}`, data, getHeaders())
    return response.data
  }
}
```

Questa parte di api.ts fornisce metodi per comunicare con il backend per tutte le operazioni relative agli assistenti. Include funzioni per creare, recuperare, aggiornare ed eliminare assistenti, gestendo automaticamente intestazioni e formattazione dei dati.

### 4.7.3 Flusso di utilizzo

1. L'utente accede alla pagina Home dove vede un elenco dei suoi assistenti
2. L'utente può creare un nuovo assistente facendo clic sul pulsante "+"
3. Selezionando un assistente, l'utente può modificarne i dettagli nella vista AssistantsView
4. L'utente può configurare diversi parametri dell'assistente:
   - Prompt di istruzione
   - Modello LLM e parametri (temperatura, token massimi)
   - Messaggio di saluto
   - Voce dell'assistente
   - Numero di telefono associato
5. L'utente può collegare l'assistente a basi di conoscenza esistenti
6. L'utente può effettuare una chiamata di test con l'assistente per verificarne il funzionamento

## 4.8 Assistant LLM

La classe AssistantLLM è il cuore dell'intelligenza artificiale degli assistenti. Questo componente è responsabile dell'elaborazione del linguaggio naturale, della gestione delle conversazioni e dell'integrazione con le basi di conoscenza.

Sono state utilizzate le seguenti tecnologie:

- **LangChain**: Framework per lo sviluppo di applicazioni basate su modelli linguistici, utilizzato per gestire prompt, memoria delle conversazioni e strumenti.
- **Groq**: Servizio di API per modelli linguistici ad alta velocità, utilizzato per generare risposte rapide durante le chiamate.
- **Chroma**: Database vettoriale per la memorizzazione e il recupero di embedding semantici, utilizzato nelle basi di conoscenza.
- **OpenAI Embeddings**: Servizio per la creazione di rappresentazioni vettoriali di testi, utilizzato per la ricerca semantica nelle basi di conoscenza.

```
class AssistantLLM:
    def __init__(self, assistant_id):
```

```
        self.assistant_id = assistant_id
        self.assistant = Assistant.query.get(self.assistant_id)

        self.llm = ChatGroq(
            model=self.assistant.llm_model,
            temperature=self.assistant.llm_temperature,
            max_tokens=self.assistant.llm_max_tokens
        )

        self.memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
        self.embedding_function = OpenAIEmbeddings()
        self.tools = []

        self.get_knowledge_base()
        self.prompt = self.get_prompt()


        self.agent = create_tool_calling_agent(self.llm, self.tools, self.prompt)
        self.agent_executor = AgentExecutor(
            agent=self.agent,
            tools=self.tools,
            memory=self.memory,
            verbose=True,
            max_iterations=2  # Limit to 2 iterations per query to avoid repeated tool calls.
        )
```

All'inizializzazione, la classe carica i dati dell'assistente dal database tramite l'ID fornito. Configura il modello linguistico (ChatGroq) con i parametri personalizzati dell'assistente: modello scelto, temperatura (che controlla la casualità delle risposte) e numero massimo di token (che limita la lunghezza delle risposte).

Viene creata una memoria di conversazione utilizzando ConversationBufferMemory per mantenere la continuità del dialogo. Viene inizializzata anche la funzione di embedding per la ricerca semantica.

Il metodo get_knowledge_base() viene chiamato per caricare le basi di conoscenza associate all'assistente, mentre get_prompt() recupera e formatta il prompt di sistema. Infine, viene configurato l'agente LLM con i tools e il prompt, e viene creato un executor che gestirà l'esecuzione delle query.

```
def create_tool_func(self, info_chain):
    """Helper to create a function that calls the corresponding info_chain."""
    def tool_func(query):
        return info_chain({"query": query})
    return tool_func
```

Questa funzione helper crea una funzione di tool che incapsula una catena di recupero di informazioni (info_chain). Quando questa funzione viene chiamata con una query, esegue la catena di recupero e restituisce il risultato. Questo pattern è utilizzato per creare strumenti dinamici per ogni base di conoscenza associata all'assistente.

```
def get_knowledge_base(self):
    try:
        base_knowledge = (
            db.session.query(BaseKnowledge)
            .join(assistant_base_knowledge)
            .filter(assistant_base_knowledge.c.assistant_id == self.assistant_id)
            .all()
        )

        for knowledge in base_knowledge:
            name = knowledge.name
```

```
            path = os.path.join(knowledge.folder_path, 'chroma_db')
            description = knowledge.description

            db_chroma = Chroma(persist_directory=path, embedding_function=self.embedding_function)
            info_chain = RetrievalQA.from_chain_type(
                llm=self.llm,
                chain_type="stuff",
                retriever=db_chroma.as_retriever(),
                return_source_documents=True,
                verbose=True
            )
            tool = Tool(
                name=name,
                description=f"Use this tool to get information about: {description}",
                func=self.create_tool_func(info_chain)
            )
            self.tools.append(tool)

    except Exception as e:
        logger.error(f"Error getting knowledge base: {str(e)}")
        return None
```

Questo metodo recupera tutte le basi di conoscenza associate all'assistente dal database. Per ogni base di conoscenza:

1. Viene estratto il nome, il percorso del database Chroma e la descrizione.
2. Viene caricato il database Chroma con la funzione di embedding configurata.
3. Viene creata una catena di recupero (RetrievalQA) che utilizza il modello LLM e il retriever di Chroma.
4. Viene creato un tool LangChain che incapsula questa catena di recupero, con un nome e una descrizione appropriati.
5. Il tool viene aggiunto alla lista degli strumenti disponibili per l'assistente.

Questo approccio consente all'assistente di accedere dinamicamente a tutte le basi di conoscenza a cui è stato associato.

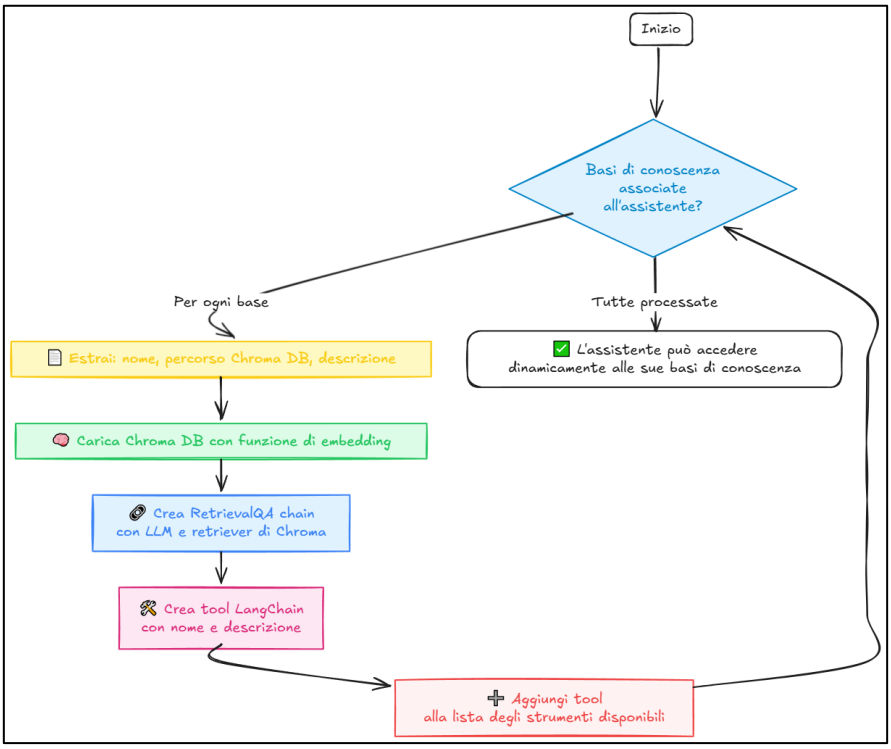



Figura 16 - Assistant LLM

```
def get_prompt(self):
    try:
        if not self.assistant:
            raise ValueError(f"No assistant found with id {self.assistant_id}")

        tools_str = "\n".join([
            f"- {tool.name}: {tool.description}"
            for tool in self.tools
        ])

        system_prompt = f"""


You are a helpful conversational AI assistant. Your goal is to assist users by providing accurate,
concise, and helpful responses.

## Rules
- **Strict Rule Compliance:** Follow these rules precisely. Any violation is not tolerated.

### Tool Usage Guidelines
- Only use a tool if it directly contributes to answering the user's query.
- Limit to one tool call per query. Do not call additional tools afterward.
- Never reveal the tool's name or internal details in your response.
- After using a tool, incorporate the findings into your response without mentioning the tool.
- Do not invent information; rely on tool outputs or your general knowledge.

### Conversation Style
- Write numbers, dates and times in words.
- Use a conversational tone, as if speaking to a friend.
- Keep responses brief to mimic natural dialogue in phone calls.
- Use casual phrases like "Umm...", "Well...", or "I mean..." to sound more human.
- Avoid unnecessary details or repetition to keep the conversation flowing.
- If you don't know an answer, simply say so.
```

```
### Additional Requirements
- Do not include tool names, descriptions, or technical details in responses.
- Base replies on provided information or general knowledge only.
- Be open to refining your responses based on user feedback to improve accuracy and relevance.

## Tools
You have access to the following tools:
{tools_str}

## Conversation Prompt
{self.assistant.prompt}

"""

        return ChatPromptTemplate.from_messages([
            ("system", system_prompt),
            MessagesPlaceholder(variable_name="chat_history"),
            ("human", "{input}"),
            ("placeholder", "{agent_scratchpad}")
        ])

    except Exception as e:
        logger.error(f"Error getting prompt: {str(e)}")
        raise
```

Questo metodo genera il prompt di sistema per l'assistente. Il prompt è composto da diverse sezioni:

1. Introduzione: Definisce lo scopo generale dell'assistente.
2. Regole di Utilizzo dei Tool: Linee guida su come utilizzare gli strumenti disponibili.
3. Stile di Conversazione: Istruzioni per adottare un tono conversazionale e naturale.
4. Requisiti Aggiuntivi: Ulteriori regole per le risposte.
5. Elenco Tools: Un elenco formattato di tutti gli strumenti disponibili.
6. Prompt Personalizzato: Il prompt specifico configurato per l'assistente.

Il prompt viene poi inserito in un template di ChatPromptTemplate che include anche:
- Il placeholder per la cronologia della chat
- Il messaggio dell'utente
- Un "scratchpad" per l'agente (utilizzato per i ragionamenti interni)

Questa struttura consente all'assistente di mantenere il contesto della conversazione e di ragionare su come utilizzare i tool disponibili.

```
def clean_response(self, text):
    """Remove function-like patterns from the response."""

    # Remove patterns like <function=name>content</function>
    cleaned_text = re.sub(r'<function=.*?</function>', '', text)
    return cleaned_text
```

Questa funzione rimuove eventuali pattern simili a funzioni dalla risposta dell'LLM. Questo è importante perché a volte i modelli linguistici possono generare contenuti che sembrano chiamate a funzioni o markup, che non dovrebbero essere visibili all'utente finale.

```python
def get_response(self, text):
    try:
        logger.info(f"Assistant is responding...")
        self.memory.chat_memory.add_user_message(text)
        logger.info(f"Added message to memory: {text}")

        response = self.agent_executor.invoke({
            "input": text,
            "chat_history": self.memory.chat_memory.messages
        })

        answer = response["output"]
        self.memory.chat_memory.add_ai_message(answer)
        logger.info(f"Added message to memory: {text}")

        logger.info(f"Memory: {self.memory.load_memory_variables({})}")
        logger.info(f"Assistant response: {answer}")
        return self.clean_response(answer)

    except Exception as e:
        logger.error(f"Error getting response: {str(e)}")
        return "I apologize, but I encountered an error while processing your request."
```

Questo è il metodo principale per generare risposte a partire dall'input dell'utente. Il processo è il seguente:

1. Il messaggio dell'utente viene aggiunto alla memoria della conversazione.
2. L'executor dell'agente viene invocato con il messaggio dell'utente e la cronologia della chat.
3. L'agente può decidere di utilizzare uno o più tool per raccogliere informazioni o eseguire azioni.
4. La risposta generata viene estratta dall'output dell'executor.
5. La risposta viene aggiunta alla memoria della conversazione come messaggio AI.
6. La risposta viene pulita utilizzando il metodo clean_response e restituita.

In caso di errore, viene restituito un messaggio di scusa generico invece di mostrare l'errore tecnico all'utente.

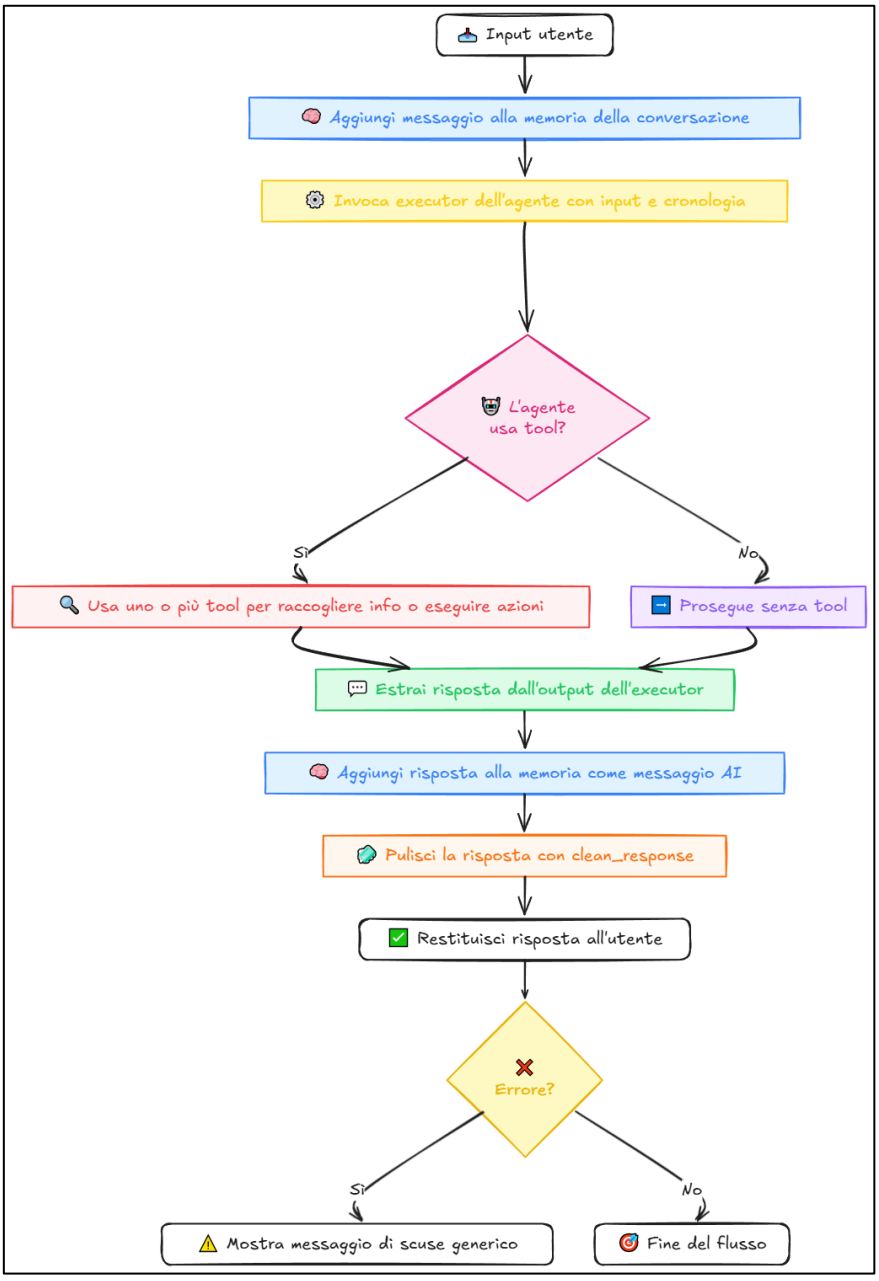



Figura 17 - Assistant LLM

Quando un assistente viene coinvolto in una chiamata, il sistema avvia il processo creando un'istanza dell'oggetto AssistantLLM, utilizzando l'ID specifico dell'assistente. Questo step rappresenta il punto di partenza per qualsiasi interazione, poiché consente di accedere ai dati e alle configurazioni personalizzate legate a quell'assistente.

Una volta creata l'istanza, l'assistente carica automaticamente i propri parametri di configurazione. Questo include l'impostazione del modello di linguaggio (LLM), la definizione di parametri come temperatura e token, e il caricamento delle basi di conoscenza collegate, che serviranno da riferimento durante la conversazione.

Quando l'utente invia un messaggio, questo viene gestito dal metodo get_response. È il punto d'ingresso per l'interazione: qui il messaggio viene preso in carico dall'agente intelligente per essere elaborato e ricevere una risposta pertinente.

Durante l'elaborazione, l'agente valuta il contenuto del messaggio e, se necessario, può decidere di attivare uno o più strumenti (tool) per cercare informazioni aggiuntive. Questo permette all'assistente di rispondere anche a richieste complesse o di natura informativa.

Dopo aver raccolto tutte le informazioni necessarie, l'agente genera una risposta che tiene conto dell'input dell'utente, della cronologia della conversazione fino a quel momento, e dei dati ottenuti attraverso gli strumenti esterni. Il risultato è una risposta contestuale, coerente e utile.

Infine, la conversazione viene salvata in memoria. Questo permette all'assistente di mantenere la continuità del dialogo, ricordando ciò che è stato detto in precedenza e offrendo un'esperienza più naturale e fluida anche in interazioni prolungate.

### 4.9 Assistant Voice

La funzionalità di selezione della voce permette agli utenti di personalizzare l'esperienza di interazione con gli assistenti virtuali, scegliendo una voce specifica che rifletta l'identità desiderata per il loro assistente.

#### 4.9.1 Backend

Nel modello dell'assistente, la voce è memorizzata come un ID di riferimento al servizio esterno Cartesia:

```
class Assistant(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), nullable=False)
    # ... altri campi ...
    cartesia_voice_id = db.Column(db.String(64), nullable=True)
    # ... altri campi ...
```

Il campo cartesia_voice_id memorizza l'identificatore univoco della voce nel sistema Cartesia, che è il servizio di text-to-speech utilizzato dall'applicazione. Questo campo è nullable, consentendo così la creazione di assistenti senza una voce specifica, nel qual caso verrà utilizzata una voce predefinita.

Il backend fornisce un endpoint per recuperare tutte le voci disponibili dal servizio Cartesia:

```
@assistants.route('/voices', methods=['GET'])
@auth.login_required
def get_voices():
    try:
        def fetch_voices():
            return requests.get(
                "https://api.cartesia.ai/voices/",
                params={},
                headers={
                    "X-API-Key": os.getenv('CARTESIA_API_KEY'),
                    "Cartesia-Version": "2024-06-10"
                },
            )
```

```
        pool = eventlet.GreenPool()
        response = pool.spawn(fetch_voices).wait()

        if response.status_code == 200:
            voices = response.json()

            return jsonify([{
                'id': voice.get('id'),
                'name': voice.get('name'),
                'description': voice.get('description'),
                'language': voice.get('language'),
                'gender': voice.get('gender'),
            } for voice in voices]), 200

        else:
            return jsonify({'error': f'Error fetching voices: {response.text}'}),
response.status_code

    except Exception as e:
        print(f"Error in get_voices: {str(e)}")
        return jsonify({'error': str(e)}), 500
```

Questo endpoint utilizza eventlet per effettuare una richiesta asincrona all'API di Cartesia. La risposta viene elaborata per estrarre solo le informazioni necessarie (id, nome, descrizione, lingua e genere) prima di essere restituita al frontend.

Quando un assistente deve parlare, il sistema utilizza il cartesia_voice_id salvato per generare l'audio corrispondente con la classe TTS che verrà descritta nel prossimo capitolo.

### 4.9.2 Frontend

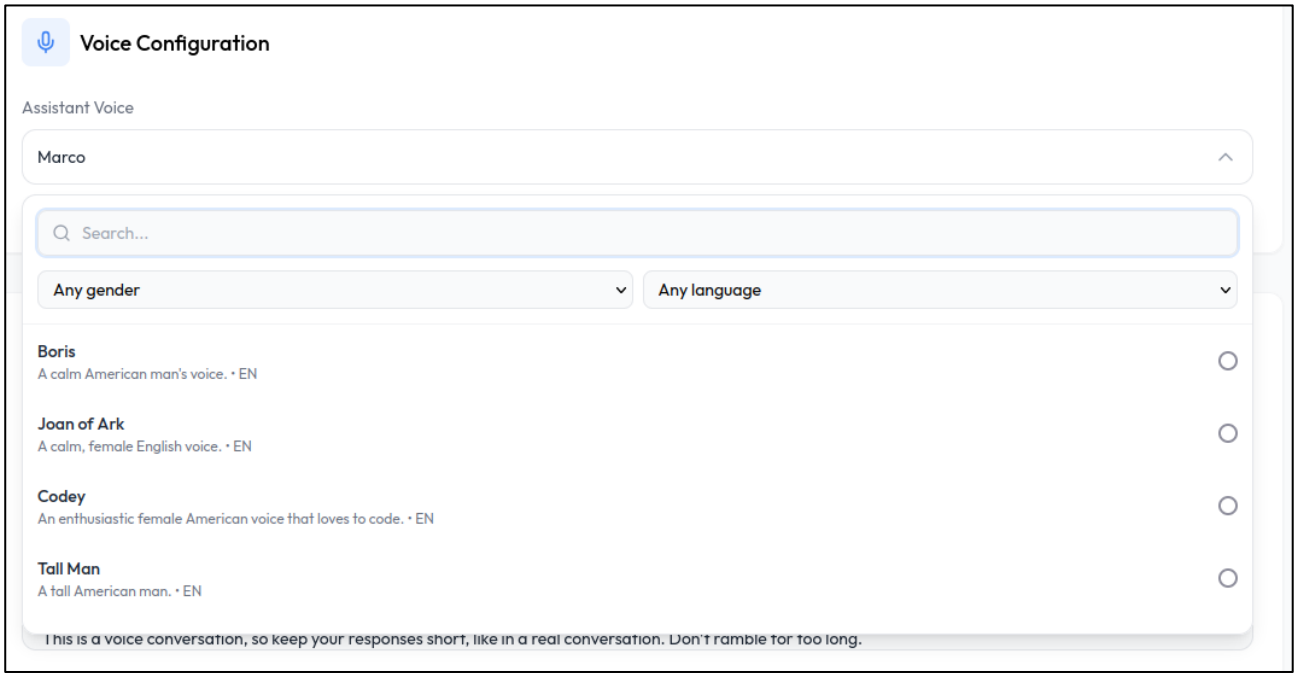



Figura 18 - Selezione voci

```
export const assistantsApi = {
  // ... altri metodi ...

  updateVoiceID: async (id: number, data: { cartesia_voice_id: string;}) => {
    const response = await axiosInstance.put(`/assistants/update/${id}`, data, getHeaders())
    return response.data
  },

  getVoices: async () => {
    const response = await axiosInstance.get('/assistants/voices', getHeaders())
    return {
      voices: response.data as Array<{
        id: string;
        name: string;
        description: string;
        language: string;
        gender: string;
      }>,
      languages: response.data.languages as string[]
    }
  }
}
```

La funzione getVoices si connette all'endpoint del backend per recuperare l'elenco completo delle voci disponibili, comprensivo di dettagli come identificatore, nome, descrizione, lingua e genere. Il metodo updateVoiceID consente invece di aggiornare la voce di un assistente specifico, inviando l'ID della voce selezionata al backend che si occuperà di salvare questa associazione nel database.

L'interfaccia utente per la selezione della voce è implementata attraverso un componente Vue dedicato, il componente offre un'interfaccia utente ricca e intuitiva per la selezione della voce.

Inizialmente, mostra la voce attualmente selezionata o un messaggio predefinito se nessuna voce è stata ancora scelta. Cliccando sul selettore, si espande un menu a discesa che presenta tutte le voci disponibili. L'utente può navigare in questo elenco, cercando voci specifiche tramite una barra di ricerca o applicando filtri per genere e lingua.

Ogni voce nell'elenco visualizza il suo nome, la descrizione e la lingua, permettendo all'utente di fare una scelta informata. Una volta identificata la voce desiderata, l'utente può selezionarla con un semplice clic, aggiornando così istantaneamente la configurazione dell'assistente.

La parte logica del componente gestisce la comunicazione con il backend e l'aggiornamento della voce:

```
const fetchVoices = async () => {
  try {
    if (!isLoading.value) {  // Prevent multiple simultaneous calls
      isLoading.value = true
      isLoadingLanguages.value = true

      const response = await assistantsApi.getVoices()

      if (Array.isArray(response)) {
        voices.value = response
        const uniqueLanguages = [...new Set(response.map(voice => voice.language))]
        languages.value = uniqueLanguages.sort()
      } else if (response.voices) {
        voices.value = response.voices
        const uniqueLanguages = [...new Set(response.voices.map(voice => voice.language))]
        languages.value = uniqueLanguages.sort()
      }

      // Set the initial selected voice if one is provided
      if (props.modelValue) {
        selectedVoice.value = props.modelValue
      }
    }
  } catch (err) {
    console.error('Failed to fetch voices:', err)
    error.value = 'Failed to load voices'
  } finally {
    isLoading.value = false
    isLoadingLanguages.value = false
  }
}

const selectVoice = async (voiceId: string) => {
  try {
    // Call API to update assistant's voice
    await assistantsApi.updateVoiceID(props.assistantId, {
      cartesia_voice_id: voiceId
    })

    selectedVoice.value = voiceId
    isExpanded.value = false
    emit('update:modelValue', voiceId)
  } catch (err) {
    console.error('Failed to update voice:', err)
    error.value = 'Failed to update voice'
  }
}
```

La logica che anima il componente di selezione della voce è progettata per fornire un'esperienza fluida e reattiva. La funzione fetchVoices si occupa di recuperare tutte le voci disponibili dal backend quando il componente viene inizializzato.

Durante questo processo, gestisce adeguatamente gli stati di caricamento per fornire feedback all'utente. Una volta ricevuti i dati, elabora le voci per estrarre l'elenco unico delle lingue disponibili, che verranno utilizzate per popolare il filtro linguistico.

Se l'assistente ha già una voce configurata, questa viene preselezionata automaticamente nell'interfaccia. La funzione selectVoice, invocata quando l'utente sceglie una nuova voce, gestisce la comunicazione con il backend per aggiornare la configurazione dell'assistente. Una volta completata con successo l'operazione, aggiorna lo stato locale e chiude il menu a discesa, fornendo un feedback immediato all'utente sulla sua azione.

Il componente di selezione della voce è integrato nella vista principale dell'assistente:

```
<div class="bg-white rounded-xl p-6 shadow-sm border border-gray-100 mb-8">
  <div class="flex items-center gap-2 mb-6">
    <div class="w-10 h-10 bg-[#4285F4]/10 rounded-lg flex items-center justify-center">
      <Mic class="w-5 h-5 text-[#4285F4]" />
    </div>
    <h2 class="text-lg font-medium">Voice Configuration</h2>
  </div>
  <VoiceSelector v-if="assistant" v-model="voiceId" :assistantId="Number(assistantId)" />
</div>
```

Questo blocco si trova nella vista principale dell'assistente e mostra il componente di selezione della voce quando i dati dell'assistente sono disponibili. Il componente riceve l'ID dell'assistente e il valore attuale della voce, consentendo all'utente di visualizzare e modificare facilmente la voce dell'assistente.

### 4.10 Assistant TTS

Il sistema di sintesi vocale, implementato in text_to_speech.py, rappresenta il componente che dà letteralmente voce agli assistenti virtuali dell'applicazione. Questo modulo trasforma i messaggi testuali generati dal modello linguistico in flussi audio naturali, creando l'illusione di una conversazione reale con l'assistente.

La sintesi vocale si basa su diverse tecnologie chiave per garantire un'esperienza audio di alta qualità:

- Cartesia API: Il sistema utilizza l'API di Cartesia, un servizio specializzato in sintesi vocale avanzata che offre voci naturali in diverse lingue e generi. L'integrazione con Cartesia permette all'applicazione di accedere a un'ampia gamma di voci personalizzabili.

- Streaming WebSocket: Per garantire una comunicazione fluida e in tempo reale, il sistema utilizza connessioni WebSocket che permettono di trasmettere l'audio generato in piccoli frammenti (chunks) non appena sono disponibili, riducendo notevolmente la latenza percepita dall'utente.

- Formato PCM Float 32: L'audio viene generato nel formato PCM (Pulse Code Modulation) a 32 bit in virgola mobile, offrendo un'elevata qualità e fedeltà del suono. Questo formato consente una rappresentazione precisa delle onde sonore, risultando in una voce chiara e naturale.

- Yield Streaming: Il sistema implementa un pattern di generazione a flusso usando yield di Python, che consente di trasmettere i frammenti audio man mano che vengono prodotti, senza dover attendere la generazione completa dell'audio.

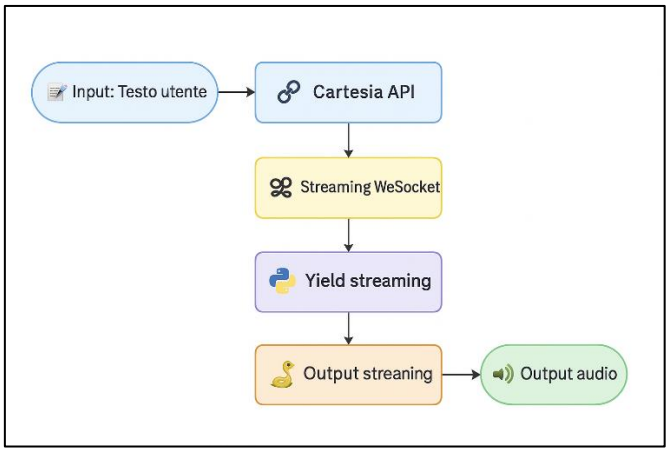



Figura 19 - Cartesia API

Il cuore del sistema è rappresentato dalla classe TTS, che gestisce l'intero processo di conversione da testo ad audio:

```
class TTS:
    def __init__(self, assistant_id):
        self.assistant_id = assistant_id
        self.voice_id = None
        self.ws = None
        self.is_connected = False

        logging.info(f"Initializing TTS for assistant_id: {assistant_id}")

        assistant = Assistant.query.get(assistant_id)

        print('Cartesia voice id: ' + assistant.cartesia_voice_id)

        if assistant:
            self.voice_id = assistant.cartesia_voice_id
            logging.debug(f"Using voice_id from assistant: {self.voice_id}")
        else:
            logging.error(f"Assistant not found with id: {assistant_id}")
            raise ValueError("Assistant not found")

        if not self.voice_id:
            self.voice_id = "79693aee-1207-4771-a01e-20c393c89e6f"
            logging.info("Using default voice_id")

        logging.debug("Initializing Cartesia client")
        self.cartesia_client = Cartesia(api_key=os.environ.get("CARTESIA_API_KEY"))
```

La fase di inizializzazione della classe TTS è fondamentale per configurare correttamente la sintesi vocale. Quando viene creata un'istanza, il costruttore accetta l'ID dell'assistente a cui sarà associata. Questo permette di recuperare dal database le informazioni specifiche dell'assistente, in particolare l'identificatore della voce Cartesia (cartesia_voice_id) che determinerà le caratteristiche vocali dell'assistente durante le conversazioni.

Il sistema implementa anche un meccanismo di fallback: se per qualche motivo l'assistente non ha una voce specifica configurata, viene automaticamente utilizzata una voce predefinita (identificata da un ID specifico nel codice). Questo garantisce che l'assistente possa sempre comunicare, anche in assenza di una configurazione vocale personalizzata.

Infine, viene inizializzato il client Cartesia utilizzando la chiave API memorizzata nelle variabili d'ambiente, stabilendo così il canale di comunicazione con il servizio di sintesi vocale.

La generazione dell'audio avviene attraverso un processo di streaming che ottimizza la reattività del sistema:

```
def get_audio_stream(self, text):
    try:
        voice = self.cartesia_client.voices.get(id=self.voice_id)

        logging.info(f"Generating audio stream for text: {text}")
        model_id = "sonic-english"
        output_format = {
            "container": "raw",
            "encoding": "pcm_f32le",
            "sample_rate": 22050
        }

        logging.debug("Creating Cartesia websocket connection")

        ws = self.cartesia_client.tts.websocket()

        try:
            audio_chunks = []  # Store chunks for local playback

            for output in ws.send(
                model_id=model_id,
                transcript=text,
                voice_id=self.voice_id,
                stream=True,
                language="it",
                output_format=output_format
            ):
                logging.debug(f"[TTS]Received output: {output.keys()}")
                if 'audio' in output.keys():
                    buffer = output['audio']
                    logging.debug(f"[TTS]Received audio chunk of size: {len(buffer)}")
                    audio_chunks.append(buffer)  # Store chunk
                    if buffer:
                        yield buffer

        except Exception as inner_e:
            logging.error(f"Inner error during TTS generation: {str(inner_e)}")
            raise
        finally:
            ws.close()

    except Exception as e:
        logging.error(f"Error in get_audio_stream: {str(e)}")
        raise
```

Questo metodo accetta come input il testo da sintetizzare e avvia un processo sofisticato per trasformarlo in un flusso audio naturale. Inizialmente, il sistema recupera le informazioni sulla voce selezionata utilizzando

l'ID memorizzato durante l'inizializzazione. Successivamente, configura i parametri tecnici della generazione audio, specificando il modello di sintesi vocale ("sonic-english") e il formato dell'output audio.

Una caratteristica particolarmente importante è la configurazione del formato audio, che utilizza un encoding PCM a 32 bit in virgola mobile con una frequenza di campionamento di 22050 Hz. Questi parametri garantiscono un equilibrio ottimale tra qualità audio e dimensione dei dati, essenziale per uno streaming fluido.

La vera magia avviene quando viene stabilita una connessione WebSocket con il servizio Cartesia. Attraverso questa connessione, il sistema invia il testo da sintetizzare insieme all'ID della voce e riceve in risposta un flusso di frammenti audio. L'utilizzo del pattern yield permette al sistema di trasmettere immediatamente ogni frammento non appena diventa disponibile, senza dover attendere la generazione completa dell'audio. Questo approccio riduce drasticamente la latenza percepita dall'utente durante la conversazione.

Il sistema include anche funzionalità di debug e test per verificare localmente la qualità dell'audio generato.

Un aspetto fondamentale del sistema è la gestione robusta degli errori che possono verificarsi durante il processo di sintesi vocale.

La gestione degli errori è implementata a vari livelli nel sistema, formando un approccio a strati che garantisce la robustezza dell'applicazione. Durante l'inizializzazione, il sistema verifica l'esistenza dell'assistente nel database e la disponibilità di un ID voce, generando log informativi o di errore appropriati. Nel processo di generazione audio, un sistema di gestione delle eccezioni a doppio livello monitora sia gli errori interni alla connessione WebSocket sia quelli esterni relativi alla comunicazione con l'API Cartesia. Tutti gli errori vengono accuratamente registrati nel sistema di logging, fornendo informazioni dettagliate che facilitano il debugging e la risoluzione dei problemi. Inoltre, le eccezioni vengono propagate verso l'alto, permettendo ai livelli superiori dell'applicazione di gestirle in modo appropriato, ad esempio informando l'utente di un problema temporaneo con la sintesi vocale.

Il sistema di sintesi vocale è progettato con un'attenzione particolare alle prestazioni e all'esperienza utente. L'utilizzo dello streaming WebSocket permette di iniziare a trasmettere l'audio non appena i primi frammenti sono disponibili, riducendo significativamente la latenza percepita. Questo approccio è particolarmente importante nelle conversazioni telefoniche, dove pause prolungate potrebbero sembrare innaturali.

### 4.11 Assistant SST

Il sistema di riconoscimento vocale, implementato in speech_to_text.py, rappresenta la componente fondamentale che permette agli assistenti virtuali di "ascoltare" e comprendere le parole degli utenti. Questo modulo trasforma i segnali audio della voce umana in testo, consentendo all'assistente di elaborare le richieste verbali e rispondere in modo appropriato, creando così un'esperienza di conversazione bidirezionale e naturale.

Il sistema di riconoscimento vocale si basa su diverse tecnologie avanzate che lavorano in sinergia per garantire un'interpretazione accurata e veloce del parlato:

- **Deepgram API**: Il cuore del sistema è rappresentato dall'integrazione con Deepgram, un servizio di riconoscimento vocale all'avanguardia basato su modelli di intelligenza artificiale. Deepgram offre una trascrizione in tempo reale ad alta precisione, con supporto multilingua e ottimizzazioni specifiche per diverse situazioni di conversazione.

- **WebSocket Streaming**: Per garantire una comunicazione fluida e reattiva, il sistema implementa uno streaming bidirezionale tramite WebSocket. Questo permette di inviare l'audio in piccoli

frammenti al servizio di trascrizione e ricevere le trascrizioni parziali in tempo reale, anche prima che l'utente abbia terminato di parlare.

- **Operazioni Asincrone**: Il sistema fa ampio uso di operazioni asincrone (async/await) tramite la libreria asyncio di Python, permettendo di gestire le comunicazioni di rete senza bloccare l'applicazione, mantenendo così la reattività durante l'elaborazione dell'audio.

- **Event-Driven Architecture**: L'architettura basata su eventi consente di reagire in tempo reale agli eventi di trascrizione, connessione e disconnessione, permettendo all'applicazione di adattarsi dinamicamente alle condizioni di comunicazione.

- **Socket.IO**: Per la comunicazione in tempo reale tra il backend e il frontend, il sistema utilizza Socket.IO, che consente di trasmettere immediatamente le trascrizioni ricevute all'interfaccia utente, creando un'esperienza fluida e reattiva.

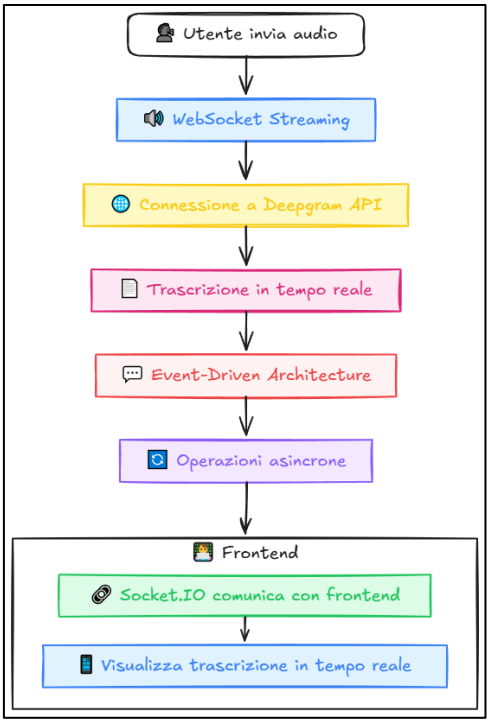



Figura 20 - SST

La classe SpeechToText gestisce l'intero ciclo di vita del processo di trascrizione:

```
class SpeechToText:
    def __init__(self, assistant_id):
        logger.info(f"Initializing SpeechToText for assistant {assistant_id}")
        deepgram_logger.info(f"=== INITIALIZING SPEECH TO TEXT ===")
        deepgram_logger.info(f"Assistant ID: {assistant_id}")

        self.assistant_id = assistant_id
        self.dg_client = None
        self.dg_connection = None
        self.transcript_parts = []
        self.is_streaming = False
        self.loop = None
        self.audio_chunks = []
```

```
        self.audio_buffer = []

        if not self.check_api_key():
            raise ValueError("Invalid or missing Deepgram API key")

        # Configure Deepgram client options
        options_dict = {"keepalive": "true", "vad_events": "true"}
        config = DeepgramClientOptions(options=options_dict)
        self.dg_client = DeepgramClient(os.getenv("DEEPGRAM_API_KEY"), config)
        logger.info("Deepgram async client initialized successfully")
```

Durante l'inizializzazione, la classe SpeechToText configura tutte le strutture dati e le connessioni necessarie per il processo di riconoscimento vocale. Ogni istanza è associata a un assistente specifico, identificato dall'assistant_id, che permette di contestualizzare e tracciare le trascrizioni. La classe inizializza vari attributi cruciali: il client Deepgram per comunicare con l'API, strutture per memorizzare parti della trascrizione e frammenti audio, e flag per monitorare lo stato della connessione streaming.

Un aspetto critico dell'inizializzazione è la verifica della chiave API di Deepgram, che viene recuperata dalle variabili d'ambiente. Se la chiave non è disponibile o non è valida, il sistema genera immediatamente un errore, impedendo l'avvio di connessioni destinate a fallire. Per motivi di sicurezza, quando la chiave viene registrata nei log, viene mascherata per mostrare solo i primi caratteri.

Il client Deepgram viene configurato con opzioni specifiche come "keepalive" per mantenere la connessione attiva e "vad_events" (Voice Activity Detection) per rilevare automaticamente quando l'utente sta parlando, ottimizzando così l'elaborazione dell'audio e riducendo il consumo di risorse durante i silenzi.

Il sistema implementa un approccio event-driven per gestire le diverse fasi del processo di riconoscimento vocale:

```
def handle_open(self, event, client):
    logger.info(f"Connection opened: {event}")
    deepgram_logger.info(f"=== DEEPGRAM CONNECTION OPENED ===")
    deepgram_logger.info(f"Assistant ID: {self.assistant_id}")
    self.is_streaming = True
    if self.audio_buffer:
        loop = asyncio.get_event_loop()
        asyncio.run_coroutine_threadsafe(self.process_buffered_chunks(), loop)

def handle_close(self, event, client):
    logger.info(f"Connection closed: {event}")
    deepgram_logger.info(f"=== DEEPGRAM CONNECTION CLOSED ===")
    self.is_streaming = False

def handle_transcript(self, client, result=None):
    deepgram_logger.info(f"=== DEEPGRAM TRANSCRIPT RECEIVED ===")
    deepgram_logger.debug(f"Transcript result: {result}")
    if result and hasattr(result, 'channel') and hasattr(result.channel, 'alternatives') and
len(result.channel.alternatives) > 0:
        sentence = result.channel.alternatives[0].transcript
        is_final = getattr(result, 'is_final', False)
        if sentence:
            self.transcript_parts.append(sentence)
            deepgram_logger.info(f"About to emit transcript event: text='{sentence}',
final={is_final}")

            from app.assistants.socket_events import socketio

            socketio.emit('stt_transcript', {
                'transcript': sentence,
                'assistant_id': self.assistant_id,
                'final': is_final
```

```
            })

            if is_final:
                self.transcript_parts = []
                deepgram_logger.info("Reset transcript parts after final message")

            deepgram_logger.info(f"Emitted transcript event: '{sentence}', Final: {is_final}")

def handle_error(self, error, client):
    logger.error(f"Deepgram error: {error}")
    deepgram_logger.error(f"=== DEEPGRAM ERROR === Error details: {error}")
```

La gestione degli eventi costituisce un componente cruciale del sistema di riconoscimento vocale. Quando viene stabilita una connessione con il servizio Deepgram, il metodo handle_open imposta il flag is_streaming su True e verifica se ci sono frammenti audio in attesa nel buffer. In caso affermativo, avvia l'elaborazione dei frammenti in sospeso, garantendo che nessun dato audio vada perso anche se è stato ricevuto prima che la connessione fosse completamente stabilita.

Quando la connessione si chiude, il metodo handle_close aggiorna lo stato del sistema di conseguenza, garantendo che non vengano più tentati invii di dati su una connessione chiusa. Questo passaggio è essenziale per la corretta gestione delle risorse e per preparare il sistema a una potenziale riapertura della connessione.

Il metodo più cruciale è handle_transcript, che viene chiamato ogni volta che Deepgram invia una parte di trascrizione. Il sistema analizza il risultato ricevuto, estraendo la trascrizione effettiva e determinando se si tratta di un risultato "finale" o di una trascrizione intermedia. Le trascrizioni intermedie permettono all'interfaccia utente di mostrare il testo in tempo reale mentre l'utente parla, migliorando notevolmente la sensazione di reattività dell'applicazione. Una volta estratta la trascrizione, il sistema la emette tramite Socket.IO al frontend, includendo l'ID dell'assistente per il corretto routing e un flag che indica se la trascrizione è finale. Quando viene ricevuta una trascrizione finale, il buffer delle parti di trascrizione viene azzerato per prepararsi alla prossima frase.

La gestione degli errori è implementata tramite il metodo handle_error, che registra dettagli sull'errore sia nel logger standard che nel logger specifico di Deepgram, facilitando il debugging e il monitoraggio del sistema.

L'inizializzazione della connessione streaming è un processo complesso che richiede la configurazione di diversi parametri e la gestione dello stato asincrono:

```
async def _initialize_connection(self):
    logger.info("Initializing connection")
    deepgram_logger.info("=== INITIALIZING DEEPGRAM CONNECTION ===")

    try:
        options = LiveOptions(
            model="nova-2-general",
            punctuate=True,
            language="it",
            smart_format=True,
            interim_results=True,
            encoding="linear16",
            sample_rate=16000,
            channels=1
        )

        # Correctly initialize the live connection with version "1"
        self.dg_connection = self.dg_client.listen.live.v("1")
        deepgram_logger.info("Created Deepgram connection")
```

```
        # Register event handlers using LiveTranscriptionEvents enum
        self.dg_connection.on(LiveTranscriptionEvents.Open, self.handle_open)
        self.dg_connection.on(LiveTranscriptionEvents.Close, self.handle_close)
        self.dg_connection.on(LiveTranscriptionEvents.Transcript, self.handle_transcript)
        self.dg_connection.on(LiveTranscriptionEvents.Error, self.handle_error)
        deepgram_logger.info("Event handlers registered")

        # Start the connection synchronously and check the result
        if not self.dg_connection.start(options):
            deepgram_logger.error("Failed to start Deepgram connection")
            return False
        deepgram_logger.info("Deepgram connection started successfully")

        # Wait for the connection to open (set by handle_open)
        for _ in range(50):
            if self.is_streaming:
                deepgram_logger.info("Connection confirmed as open")
                return True
            await asyncio.sleep(0.1)

        deepgram_logger.error("Timeout waiting for connection to open")
        return False

    except Exception as e:
        error_msg = f"Error initializing connection: {str(e)}"
        logger.error(error_msg, exc_info=True)
        deepgram_logger.error(error_msg)
        return False
```

Il metodo _initialize_connection rappresenta il cuore del processo di configurazione della connessione con il servizio Deepgram. Operando in modo asincrono, questo metodo configura tutti i parametri necessari per una trascrizione ottimale. Il modello utilizzato, "nova-2-general", è stato selezionato per la sua accuratezza generale nel riconoscimento del parlato.

Le opzioni di configurazione specificate includono l'attivazione della punteggiatura automatica, l'impostazione della lingua italiana, l'abilitazione dello smart formatting per migliorare la leggibilità delle trascrizioni (ad esempio convertendo i numeri in cifre), e l'attivazione dei risultati intermedi che permettono di ricevere trascrizioni parziali mentre l'utente sta ancora parlando. Le specifiche tecniche dell'audio includono un encoding linear16 (PCM a 16 bit), una frequenza di campionamento di 16kHz e un singolo canale audio (mono), configurazione ottimale per il riconoscimento vocale a qualità telefonica.

Dopo aver creato la connessione con la versione specifica dell'API, il sistema registra i gestori di eventi per i diversi tipi di eventi che possono verificarsi: apertura della connessione, chiusura, ricezione di trascrizioni ed errori. Una volta avviata la connessione, il sistema implementa un meccanismo di attesa attiva per confermare che la connessione sia effettivamente aperta, aspettando fino a 5 secondi (50 iterazioni con 0.1 secondi di pausa) prima di dichiarare un timeout. Questo approccio garantisce che la connessione sia completamente stabilita prima di iniziare a inviare dati audio.

La gestione degli errori è particolarmente robusta, con intercettazione di tutte le eccezioni che potrebbero verificarsi durante l'inizializzazione e registrazioni dettagliate per facilitare il debugging.

Lo streaming audio e l'elaborazione dei frammenti:

```
def start_stream(self):
    logger.info("Starting stream")
    deepgram_logger.info("=== STARTING STREAM ===")
    deepgram_logger.info(f"Assistant ID: {self.assistant_id}")

    if self.is_streaming:
        logger.warning("Stream already started")
        deepgram_logger.warning("Stream already started")
```

```
        return True

    try:
        # Reset all state
        self.audio_chunks = []
        self.audio_buffer = []
        self.dg_connection = None

        # Create or get the event loop
        try:
            self.loop = asyncio.get_event_loop()
            if self.loop.is_closed():
                self.loop = asyncio.new_event_loop()
                asyncio.set_event_loop(self.loop)
        except RuntimeError:
            self.loop = asyncio.new_event_loop()
            asyncio.set_event_loop(self.loop)

        # Run the async connection initialization
        connection_success = self.loop.run_until_complete(self._initialize_connection())

        if not connection_success:
            logger.error("Failed to initialize Deepgram connection")
            deepgram_logger.error("Failed to start stream - connection initialization failed")
            return False

        logger.info("Stream initialized successfully")
        deepgram_logger.info("Stream initialized successfully")
        return True
    except Exception as e:
        error_msg = f"Error starting stream: {str(e)}"
        logger.error(error_msg, exc_info=True)
        deepgram_logger.error(error_msg)
        return False

async def process_chunk(self, audio_chunk):
    logger.info("Processing chunk")
    if not self.is_streaming:
        logger.info("STT stream not started, buffering audio chunk")
        self.audio_buffer.append(audio_chunk)
        return

    try:
        if isinstance(audio_chunk, dict) and 'audio' in audio_chunk:
            audio_data = audio_chunk['audio']
        elif isinstance(audio_chunk, str):
            audio_data = base64.b64decode(audio_chunk)
        elif isinstance(audio_chunk, (bytes, bytearray)):
            audio_data = audio_chunk
        elif hasattr(audio_chunk, 'tolist') and callable(audio_chunk.tolist):
            audio_data = np.array(audio_chunk.tolist(), dtype=np.int16).tobytes()
        else:
            audio_data = bytes(audio_chunk)

        deepgram_logger.info(f"Audio chunk length: {len(audio_data)} bytes")
        self.audio_chunks.append(audio_data)

        if len(audio_data) > 0:
            # Send audio data synchronously
            self.dg_connection.send(audio_data)
            deepgram_logger.info(f"Sent {len(audio_data)} bytes to Deepgram")
    except Exception as e:
        error_msg = f"Error processing chunk: {str(e)}"
```

```
        logger.error(error_msg, exc_info=True)
        deepgram_logger.error(error_msg)
        raise
```

La gestione dello streaming audio inizia con il metodo start_stream, che prepara il sistema per ricevere e processare l'audio dell'utente. Prima di avviare un nuovo stream, il metodo verifica che non ci sia già uno stream attivo, evitando duplicazioni e potenziali conflitti. Se nessuno stream è attivo, il sistema reimposta tutti gli stati relativi all'audio e inizializza un nuovo loop di eventi asyncio, componente fondamentale per la gestione delle operazioni asincrone.

L'inizializzazione della connessione viene eseguita utilizzando il metodo _initialize_connection discusso in precedenza, convertendo l'operazione asincrona in sincrona tramite `run_until_complete`. Questo approccio garantisce che la connessione sia completamente stabilita prima di procedere, migliorando la robustezza del sistema.

L'elaborazione effettiva dei frammenti audio avviene nel metodo asincrono process_chunk. Questo metodo accetta frammenti audio in vari formati (dizionari, stringhe base64, oggetti bytes o array NumPy) e li converte nel formato appropriato per l'invio a Deepgram. Questa flessibilità nell'accettare diversi formati di input facilita l'integrazione con varie fonti audio, dalle API web ai dispositivi hardware.

Quando un frammento audio arriva prima che lo stream sia iniziato, il sistema lo memorizza temporaneamente in un buffer, assicurando che nessun dato venga perso. Una volta che la connessione viene stabilita, i frammenti in buffer vengono automaticamente elaborati in ordine. Ogni frammento audio elaborato viene anche salvato nella lista audio_chunks, che viene utilizzata per la registrazione completa della conversazione.

L'invio del frammento audio a Deepgram avviene in modo sincrono tramite il metodo send della connessione Deepgram, che gestisce internamente la trasmissione attraverso WebSocket. Il sistema registra la dimensione dei frammenti inviati, informazione utile per diagnosticare problemi di prestazioni o di connettività.

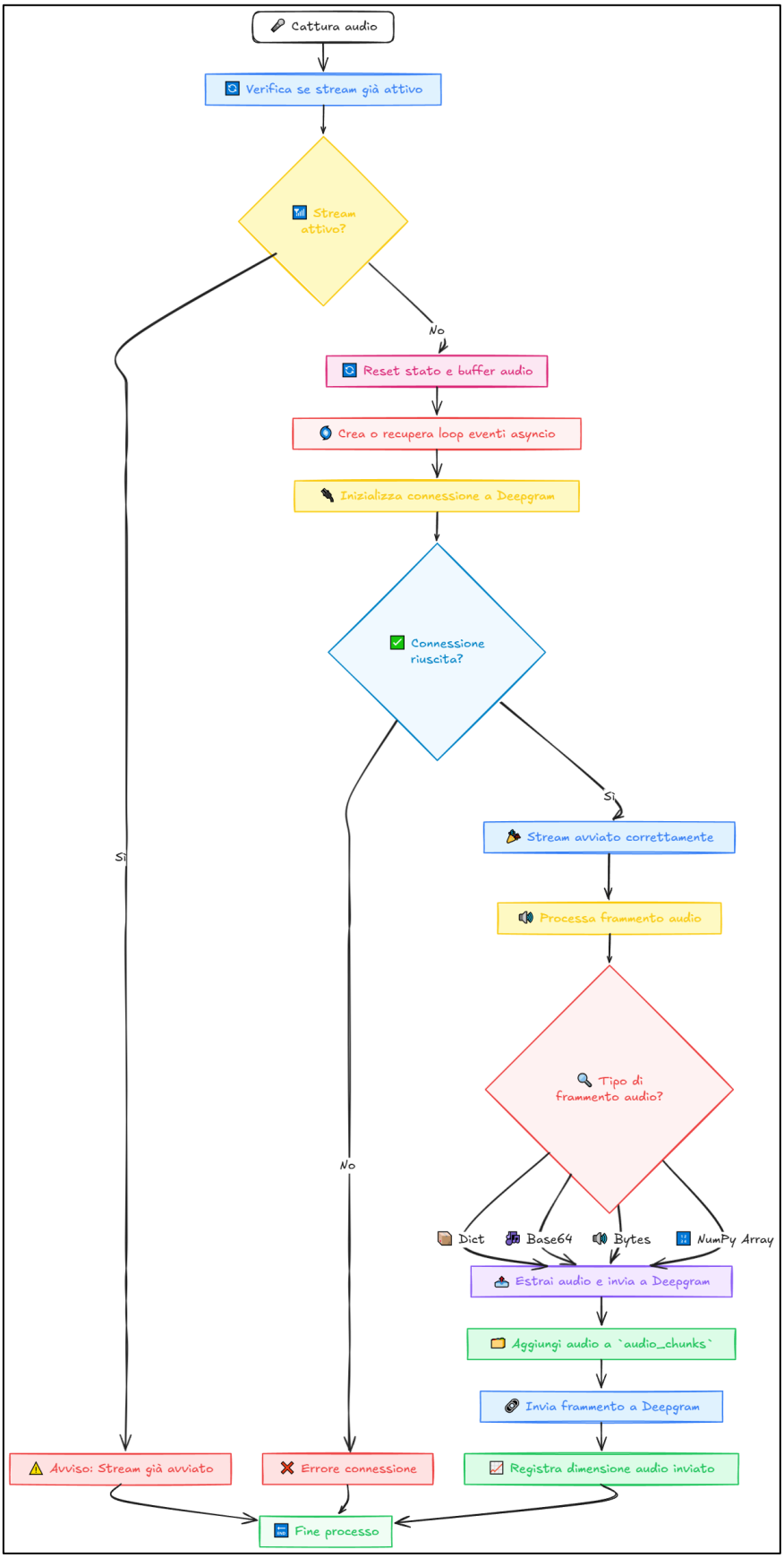



Figura 21 - SST

### 4.11.1 Sintesi

Il processo completo di riconoscimento vocale nell'applicazione segue un flusso ben definito:

Quando un utente inizia una conversazione con un assistente virtuale, il sistema avvia una sessione di riconoscimento vocale chiamando il metodo start_stream della classe `SpeechToText`. Questo inizializza la connessione con Deepgram e prepara il sistema a ricevere l'audio.

Man mano che l'utente parla, l'audio viene catturato dal dispositivo, compresso e inviato al server attraverso una connessione WebSocket. Qui, i frammenti audio vengono ricevuti e immediatamente inoltrati al sistema di riconoscimento vocale tramite il metodo process_chunk.

Il sistema invia questi frammenti a Deepgram in tempo reale, che analizza l'audio utilizzando modelli di intelligenza artificiale avanzati. Deepgram inizia a restituire trascrizioni parziali anche mentre l'utente sta ancora parlando, permettendo all'interfaccia di mostrare il testo in tempo reale.

Queste trascrizioni intermedie vengono emesse al frontend attraverso Socket.IO, creando un'esperienza responsiva per l'utente. Quando Deepgram identifica una pausa naturale nel discorso, contrassegna la trascrizione come "final", permettendo al sistema di consolidare quella parte della conversazione.

Le trascrizioni finali vengono processate dall'assistente virtuale, che può così comprendere la richiesta dell'utente e formulare una risposta appropriata. Questa risposta viene poi convertita in audio tramite il sistema TTS e restituita all'utente, completando il ciclo di comunicazione.

Quando la conversazione termina, il metodo stop_stream viene chiamato per chiudere la connessione con Deepgram, salvare l'audio per futuri riferimenti e liberare le risorse di sistema.

## 4.12 Chat Assistants

Il sistema di chat rappresenta un'interfaccia di comunicazione fondamentale, usata per testare i componenti delle chiamate.
Permette agli utenti di interagire testualmente con gli assistenti virtuali. Questa funzionalità consente conversazioni naturali e fluide con l'assistente, sfruttando modelli di linguaggio avanzati per generare risposte contestuali e pertinenti. La chat si integra perfettamente con gli altri sistemi, come il riconoscimento vocale (STT) e la sintesi vocale (TTS), creando un'esperienza di comunicazione completa e multimodale.

Il sistema di chat si basa su diverse tecnologie chiave che lavorano in sinergia per offrire un'esperienza utente reattiva e fluida:

- Socket.IO: Libreria che abilita la comunicazione bidirezionale e in tempo reale tra backend e frontend. Socket.IO gestisce automaticamente riconnessioni, fallback e garantisce una trasmissione affidabile dei messaggi, essenziale per l'esperienza conversazionale.

- EventLet: Libreria di programmazione concorrente che consente al server di gestire simultaneamente multiple connessioni e richieste, fondamentale per scalare a numerose sessioni di chat contemporanee.

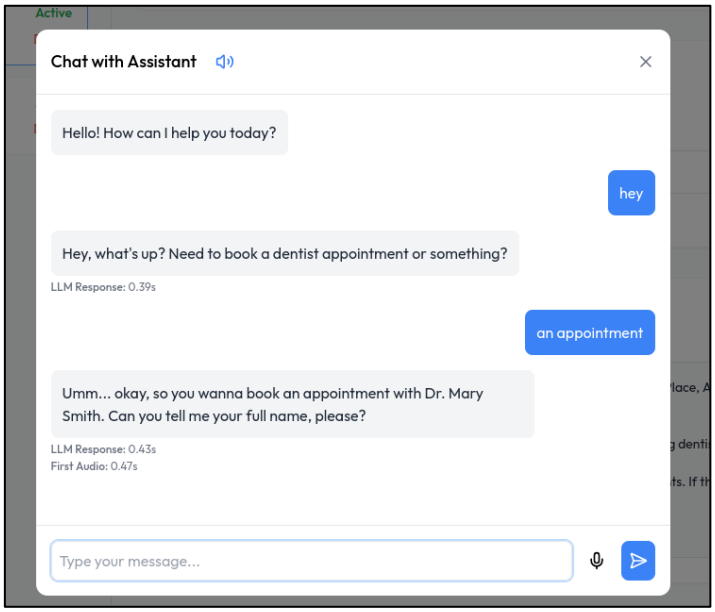



Figura 22 - Chat Assistant

### 4.12.1 Backend

La comunicazione in tempo reale tra frontend e backend è gestita attraverso eventi Socket.IO, implementati nel modulo socket_events.py:

```
@socketio.on('chat_message')
def handle_message(data):
    logging.info(f"Received chat_message: {data}")
    try:
        assistant_id = data.get('assistant_id')
        question = data.get('question')
        use_tts = data.get('use_tts', False)

        if not assistant_id:
            socketio.emit('error', {'message': 'Missing assistant_id'})
            return

        assistant = Assistant.query.get_or_404(assistant_id)

        if assistant_id not in assistant_llm_cache:
            assistant_llm_cache[assistant_id] = AssistantLLM(assistant_id)

        if question:
            llm_start_time = time.time()
            assistant_llm = assistant_llm_cache[assistant_id]
            response = assistant_llm.get_response(question)
            llm_response_time = time.time() - llm_start_time

            socketio.emit('chat_response', {
                'content': response,
                'assistant_id': assistant_id,
                'llm_response_time': llm_response_time
            })

            if use_tts:
                handle_tts(assistant_id, response, tts_cache)
```

```
    except Exception as e:
        logging.error(f"Error in chat_message handler: {str(e)}")
        socketio.emit('error', {'message': str(e)})
```

Il metodo handle_message è il centro nevralgico della funzionalità di chat. Quando un messaggio viene ricevuto dal frontend, questo metodo si occupa di elaborarlo e di generare una risposta appropriata. Il primo passo è estrarre dal messaggio l'ID dell'assistente, il testo della domanda e un flag che indica se utilizzare la sintesi vocale per la risposta.

Se l'ID dell'assistente non è fornito, il sistema emette un errore e termina l'elaborazione. Altrimenti, verifica che l'assistente esista nel database e, se necessario, crea una nuova istanza di AssistantLLM per quell'assistente. Questo approccio di cache migliora significativamente le prestazioni, poiché evita di reinizializzare l'assistente ad ogni messaggio.

A questo punto, il sistema misura il tempo di inizio elaborazione, passa la domanda all'istanza dell'assistente per generare una risposta, e calcola il tempo totale impiegato. La risposta viene quindi inviata al client tramite l'evento chat_response, includendo il contenuto, l'ID dell'assistente e il tempo di risposta per scopi di monitoraggio.

Se l'utente ha richiesto la sintesi vocale, il sistema invoca il metodo handle_tts per convertire la risposta testuale in audio e trasmettere i frammenti audio al client.

```
def handle_tts(assistant_id, response, tts_cache):
    try:
        if assistant_id not in tts_cache:
            tts_cache[assistant_id] = TTS(assistant_id)

        tts = tts_cache[assistant_id]
        tts_start_time = time.time()
        audio_stream = tts.get_audio_stream(response)

        first_chunk = True
        for chunk in audio_stream:
            if chunk and len(chunk) > 0:
                if first_chunk:
                    first_chunk_time = time.time() - tts_start_time
                    socketio.emit('audio_chunk', {
                        'audio': chunk,
                        'assistant_id': assistant_id,
                        'final': False,
                        'first_chunk_time': first_chunk_time
                    })
                    first_chunk = False
                else:
                    socketio.emit('audio_chunk', {
                        'audio': chunk,
                        'assistant_id': assistant_id,
                        'final': False
                    })

        socketio.emit('audio_chunk', {
            'assistant_id': assistant_id,
            'final': True
        })

    except Exception as e:
        logging.error(f"TTS error: {str(e)}")
```

```
        socketio.emit('error', {'message': f'TTS error: {str(e)}'})
```

Il metodo handle_tts gestisce la conversione del testo in audio utilizzando il servizio di sintesi vocale. Come per l'assistente LLM, il sistema utilizza una cache per evitare di creare nuove istanze TTS per ogni richiesta, migliorando così le prestazioni.

Il processo inizia misurando il tempo di inizio, quindi richiede uno stream audio dalla classe TTS. Lo stream restituisce i frammenti audio uno alla volta, consentendo un'esperienza di streaming fluida. Per il primo frammento, il sistema calcola il tempo trascorso dall'inizio della sintesi e lo include nelle informazioni inviate al client. Questo è utile per misurare la reattività del sistema TTS.

Ogni frammento audio viene inviato al client tramite l'evento audio_chunk, con un flag final impostato a false per indicare che ci sono altri frammenti in arrivo. Quando tutti i frammenti sono stati inviati, il sistema emette un ultimo evento audio_chunk con final impostato a true, segnalando al client che la trasmissione audio è completata.

```
@socketio.on('start_stt')
def handle_start_stt():
    try:
        assistant_id = request.args.get('assistant_id')
        if not assistant_id:
            logging.warning("No assistant_id provided for start_stt")
            socketio.emit('error', {'message': 'No assistant_id provided'})
            return

        # Create new STT instance if not exists
        if assistant_id not in stt_cache:
            stt_cache[assistant_id] = SpeechToText(assistant_id)

        speech_to_text = stt_cache[assistant_id]

        def start_stream_task():
            try:
                loop = asyncio.new_event_loop()
                asyncio.set_event_loop(loop)
                try:
                    loop.run_until_complete(speech_to_text.start_stream())
                finally:
                    loop.close()
            except Exception as e:
                logging.error(f"Error starting STT stream: {str(e)}")
                socketio.emit('error', {'message': str(e)})

        eventlet.spawn(start_stream_task)
        socketio.emit('stt_started', {'status': 'ready', 'assistant_id': assistant_id})

    except Exception as e:
        logging.error(f"Error in start_stt: {str(e)}")
        socketio.emit('error', {'message': str(e)})
```

Il metodo handle_start_stt gestisce l'avvio del riconoscimento vocale quando un utente decide di utilizzare l'input vocale anziché testuale. Come prima cosa, estrae l'ID dell'assistente dai parametri della richiesta e verifica che sia presente.

Se l'assistente non ha ancora un'istanza SpeechToText nella cache, ne crea una nuova. Poi avvia il processo di stream asincrono all'interno di un nuovo task eventlet. Questo approccio è fondamentale perché l'avvio dello stream è un'operazione potenzialmente lunga che non deve bloccare il thread principale del server.

L'utilizzo di asyncio all'interno del task eventlet permette di gestire correttamente le operazioni asincrone necessarie per il riconoscimento vocale. Una volta avviato lo stream, il sistema notifica il client tramite l'evento stt_started, indicando che è pronto a ricevere frammenti audio.

```
@socketio.on('stt_audio_chunk')
def handle_audio_chunk(data):
    assistant_id = data.get('assistant_id')
    audio_data = data.get('audio')

    try:
        if assistant_id and audio_data:
            if assistant_id in stt_cache:
                speech_to_text = stt_cache[assistant_id]
                eventlet.spawn(process_stt_chunk, speech_to_text, {
                    'audio': audio_data
                })
            else:
                # Automatically restart STT if not in cache
                stt_cache[assistant_id] = SpeechToText(assistant_id)
                speech_to_text = stt_cache[assistant_id]

                def start_stream_task():
                    try:
                        loop = asyncio.new_event_loop()
                        asyncio.set_event_loop(loop)
                        try:
                            loop.run_until_complete(speech_to_text.start_stream())
                        finally:
                            loop.close()
                    except Exception as e:
                        logging.error(f"Error starting STT stream: {str(e)}")
                        socketio.emit('error', {'message': str(e)})

                eventlet.spawn(start_stream_task)
                eventlet.spawn(process_stt_chunk, speech_to_text, {
                    'audio': audio_data
                })
                socketio.emit('stt_started', {'status': 'ready', 'assistant_id': assistant_id})
    except Exception as e:
        logging.error(f"Exception in handle_audio_chunk: {str(e)}")
        socketio.emit('error', {'message': f'Error processing audio: {str(e)}'})
```

Il metodo handle_audio_chunk riceve i frammenti audio dal client e li invia al sistema di riconoscimento vocale. Estrae l'ID dell'assistente e i dati audio dal messaggio ricevuto, quindi verifica che entrambi siano presenti.

Se esiste già un'istanza SpeechToText per l'assistente nella cache, il sistema avvia un nuovo task eventlet per processare il frammento audio. Se invece l'istanza non esiste (ad esempio perché la cache è stata svuotata o a causa di problemi di rete), il sistema crea automaticamente una nuova istanza e avvia contemporaneamente due task: uno per inizializzare lo stream e uno per processare il frammento audio. Questo approccio resiliente garantisce che il sistema possa riprendersi da eventuali interruzioni senza richiedere interventi manuali.

```python
@socketio.on('stop_stt')
def handle_stop_stt():
    try:
        assistant_id = request.args.get('assistant_id')
        if not assistant_id:
            logging.warning("No assistant_id provided for stop_stt")
            socketio.emit('error', {'message': 'No assistant_id provided'})
            return

        if assistant_id in stt_cache:
            speech_to_text = stt_cache[assistant_id]

            def stop_stream_task():
                try:
                    loop = asyncio.new_event_loop()
                    asyncio.set_event_loop(loop)
                    try:
                        loop.run_until_complete(speech_to_text.stop_stream())
                    finally:
                        loop.close()
                    logging.info(f"STT stream stopped for assistant {assistant_id}")

                    # Don't remove from cache here - only on disconnect

                except Exception as e:
                    logging.error(f"Error stopping STT stream: {str(e)}")

            eventlet.spawn(stop_stream_task)
            socketio.emit('stt_stopped', {'status': 'stopped', 'assistant_id': assistant_id})
        else:
            logging.warning(f"No STT stream found for assistant {assistant_id}")
            socketio.emit('stt_stopped', {'status': 'not_found', 'assistant_id': assistant_id})

    except Exception as e:
        logging.error(f"Error stopping STT: {str(e)}")
        socketio.emit('error', {'message': f'Error stopping STT: {str(e)}'})
```

Il metodo handle_stop_stt gestisce la terminazione del riconoscimento vocale quando l'utente smette di parlare o decide di passare all'input testuale. Come per gli altri metodi, estrae l'ID dell'assistente e verifica che esista un'istanza STT nella cache.

Se l'istanza esiste, avvia un nuovo task eventlet per fermare lo stream in modo asincrono. Questo è importante perché la terminazione dello stream può includere operazioni come il salvataggio dell'audio raccolto o la finalizzazione delle trascrizioni, che non devono bloccare il thread principale. Al termine, emette un evento stt_stopped per notificare al client che il riconoscimento vocale è stato fermato.

È interessante notare che il sistema deliberatamente non rimuove l'istanza STT dalla cache in questo punto, ma solo quando il client si disconnette completamente. Questo perché l'utente potrebbe voler riattivare il riconoscimento vocale in seguito durante la stessa sessione, e mantenere l'istanza in cache migliora la velocità di ripresa.

```python
def process_stt_chunk(speech_to_text, audio_chunk_data):
    try:
        if not speech_to_text.is_streaming:
            logging.warning("STT stream not started, ignoring audio chunk")
            return
```

```
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)
        try:
            loop.run_until_complete(speech_to_text.process_chunk(audio_chunk_data))
        finally:
            loop.close()
    except Exception as e:
        logging.error(f"Error in process_stt_chunk: {str(e)}")
```

Il metodo process_stt_chunk è una funzione di supporto utilizzata per elaborare i frammenti audio ricevuti dal client nel contesto del riconoscimento vocale. Questo metodo viene eseguito in un task eventlet separato per evitare di bloccare il thread principale del server.

Prima di tutto, il metodo verifica che lo stream STT sia effettivamente attivo; in caso contrario, ignora il frammento audio e registra un avviso nei log. Se lo stream è attivo, il metodo crea un nuovo loop asyncio e invoca il metodo process_chunk dell'istanza SpeechToText per elaborare il frammento audio.

L'elaborazione avviene in modo asincrono, permettendo al sistema di gestire simultaneamente molti frammenti audio da diversi client senza degradare le prestazioni. Al termine dell'elaborazione, il loop asyncio viene chiuso per liberare le risorse.

È importante notare che i risultati del riconoscimento vocale non vengono gestiti direttamente in questo metodo. Invece, l'istanza SpeechToText emette eventi stt_transcript quando rileva parole o frasi nel flusso audio, e questi eventi vengono ricevuti e gestiti dal frontend.

Il sistema implementa anche gestori per altri eventi come connect, che registra la connessione di un nuovo client, e chat_cleanup, che pulisce le risorse quando una sessione di chat viene terminata:

```
@socketio.on('chat_cleanup')
def handle_chat_cleanup(data):
    try:
        assistant_id = data.get('assistant_id')
        if assistant_id:
            # Pulisci TTS cache
            if assistant_id in tts_cache:
                del tts_cache[assistant_id]

            # Pulisci LLM cache
            if assistant_id in assistant_llm_cache:
                del assistant_llm_cache[assistant_id]

            # Pulisci STT se esiste
            if assistant_id in stt_cache:
                speech_to_text = stt_cache[assistant_id]

                def cleanup_task():
                    try:
                        loop = asyncio.new_event_loop()
                        asyncio.set_event_loop(loop)
                        try:
                            loop.run_until_complete(speech_to_text.stop_stream())
                        finally:
                            loop.close()

                        del stt_cache[assistant_id]
                        logging.info(f"Cleaned up resources for assistant {assistant_id}")
```

```
                except Exception as e:
                    logging.error(f"Error in cleanup task: {str(e)}")

            eventlet.spawn(cleanup_task)

    except Exception as e:
        logging.error(f"Error in chat cleanup handler: {str(e)}")
```

Questo evento di pulizia è fondamentale per la gestione efficiente delle risorse del server, poiché garantisce che istanze non più necessarie vengano correttamente rimosse dalla memoria. La pulizia coinvolge la rimozione delle istanze dalla cache TTS, dalla cache LLM e l'arresto corretto di eventuali stream STT attivi. L'utilizzo di eventlet.spawn permette di eseguire operazioni potenzialmente lunghe in modo asincrono, senza bloccare il thread principale del server.

### 4.12.2 Frontend

Il componente principale ChatModal.vue che gestisce l'interfaccia utente e l'interazione con il backend. L'interfaccia utente della chat è strutturata in tre sezioni principali: l'intestazione con il titolo e i controlli per attivare/disattivare la sintesi vocale, la sezione centrale che visualizza i messaggi scambiati tra l'utente e l'assistente, e il footer con i controlli per l'input testuale e vocale.

I messaggi sono visualizzati in modo differenziato in base al mittente: i messaggi dell'utente appaiono allineati a destra con sfondo blu, mentre quelli dell'assistente sono allineati a sinistra con sfondo grigio. Per i messaggi dell'assistente, vengono visualizzati anche i tempi di risposta del modello linguistico e, se applicabile, il tempo necessario per generare l'audio.

La sezione di input include un campo di testo, un pulsante per attivare il riconoscimento vocale (che cambia colore e pulsa quando attivo) e un pulsante di invio. Questa combinazione offre versatilità all'utente, permettendo di scegliere il metodo di comunicazione più comodo in base al contesto.

La logica che gestisce queste interazioni è implementata nella parte script del componente:

```
const sendMessage = async () => {
  if (!newMessage.value.trim()) return

  try {
    const messageText = newMessage.value
    messages.value.push({ sender: 'user', content: messageText })
    newMessage.value = ''
    scrollToBottom()

    const assistantIdNumber = Number(props.assistantId)
    if (isNaN(assistantIdNumber)) {
      throw new Error('Invalid assistant ID')
    }

    messages.value.push({ sender: 'assistant', content: '' })

    // Send message through socket
    socket.emit('chat_message', {
      assistant_id: assistantIdNumber,
      question: messageText,
      use_tts: ttsEnabled.value
    })
```

```
  } catch (error) {
    console.error('Error sending message:', error)
    messages.value.push({
      sender: 'assistant',
      content: 'Sorry, there was an error processing your message.'
    })
    scrollToBottom()
  }
}
```

La funzione sendMessage gestisce l'invio dei messaggi al backend. Verifica innanzitutto che il messaggio non sia vuoto, quindi lo aggiunge all'array dei messaggi visualizzati, svuota il campo di input e fa scorrere la visualizzazione verso il basso per mostrare il messaggio appena inviato. Aggiunge anche immediatamente un messaggio vuoto da parte dell'assistente, che verrà riempito non appena arriva la risposta, creando così un'esperienza più reattiva per l'utente.

Il messaggio viene poi inviato al backend tramite l'evento Socket.IO chat_message, includendo l'ID dell'assistente, il testo della domanda e una flag che indica se utilizzare la sintesi vocale. In caso di errore, viene mostrato un messaggio di scusa generico per mantenere l'esperienza utente fluida anche in presenza di problemi.

Al montaggio del componente, viene stabilita la connessione socket e vengono configurati i gestori per vari eventi. L'evento più importante è chat_response, che aggiorna il contenuto dell'ultimo messaggio dell'assistente con la risposta ricevuta dal backend, includendo anche il tempo di risposta del modello linguistico.

Il componente gestisce anche eventi come error per mostrare messaggi di errore all'utente, e transcript per aggiornare i messaggi con le trascrizioni vocali quando l'utente utilizza il riconoscimento vocale come input.

All'unmount del componente, viene eseguita una pulizia completa delle risorse, disattivando tutti i listener degli eventi socket, chiudendo eventuali contesti audio aperti, fermando il registratore multimediale e terminando qualsiasi sintesi vocale in corso. Questa pulizia accurata previene memory leak e garantisce che le risorse vengano rilasciate correttamente.

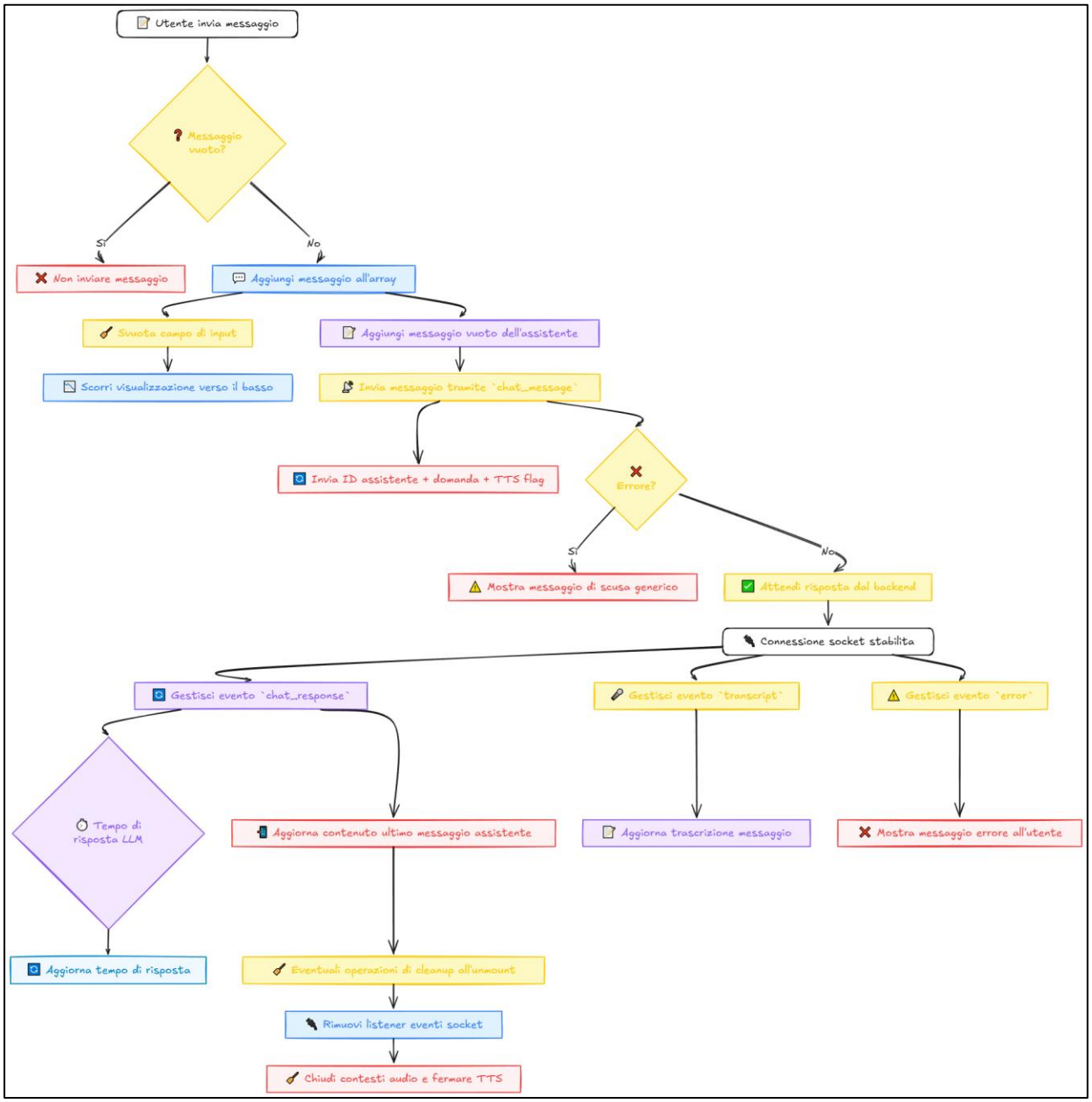



Figura 23 - Frontend Chat

### 4.13 Knowledge Base

L'elaborazione delle basi di conoscenza è un'operazione computazionalmente intensiva che potrebbe bloccare il thread principale del server. Per evitare questo problema, l'applicazione utilizza Celery, un sistema di codice asincrono.

```
def make_celery(app):
    celery = Celery(
        app.import_name,
        backend=app.config['CELERY_RESULT_BACKEND'],
        broker=app.config['CELERY_BROKER_URL']
```

```
    )

    celery.conf.update(app.config)

    class ContextTask(celery.Task):
        def __call__(self, *args, **kwargs):
            with app.app_context():
                return self.run(*args, **kwargs)

    celery.Task = ContextTask
    celery.autodiscover_tasks(['app.tasks'])

    return celery
```

Celery viene configurato per utilizzare Redis come broker di messaggi e backend di risultati. Questa combinazione offre un'architettura robusta e scalabile per l'elaborazione di task asincroni. La funzione make_celery crea un'istanza di Celery integrata con il contesto dell'applicazione Flask, garantendo che ogni task abbia accesso alle stesse risorse disponibili nell'applicazione principale.

La classe ContextTask è particolarmente importante: estende la classe Task di Celery per eseguire ogni task all'interno di un contesto di applicazione Flask. Questo permette ai task di accedere a componenti come il database e le configurazioni senza dover reinizializzare queste risorse.

Il sistema di basi di conoscenza è strutturato attorno a un modello di dati ben definito:

```
assistant_base_knowledge = db.Table('assistant_base_knowledge',
    db.Column('assistant_id', db.Integer, db.ForeignKey('assistant.id'), primary_key=True),
    db.Column('base_knowledge_id', db.Integer, db.ForeignKey('base_knowledge.id'), primary_key=True)
)

class BaseKnowledge(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(128), nullable=False)
    description = db.Column(db.Text, nullable=True)
    created_at = db.Column(db.DateTime, server_default=db.func.now())
    updated_at = db.Column(db.DateTime, server_default=db.func.now(), onupdate=db.func.now())
    last_loaded = db.Column(db.DateTime, nullable=True)
    needs_reload = db.Column(db.Boolean, default=False)

    profile_id = db.Column(db.Integer, db.ForeignKey('profile.id'), nullable=False)
    profile = db.relationship('Profile', backref='base_knowledge')

    folder_path = db.Column(db.String(256), nullable=False)

    files = db.relationship('BaseKnowledgeFile', backref='parent_knowledge', lazy=True)

    assistants = db.relationship('Assistant', secondary=assistant_base_knowledge,
                                 backref=db.backref('knowledge_bases', lazy='dynamic'))

class BaseKnowledgeFile(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(128), nullable=False)
    description = db.Column(db.Text, nullable=True)
    created_at = db.Column(db.DateTime, server_default=db.func.now())
    updated_at = db.Column(db.DateTime, server_default=db.func.now(), onupdate=db.func.now())

    base_knowledge_id = db.Column(db.Integer, db.ForeignKey('base_knowledge.id'), nullable=False)
    base_knowledge = db.relationship('BaseKnowledge', backref='knowledge_files')

    file_path = db.Column(db.String(256), nullable=False)
    file_type = db.Column(db.String(128), nullable=False)
```

```
    file_size = db.Column(db.Integer, nullable=False)

class TaskStatusBaseKnowledge(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    task_id = db.Column(db.String(128), unique=True, nullable=False)
    status = db.Column(db.String(50), nullable=False)  # PENDING, STARTED, SUCCESS, FAILURE
    progress = db.Column(db.Integer, default=0)  # Progress from 0 to total_steps
    total_steps = db.Column(db.Integer, default=4)  # Total number of steps
    status_message = db.Column(db.String(255))  # Current status message
    result = db.Column(db.JSON, nullable=True)
    error = db.Column(db.Text, nullable=True)
    created_at = db.Column(db.DateTime, server_default=db.func.now())
    updated_at = db.Column(db.DateTime, server_default=db.func.now(), onupdate=db.func.now())
    base_knowledge_id = db.Column(db.Integer, db.ForeignKey('base_knowledge.id'))
```

- BaseKnowledge: Rappresenta una base di conoscenza che può essere associata a uno o più assistenti. Memorizza informazioni come nome, descrizione, data di creazione e ultimo caricamento. Il campo needs_reload è particolarmente importante, in quanto segnala se la base di conoscenza richiede una rigenerazione degli embedding a causa di modifiche ai file.

- BaseKnowledgeFile: Rappresenta un singolo file all'interno di una base di conoscenza. Memorizza dettagli come nome, percorso, tipo (PDF, TXT) e dimensione del file. Ogni file è collegato a una specifica base di conoscenza attraverso una relazione uno-a-molti.

- TaskStatusBaseKnowledge: Tiene traccia dello stato dell'elaborazione asincrona di una base di conoscenza. Include campi per monitorare il progresso (da 0 al numero totale di passaggi), messaggi di stato, risultati e eventuali errori. Questo permette all'interfaccia utente di mostrare in tempo reale lo stato di avanzamento dell'elaborazione.

- La tabella di join assistant_base_knowledge implementa una relazione molti-a-molti tra assistenti e basi di conoscenza, permettendo a ciascun assistente di attingere a più fonti di conoscenza e a ciascuna base di conoscenza di essere utilizzata da più assistenti.

Il processamento di una base di conoscenza è un task Celery che converte documenti testuali in embedding vettoriali che possono essere utilizzati per ricerche semantiche:

```
@celery.task(bind=True)
def process_base_knowledge(self, base_knowledge_id):
    start_time = time.time()
    logger.info(f"Starting process_base_knowledge task for base_knowledge_id: {base_knowledge_id}")

    session = db.session
    task_status = None

    try:
        # Initialize task status
        task_status = TaskStatusBaseKnowledge(
            task_id=self.request.id,
            status='STARTED',
            progress=0,
            total_steps=4,
            base_knowledge_id=base_knowledge_id
        )
        session.add(task_status)
        session.commit()

        base_knowledge = session.query(BaseKnowledge).get(base_knowledge_id)
        if not base_knowledge:
            raise ValueError(f"Base knowledge {base_knowledge_id} not found")
```

```
        # Step 1: Document loading - avoid ProcessPoolExecutor for now
        task_status.status_message = "Loading documents"
        task_status.progress = 1
        session.commit()

        documents = []
        files_path = Path(BASE_DIR) / base_knowledge.folder_path / 'files'

        # Process files sequentially for now
        for file in base_knowledge.files:
            file_path = files_path / file.name
            try:
                if file.file_type == 'pdf':
                    loader = PyPDFLoader(str(file_path))
                    documents.extend(loader.load())
                elif file.file_type == 'txt':
                    loader = TextLoader(str(file_path))
                    documents.extend(loader.load())
            except Exception as e:
                logger.error(f"Error loading file {file.name}: {str(e)}")

        # Step 2: Document splitting - use single-threaded approach for debugging
        task_status.status_message = "Splitting documents"
        task_status.progress = 2
        session.commit()

        # Optimize chunk size based on document length
        avg_doc_length = sum(len(doc.page_content) for doc in documents) / len(documents) if
documents else 1000
        chunk_size = min(max(int(avg_doc_length / 3), 500), 2000)  # Dynamic chunk size between 500-
2000

        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=chunk_size,
            chunk_overlap=int(chunk_size * 0.1),
            length_function=len
        )

        # Process documents sequentially
        splits = []
        for doc in documents:
            splits.extend(text_splitter.split_documents([doc]))

        # Step 3 & 4: Embedding creation and storage
        task_status.status_message = "Creating embeddings and storing"
        task_status.progress = 3
        session.commit()

        persist_dir = Path(BASE_DIR) / base_knowledge.folder_path / 'chroma_db'
        persist_dir.mkdir(exist_ok=True)

        batch_size = 50
        embeddings = OpenAIEmbeddings()
        vectorstore = Chroma(
            persist_directory=str(persist_dir),
            embedding_function=embeddings
        )

        for i in range(0, len(splits), batch_size):
            batch = splits[i:i + batch_size]
            vectorstore.add_documents(documents=batch)
```

```
            # Update progress
            progress = min(3 + ((i + batch_size) / len(splits)), 4)
            task_status.progress = progress
            session.commit()

        vectorstore.persist()

        # Update completion status
        base_knowledge.last_loaded = datetime.now()
        base_knowledge.needs_reload = False
        task_status.status = 'SUCCESS'
        task_status.progress = task_status.total_steps
        session.commit()

        total_time = time.time() - start_time
        logger.info(f"Task completed successfully. Total execution time: {total_time:.2f} seconds")
        return {'status': 'success', 'processing_time': total_time}
```

Il processo di elaborazione delle basi di conoscenza è diviso in quattro fasi principali:

- Caricamento dei documenti: I file caricati (PDF e TXT) vengono letti utilizzando i loader appropriati della libreria LangChain. Questa fase estrae il contenuto testuale da ogni documento.
- Suddivisione dei documenti: Il testo estratto viene suddiviso in "chunk" (frammenti) di dimensioni ottimali per l'elaborazione. La dimensione di questi chunk viene determinata dinamicamente in base alla lunghezza media dei documenti, con un valore compreso tra 500 e 2000 caratteri. Viene applicata anche una sovrapposizione del 10% tra i chunk per garantire che le informazioni a cavallo tra due frammenti non vengano perse.
- Creazione degli embedding: Ogni chunk di testo viene convertito in un vettore numerico (embedding) utilizzando il modello OpenAI. Questi vettori catturano il significato semantico del testo, permettendo ricerche basate sulla similitudine concettuale piuttosto che sulla semplice corrispondenza di parole chiave.
- Archiviazione vettoriale: Gli embedding vengono salvati in un database vettoriale Chroma, insieme ai testi originali. Questo database ottimizzato per la ricerca di similitudine permette di recuperare rapidamente i frammenti di testo più rilevanti per una determinata query.

Durante tutto il processo, lo stato di avanzamento viene aggiornato nel database, permettendo all'interfaccia utente di mostrare un feedback in tempo reale all'utente. Al termine dell'elaborazione, la base di conoscenza viene marcata come aggiornata, ripristinando il flag needs_reload a falso.

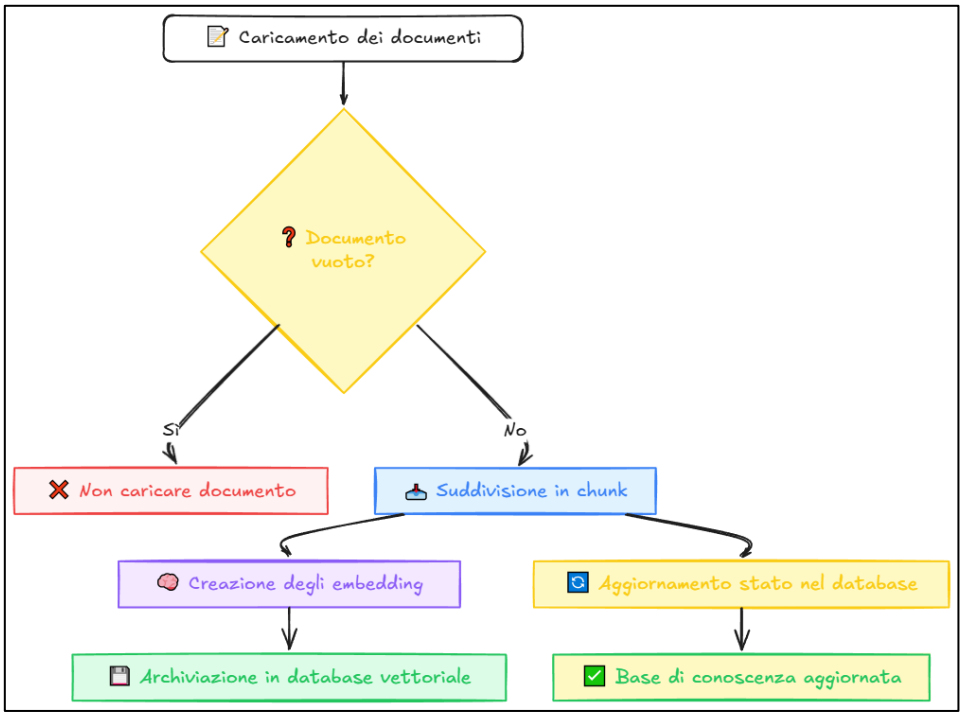



Figura 24 - Caricamento documenti

L'API REST esposta dal backend offre endpoint per la gestione completa delle basi di conoscenza:

```
@base_knowledge.route('/', methods=['POST'])
@auth.login_required
def create_base_knowledge():
    data = request.get_json()
    current_user = auth.current_user()
    current_profile = current_user.profile

    if not current_profile:
        return jsonify({'error': 'No profile found for user'}), 400

    name = data.get('name')
    if not name:
        return jsonify({'error': 'Name is required'}), 400

    # Get list of assistant IDs
    assistant_ids = data.get('assistant_ids', [])

    # Verify all assistants exist and belong to the user
    assistants = Assistant.query.filter(
        Assistant.id.in_(assistant_ids),
        Assistant.profile_id == current_profile.id
    ).all()

    if len(assistants) != len(assistant_ids):
        return jsonify({'error': 'One or more invalid assistant IDs'}), 400

    folder_name = f"base_knowledge_{current_profile.id}_{name.lower().replace(' ', '_')}"
    folder_path = str(Path('files') / folder_name)
    full_folder_path = BASE_DIR / folder_path
    full_folder_path.mkdir(exist_ok=True)
```

```
    new_base_knowledge = BaseKnowledge(
        name=name,
        description=data.get('description'),
        profile_id=current_profile.id,
        folder_path=folder_path,
        assistants=assistants
    )

    try:
        db.session.add(new_base_knowledge)
        db.session.commit()

        return jsonify({
            'message': 'Base knowledge created successfully',
            'base_knowledge': {
                'id': new_base_knowledge.id,
                'name': new_base_knowledge.name,
                'description': new_base_knowledge.description,
                'assistant_ids': [a.id for a in new_base_knowledge.assistants],
                'folder_path': new_base_knowledge.folder_path,
                'created_at': new_base_knowledge.created_at,
                'updated_at': new_base_knowledge.updated_at
            }
        }), 201
```

L'endpoint di creazione verifica l'autenticazione dell'utente, valida i dati ricevuti e crea una nuova base di conoscenza nel database. Un aspetto importante di questo processo è la validazione degli assistenti a cui la base di conoscenza sarà associata: il sistema verifica che tutti gli ID degli assistenti forniti esistano e appartengano all'utente corrente.

Viene creata una struttura di cartelle per memorizzare i file della base di conoscenza, con un nome derivato dall'ID del profilo e dal nome della base di conoscenza. Questo approccio garantisce l'isolamento dei dati tra diversi profili e previene conflitti di nome.

```
@base_knowledge.route('/<int:base_knowledge_id>/process', methods=['POST'])
@auth.login_required
def start_processing(base_knowledge_id):
    from ..tasks import process_base_knowledge

    current_user = auth.current_user()
    current_profile = current_user.profile

    if not current_profile:
        return jsonify({'error': 'No profile found for user'}), 400

    base_knowledge = BaseKnowledge.query.get_or_404(base_knowledge_id)

    if base_knowledge.profile_id != current_profile.id:
        return jsonify({'error': 'Unauthorized access'}), 403

    # Reset the needs_reload flag when starting processing
    base_knowledge.needs_reload = False
    db.session.commit()

    task = process_base_knowledge.apply_async(args=[base_knowledge_id])
    return jsonify({
        'task_id': task.id,
        'status': 'PENDING'
    })
```

Questo endpoint avvia l'elaborazione asincrona di una base di conoscenza. Dopo aver verificato che l'utente sia autorizzato ad accedere alla base di conoscenza, il sistema resetta il flag needs_reload (che verrà eventualmente ripristinato se l'elaborazione fallisce) e crea un nuovo task Celery. L'ID del task viene restituito al client, che può utilizzarlo per monitorare lo stato di avanzamento dell'elaborazione.

```
@base_knowledge.route('/<int:base_knowledge_id>/files', methods=['POST'])
@auth.login_required
def upload_file(base_knowledge_id):
    current_user = auth.current_user()
    current_profile = current_user.profile

    if not current_profile:
        return jsonify({'error': 'No profile found for user'}), 400

    base_knowledge = BaseKnowledge.query.get_or_404(base_knowledge_id)

    if base_knowledge.profile_id != current_profile.id:
        return jsonify({'error': 'Unauthorized access'}), 403

    if 'file' not in request.files:
        return jsonify({'error': 'No file part'}), 400

    file = request.files['file']

    if file.filename == '':
        return jsonify({'error': 'No selected file'}), 400

    if file:
        filename = secure_filename(file.filename)
        file_folder_path = Path(BASE_DIR) / base_knowledge.folder_path / 'files'
        file_folder_path.mkdir(exist_ok=True)

        file_path = file_folder_path / filename
        file.save(str(file_path))

        file_type = filename.rsplit('.', 1)[1].lower() if '.' in filename else 'unknown'
        file_size = os.path.getsize(str(file_path))

        new_file = BaseKnowledgeFile(
            name=filename,
            description=request.form.get('description', ''),
            base_knowledge_id=base_knowledge.id,
            file_path=str(Path('files') / filename),
            file_type=file_type,
            file_size=file_size
        )

        try:
            db.session.add(new_file)
            base_knowledge.needs_reload = True  # Set the flag
            db.session.commit()

            return jsonify({
                'message': 'File uploaded successfully',
                'file': {
                    'id': new_file.id,
                    'name': new_file.name,
                    'description': new_file.description,
```

```
                'file_path': new_file.file_path,
                'file_type': new_file.file_type,
                'file_size': new_file.file_size,
                'created_at': new_file.created_at,
                'updated_at': new_file.updated_at
            }
        }), 201
```

L'endpoint per il caricamento dei file accetta un file inviato tramite una richiesta multipart/form-data. Il file viene salvato nella cartella appropriata, e le informazioni sul file (nome, percorso, tipo, dimensione) vengono memorizzate nel database. Importante notare che questo processo imposta il flag needs_reload sulla base di conoscenza, indicando che è necessario rigenerare gli embedding per includere il nuovo contenuto.

### 4.14 Chiamate di Test

Il sistema di gestione delle chiamate è un componente fondamentale dell'applicazione che permette agli assistenti virtuali di interagire con gli utenti attraverso chiamate telefoniche. Questo modulo gestisce sia le chiamate in entrata (inbound) provenienti da utenti esterni, sia le chiamate di test che consentono agli sviluppatori e agli utenti di verificare il funzionamento degli assistenti senza effettuare vere chiamate telefoniche.

Il modello dati per le chiamate è strutturato per gestire diversi tipi di chiamate, tracciare le conversazioni e memorizzare informazioni di sistema:

```
class CallType(enum.Enum):
    inbound = 'inbound'
    outbound = 'outbound'
    test = 'test'

class Call(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    call_sid = db.Column(db.String(64), nullable=False)

    phone_number_id = db.Column(db.Integer, db.ForeignKey('phone_number.id'), nullable=False)
    phone_number = db.relationship('PhoneNumber', backref='calls')

    external_phone_number_id = db.Column(db.Integer, db.ForeignKey('external_phone_number.id'),
nullable=True)
    external_phone_number = db.relationship('ExternalPhoneNumber', backref='calls')

    call_type = db.Column(db.Enum(CallType), nullable=False)

    status = db.Column(db.String(20), nullable=False)
    direction = db.Column(db.String(15), nullable=False)
    duration = db.Column(db.Integer, nullable=True)

    recording_url = db.Column(db.String(255), nullable=True)

    started_at = db.Column(db.DateTime, default=datetime.utcnow)
    ended_at = db.Column(db.DateTime, nullable=True)
```

Per gestire le trascrizioni delle conversazioni, viene utilizzato il modello ConversationTranscript:

```
class ConversationRole(enum.Enum):
    caller = 'caller'
    assistant = 'assistant'

class ConversationTranscript(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    transcript = db.Column(db.Text, nullable=False)

    start_time = db.Column(db.Integer, nullable=True)
    end_time = db.Column(db.Integer, nullable=True)

    call_id = db.Column(db.Integer, db.ForeignKey('call.id'), nullable=False)
    call = db.relationship('Call', backref='transcripts')

    role = db.Column(db.Enum(ConversationRole), nullable=False)
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
```

Questo modello memorizza ogni segmento di conversazione, distinguendo tra messaggi dell'utente (caller) e dell'assistente. Ciascuna trascrizione include:

- Il testo della trascrizione
- I timestamp di inizio e fine relativi all'inizio della chiamata
- Un riferimento alla chiamata a cui appartiene
- Il ruolo (caller o assistant) che indica da chi proviene il messaggio
- Il timestamp di creazione del record

Per registrare messaggi di sistema e informazioni diagnostiche, è stato implementata la tabella CallSystemMessage:

```
class CallSystemMessageType(enum.Enum):
    call_info = 'call_info'
    call_error = 'call_error'
    call_warning = 'call_warning'
    call_debug = 'call_debug'

class CallSystemMessage(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    message = db.Column(db.Text, nullable=False)

    call_id = db.Column(db.Integer, db.ForeignKey('call.id'), nullable=False)
    call = db.relationship('Call', backref='system_messages')

    type = db.Column(db.Enum(CallSystemMessageType), nullable=False)

    created_at = db.Column(db.DateTime, default=datetime.utcnow)
```

Questo modello è utilizzato per registrare eventi di sistema, errori, avvisi e informazioni di debug relative a una chiamata. Categorizzare questi messaggi per tipo permette di filtrarli facilmente durante l'analisi dei log.

Infine, il sistema tiene traccia dei numeri di telefono esterni attraverso il modello ExternalPhoneNumber:

```
class ExternalPhoneNumber(db.Model):
```

```
    id = db.Column(db.Integer, primary_key=True)
    phone_number = db.Column(db.String(15), nullable=False)
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
```

Questo modello memorizza i numeri di telefono degli utenti che chiamano o che vengono chiamati dal sistema, consentendo di tracciare la storia delle interazioni con ciascun numero.

Le chiamate in entrata (provenienti da utenti esterni) vengono gestite attraverso l'integrazione con Twilio, un servizio di comunicazione cloud che espone API per la telefonia. Il flusso di gestione delle chiamate è implementato negli endpoint di routes.py.

L'endpoint principale per le chiamate in entrata è:

```
@calls.route('/incoming-call', methods=['POST'])
def handle_incoming_call():
    try:
        # Get call details from Twilio
        call_sid = request.values.get('CallSid')
        from_number = request.values.get('From')
        to_number = request.values.get('To')

        # Find the phone number in our system
        phone_number = PhoneNumber.query.filter_by(phone_number=to_number).first()
        if not phone_number:
            logging.error(f"No phone number found in system for {to_number}")
            return jsonify({'error': 'No phone number found in system'}), 404

        # Create call record in database
        call = Call(
            call_sid=call_sid,
            phone_number_id=phone_number.id,
            call_type=CallType.inbound,
            status="in-progress",
            direction="inbound",
            started_at=datetime.utcnow()
        )
        db.session.add(call)
        db.session.commit()

        # Create TwiML response
        response = VoiceResponse()

        # Start a websocket connection with call info

response.start().stream(url=f'/socket.io/?call_id={call.id}&phone_number_id={phone_number.id}')

        return str(response)

    except Exception as e:
        logging.error(f"Error handling incoming call: {str(e)}")
        return jsonify({'error': str(e)}), 500
```

Quando Twilio riceve una chiamata diretta a uno dei numeri di telefono registrati nel sistema, invia una richiesta POST a questo endpoint. Il controller estrae le informazioni essenziali dalla richiesta (come l'ID della chiamata, il numero chiamante e il numero chiamato), verifica che il numero chiamato sia effettivamente registrato nel sistema e crea un nuovo record di chiamata nel database.

La parte più interessante è la creazione di una risposta TwiML (un linguaggio XML specifico di Twilio) che istruisce Twilio a stabilire uno stream WebSocket con l'applicazione. Questo stream consente la comunicazione in tempo reale tra l'applicazione e l'utente chiamante, permettendo all'assistente virtuale di ascoltare ciò che l'utente dice e di rispondere in modo appropriato.

Un secondo endpoint importante è quello che gestisce gli aggiornamenti di stato delle chiamate:

```
@calls.route('/call-status', methods=['POST'])
def handle_call_status():
    try:
        call_sid = request.values.get('CallSid')
        call_status = request.values.get('CallStatus')

        # Update call status in database
        call = Call.query.filter_by(call_sid=call_sid).first()
        if call:
            call.status = call_status

            # If call ended, update end time and clean up websocket connection
            if call_status in ['completed', 'busy', 'failed', 'no-answer', 'canceled']:
                call.ended_at = datetime.utcnow()
                socketio.emit('call_ended', {'call_id': call.id})

            db.session.commit()

        return jsonify({'status': 'success'})

    except Exception as e:
        logging.error(f"Error handling call status: {str(e)}")
        return jsonify({'error': str(e)}), 500
```

Twilio invia aggiornamenti di stato delle chiamate a questo endpoint, che aggiorna di conseguenza il record della chiamata nel database. Quando una chiamata termina (sia per conclusione normale che per errore), il sistema aggiorna il timestamp di fine e notifica la fine della chiamata attraverso un evento Socket.IO, che permette al frontend e ai componenti in ascolto di reagire di conseguenza.

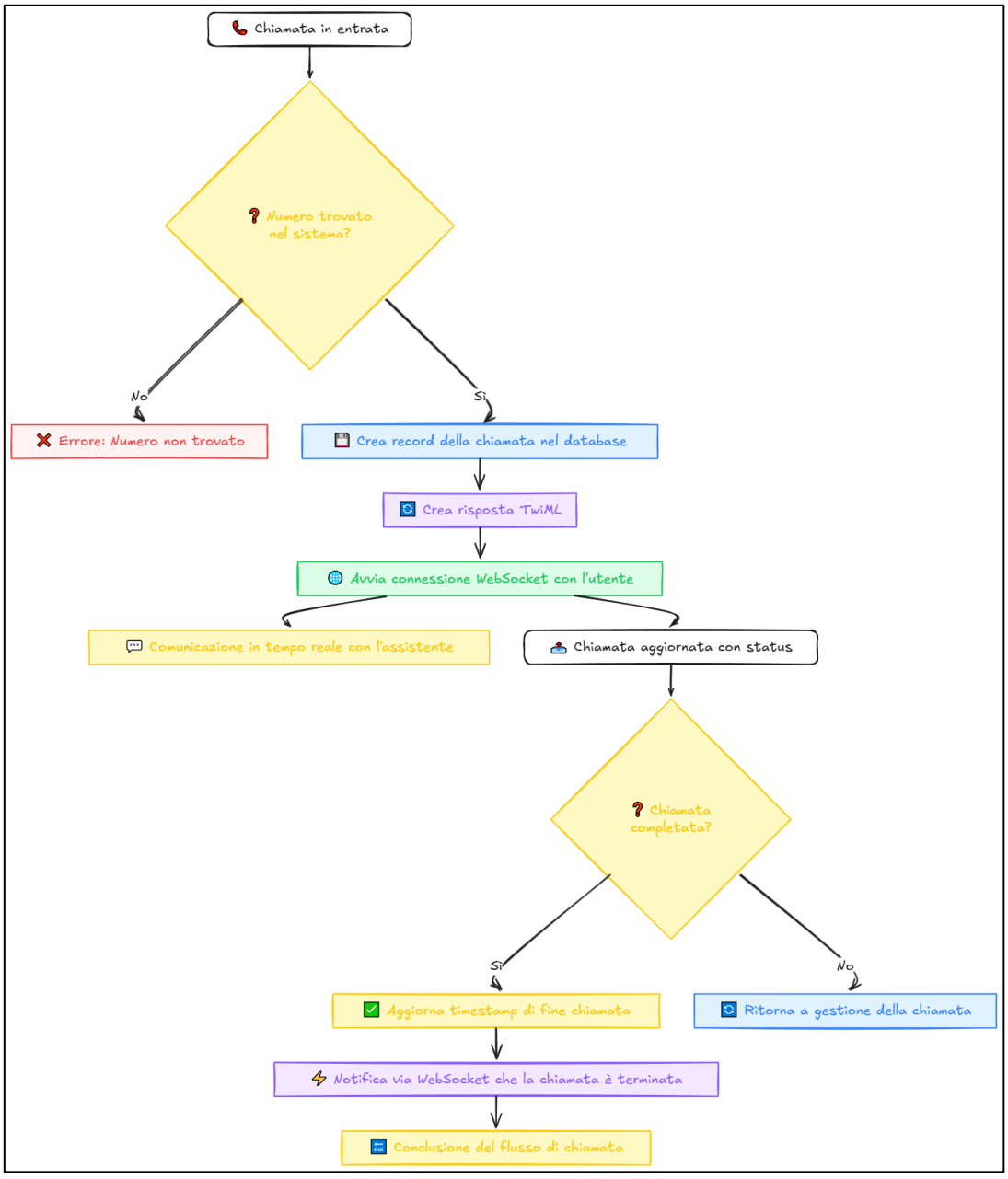



Figura 25 - Chiamate test

Le chiamate di test sono un'importante funzionalità di sviluppo e debugging che consente agli utenti di simulare una chiamata telefonica direttamente dall'interfaccia web, senza dover effettuare una vera chiamata telefonica. Questa funzionalità è implementata in test_call_routes.py.

L'endpoint principale per avviare una chiamata di test è:

```
@calls.route('test-call/start', methods=['POST'])
@auth.login_required
def start_test_call():
    try:
        data = request.get_json()
        phone_number_id = data.get('phone_number_id')

        logging.warning(f"📞 [TEST-CALL] Starting test call with phone_number_id {phone_number_id}")
```

```
        if not phone_number_id:
            logging.error(f"❌ [ERROR] Phone number ID is required")
            return jsonify({'error': 'Phone number ID is required'}), 400

        # Verify phone number exists and is available
        phone_number = PhoneNumber.query.get(phone_number_id)
        if not phone_number:
            logging.error(f"❌ [ERROR] Phone number not found with ID {phone_number_id}")
            return jsonify({'error': 'Phone number not found'}), 404

        # Create a new call record
        new_call = Call(
            call_sid=f"test-{datetime.utcnow().timestamp()}",
            phone_number_id=phone_number_id,
            call_type=CallType.test,
            status='initiated',
            direction='outbound',
            started_at=datetime.utcnow()
        )

        db.session.add(new_call)
        db.session.commit()
        logging.warning(f"📞 [TEST-CALL] Created new call record with ID {new_call.id}")

        # Get the assistant from the phone number's assistants collection
        if not phone_number.assistants or len(phone_number.assistants) == 0:
            logging.error(f"❌ [ERROR] No assistants associated with phone number
{phone_number_id}")
            return jsonify({'error': 'No assistant is associated with this phone number'}), 400

        # Use the first assistant associated with this phone number
        assistant = phone_number.assistants[0]
        logging.warning(f"🤖 [LLM] Using assistant {assistant.id} ({assistant.name if
hasattr(assistant, 'name') else 'unnamed'}) for test call")

        # Initialize call components in active_calls cache
        call_id_str = str(new_call.id)
        active_calls[call_id_str] = {
            'assistant_llm': AssistantLLM(assistant.id),
            'tts': TTS(assistant.id),
            'stt': SpeechToText(assistant.id),
            'is_speaking': False,
            'client_sid': None,
            'pending_audio': [],
            'is_greeting_audio': True,
            'call_started_time': datetime.utcnow().timestamp()
        }
        logging.warning(f"📞 [TEST-CALL] Initialized components for call {new_call.id}")

        # Store greeting as assistant message
        greeting = "Hello! How can I help you today?"
        transcript = ConversationTranscript(
            call_id=new_call.id,
            transcript=greeting,
            role=ConversationRole.assistant,
            created_at=datetime.utcnow()
        )
        db.session.add(transcript)
        db.session.commit()

        # Generate and store greeting for later delivery
        logging.warning(f"📞 [TEST-CALL] Generating greeting for call {new_call.id}")
```

```
        # Use a short delay before generating audio to give client time to connect
        def delayed_greeting_generation():
            # Check if client has connected yet
            if call_id_str in active_calls:
                call_data = active_calls[call_id_str]
                client_sid = call_data.get('client_sid')

                if client_sid:
                    logging.warning(f"📞 [TEST-CALL] Client {client_sid} already connected,
generating greeting")
                else:
                    logging.warning(f"📞 [TEST-CALL] No client connected yet for call {call_id_str},
will store greeting for later")

                # Generate greeting regardless - it will be delivered when client connects
                handle_tts(call_id_str, greeting, thread_safe=True)

        # Use a 1-second delay to give client time to connect
        eventlet.spawn_after(1.0, delayed_greeting_generation)

        logging.warning(f"📞 [TEST-CALL] Greeting preparation scheduled for call {new_call.id}")

        return jsonify({
            'status': 'success',
            'call_id': new_call.id,
            'phone_number_id': phone_number.id
        })
```

Questo endpoint richiede l'autenticazione dell'utente e accetta l'ID di un numero di telefono registrato nel sistema. Dopo aver verificato la validità del numero di telefono e la sua associazione con almeno un assistente, il sistema crea un nuovo record di chiamata di test nel database.

La parte più interessante è l'inizializzazione dei componenti necessari per la chiamata:

- Un'istanza di AssistantLLM che gestirà la generazione delle risposte dell'assistente
- Un'istanza di TTS (Text-to-Speech) per convertire il testo dell'assistente in audio
- Un'istanza di SpeechToText per convertire l'audio dell'utente in testo

Questi componenti vengono memorizzati in una cache di chiamate attive identificata dall'ID della chiamata. Il sistema genera anche un messaggio di benvenuto e lo memorizza sia nel database (come trascrizione) che in forma audio, utilizzando una funzione asincrona con un breve ritardo per dare tempo al client di connettersi.

Per terminare una chiamata di test, è disponibile l'endpoint:

```
@calls.route('test-call/end', methods=['POST'])
@auth.login_required
def end_test_call():
    try:
        data = request.get_json()
        call_id = data.get('call_id')

        if not call_id:
            return jsonify({'error': 'Missing call_id'}), 400

        # Get the call
        call = Call.query.get(call_id)
        if not call:
            return jsonify({'error': 'Call not found'}), 404
```

```
        # Update call status
        call.status = 'completed'
        call.ended_at = datetime.utcnow()
        db.session.commit()

        # Clean up active calls cache
        call_id_str = str(call_id)
        if call_id_str in active_calls:
            del active_calls[call_id_str]

        # Emit call_ended event to socket
        socketio.emit('call_ended', {'call_id': call_id})

        return jsonify({'status': 'success'})
```

Questo endpoint aggiorna lo stato della chiamata a "completed", imposta il timestamp di fine, rimuove la chiamata dalla cache delle chiamate attive e notifica la fine della chiamata tramite un evento Socket.IO.

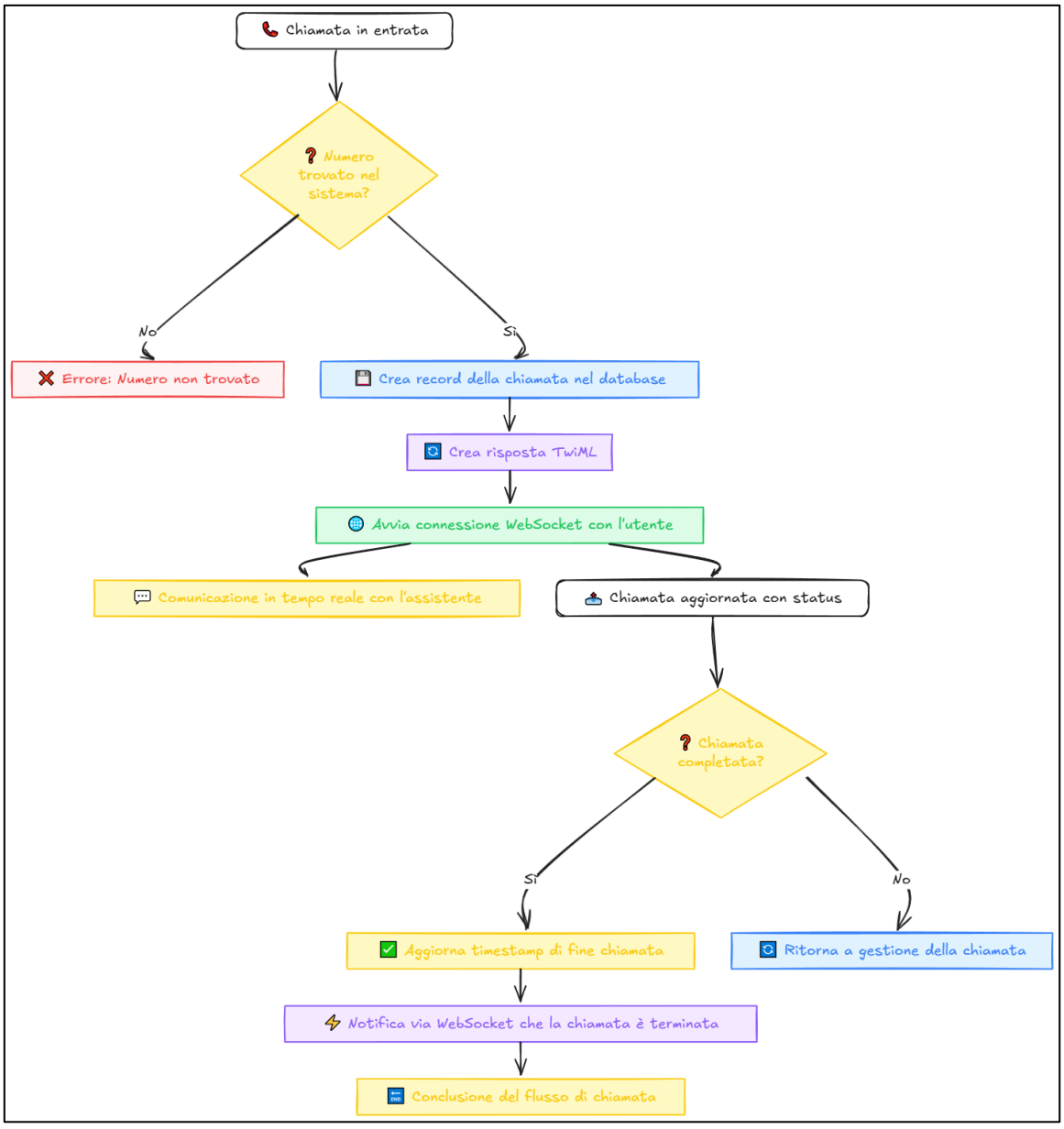



Figura 26 - Chiamata test

# 5 Test

## 5.1 Protocollo di test

| | |
|---|---|
| Riferimento | TC-001 |
| Requisito | REQ-001 |
| Nome | Registrazione |
| Descrizione | Registrazione all'applicativo |
| Prerequisiti | Database e backend funzionanti |
| Procedura | 1. Andare in https://172.20.0.14:6969/<br>2. Cliccare "create new account"<br>3. Inserire username, email e password<br>4. Cliccare "Create account" |
| Risultati attesi | Non ci sono errori nell'interfaccia e si viene reindirizzati alla dashboard dell'applicativo |

| | |
|---|---|
| Riferimento | TC-002 |
| Requisito | REQ-001 |
| Nome | Login |
| Descrizione | Login all'applicativo |
| Prerequisiti | Database e backend funzionanti |
| Procedura | 1. Andare in https://172.20.0.14:6969/<br>2. Inserire username/email e password<br>3. Cliccare "Sign in" |
| Risultati attesi | Non ci sono errori nell'interfaccia e si viene reindirizzati alla dashboard dell'applicativo |

| | |
|---|---|
| Riferimento | TC-003 |
| Requisito | REQ-002 |
| Nome | Creazione assistente |
| Descrizione | Creazione di un nuovo assistente vocale |
| Prerequisiti | Login effettuato |
| Procedura | 1. Andare in https://172.20.0.14:6969/<br>2. Cliccare "+" |
| Risultati attesi | Un nuovo assistente chiamato "New assistant" comparirà |

| | |
|---|---|
| Riferimento | TC-004 |
| Requisito | REQ-002 |
| Nome | Cambio voce assistente |
| Descrizione | Modifica della voce dell'assistente vocale |
| Prerequisiti | Login effettuato e assistente creato |
| Procedura | 1. Andare in https://172.20.0.14:6969/<br>2. Cliccare un assistente<br>3. Cambiare la voce in "Voice Configuration" |
| Risultati attesi | Il nome della voce sarà diverso e il cambiamento verrà salvato |

| | |
|---|---|
| Riferimento | TC-005 |

| | |
|---|---|
| Requisito | REQ-002 |
| Nome | Cambio prompt |
| Descrizione | Modifica del prompt dell'assistente |
| Prerequisiti | Login effettuato e assistente creato |
| Procedura | 1. Accedere all'assistente<br>2. Inserire nuovo prompt in "Prompt Configuration"<br>3. Salvare<br>4. Ricaricare pagina |
| Risultati attesi | Il prompt sarà aggiornato con il nuovo testo inserito |

| | |
|---|---|
| Riferimento | TC-006 |
| Requisito | REQ-002 |
| Nome | Configurazione LLM |
| Descrizione | Modifica delle impostazioni del modello linguistico |
| Prerequisiti | Login effettuato e assistente creato |
| Procedura | 1. Accedere a un assistente<br>2. Selezionare "Configuration" → "LLM"<br>3. Cambiare modello<br>4. Impostare temperature a 0.7 e Max Tokens a 2000<br>5. Salvare |
| Risultati attesi | Le impostazioni di LLM vengono salvate correttamente |

| | |
|---|---|
| Riferimento | TC-008 |
| Requisito | REQ-002 |
| Nome | Chat con assistente |
| Descrizione | Comunicare con l'assistente creato |
| Prerequisiti | Login effettuato e assistente creato |
| Procedura | 1. Selezionare assistente<br>2. Cliccare "Chat"<br>3. Inviare messaggio |
| Risultati attesi | L'assistente risponde al messaggio |

| | |
|---|---|
| Riferimento | TC-009 |
| Requisito | REQ-002 |
| Nome | Chat con audio |
| Descrizione | Comunicare con l'assistente con risposta audio |
| Prerequisiti | Login effettuato e assistente creato |
| Procedura | 1. Aprire chat<br>2. Attivare speaker<br>3. Inviare messaggio |
| Risultati attesi | L'assistente risponde con testo e audio |

| | |
|---|---|
| Riferimento | TC-010 |
| Requisito | REQ-002 |
| Nome | Chat con microfono |
| Descrizione | Comunicare con input vocale |
| Prerequisiti | Login effettuato e assistente creato |
| Procedura | 1. Avviare chat |

| | |
|---|---|
| | 2. Attivare speaker e microfono<br>3. Parlare |
| Risultati attesi | Appare trascrizione e risposta dell'assistente |

| | |
|---|---|
| Riferimento | TC-011 |
| Requisito | REQ-002 |
| Nome | Eliminazione assistente |
| Descrizione | Eliminare un assistente esistente |
| Prerequisiti | Login effettuato e assistente creato |
| Procedura | 1. Selezionare assistente<br>2. Cliccare "Delete Assistant"<br>3. Confermare |
| Risultati attesi | L'assistente viene eliminato dalla lista |

| | |
|---|---|
| Riferimento | TC-012 |
| Requisito | REQ-004 |
| Nome | Importazione numero |
| Descrizione | Importazione di un numero telefonico Twilio |
| Prerequisiti | Login effettuato |
| Procedura | 1. Menu "Phone numbers"<br>2. Cliccare "Import from Twilio"<br>3. Inserire SID, Token, Numero<br>4. Cliccare "Import" |
| Risultati attesi | Il numero viene importato e visibile in lista |

| | |
|---|---|
| Riferimento | TC-013 |
| Requisito | REQ-004 |
| Nome | Configurazione numero per assistente |
| Descrizione | Assegnare numero Twilio a un assistente |
| Prerequisiti | Login, assistente creato, numero importato |
| Procedura | 1. Aprire un assistente<br>2. Cliccare "Configure Number"<br>3. Selezionare numero<br>4. Salvare |
| Risultati attesi | Il numero è assegnato e visibile nella configurazione |

| | |
|---|---|
| Riferimento | TC-014 |
| Requisito | REQ-005 |
| Nome | Creazione knowledge base |
| Descrizione | Creazione di una knowledge base |
| Prerequisiti | Login e assistente creato |
| Procedura | 1. Menu “Knowledge base”<br>2. Cliccare “Create new”<br>3. Inserire nome, descrizione e selezionare assistente<br>4. Cliccare “Create” |
| Risultati attesi | La knowledge base è visibile nella lista |

| | |
|---|---|
| Riferimento | TC-015 |
| Requisito | REQ-005 |
| Nome | Aggiunta testo in knowledge base |
| Descrizione | Inserire contenuto testuale |
| Prerequisiti | Login e KB creata |
| Procedura | 1. Selezionare una KB<br>2. Scheda testo<br>3. Scrivere contenuto e titolo<br>4. Salvare |
| Risultati attesi | Documento salvato e visibile nella lista |

| | |
|---|---|
| Riferimento | TC-016 |
| Requisito | REQ-005 |
| Nome | Aggiunta URL in knowledge base |
| Descrizione | Importare contenuti da URL |
| Prerequisiti | Login e KB creata |
| Procedura | 1. Selezionare una KB<br>2. Scheda URL<br>3. Inserire URL<br>4. Cliccare “Import” |
| Risultati attesi | Contenuto visibile nella lista |

| | |
|---|---|
| Riferimento | TC-017 |
| Requisito | REQ-005 |
| Nome | Caricamento file in knowledge base |
| Descrizione | Upload di file |
| Prerequisiti | Login e KB creata |
| Procedura | 1. Selezionare una KB<br>2. Scheda file<br>3. Caricare e cliccare “Upload” |
| Risultati attesi | File appare nella lista dei documenti |

| | |
|---|---|
| Riferimento | TC-018 |
| Requisito | REQ-005 |
| Nome | Domanda all'assistente su knowledge base |
| Descrizione | Verifica uso dei contenuti |
| Prerequisiti | Login, KB e documenti configurati |
| Procedura | 1. Aprire assistente collegato a KB<br>2. Cliccare "Chat"<br>3. Fare domanda |
| Risultati attesi | L'assistente usa la KB per rispondere |

| | |
|---|---|
| Riferimento | TC-019 |
| Requisito | REQ-005 |
| Nome | Eliminazione documento da knowledge base |
| Descrizione | Eliminare un documento |
| Prerequisiti | Login e almeno un documento nella KB |
| Procedura | 1. Selezionare KB<br>2. Cliccare cestino vicino al documento<br>3. Confermare |
| Risultati attesi | Documento eliminato dalla lista |

| | |
|---|---|
| Riferimento | TC-020 |
| Requisito | REQ-005 |
| Nome | Aggiornamento knowledge base |
| Descrizione | Ricaricare la KB |
| Prerequisiti | Login e modifiche nella KB |
| Procedura | 1. Selezionare KB con indicatore "Needs reload"<br>2. Cliccare "Reload" |
| Risultati attesi | KB aggiornata e indicatore scompare |

| | |
|---|---|
| Riferimento | TC-021 |
| Requisito | REQ-005 |
| Nome | Visualizzazione file PDF |
| Descrizione | Aprire e leggere file PDF |
| Prerequisiti | Login e file PDF caricato |
| Procedura | 1. Selezionare KB<br>2. Cliccare nome del file o icona visualizzazione |
| Risultati attesi | PDF aperto in visualizzatore |

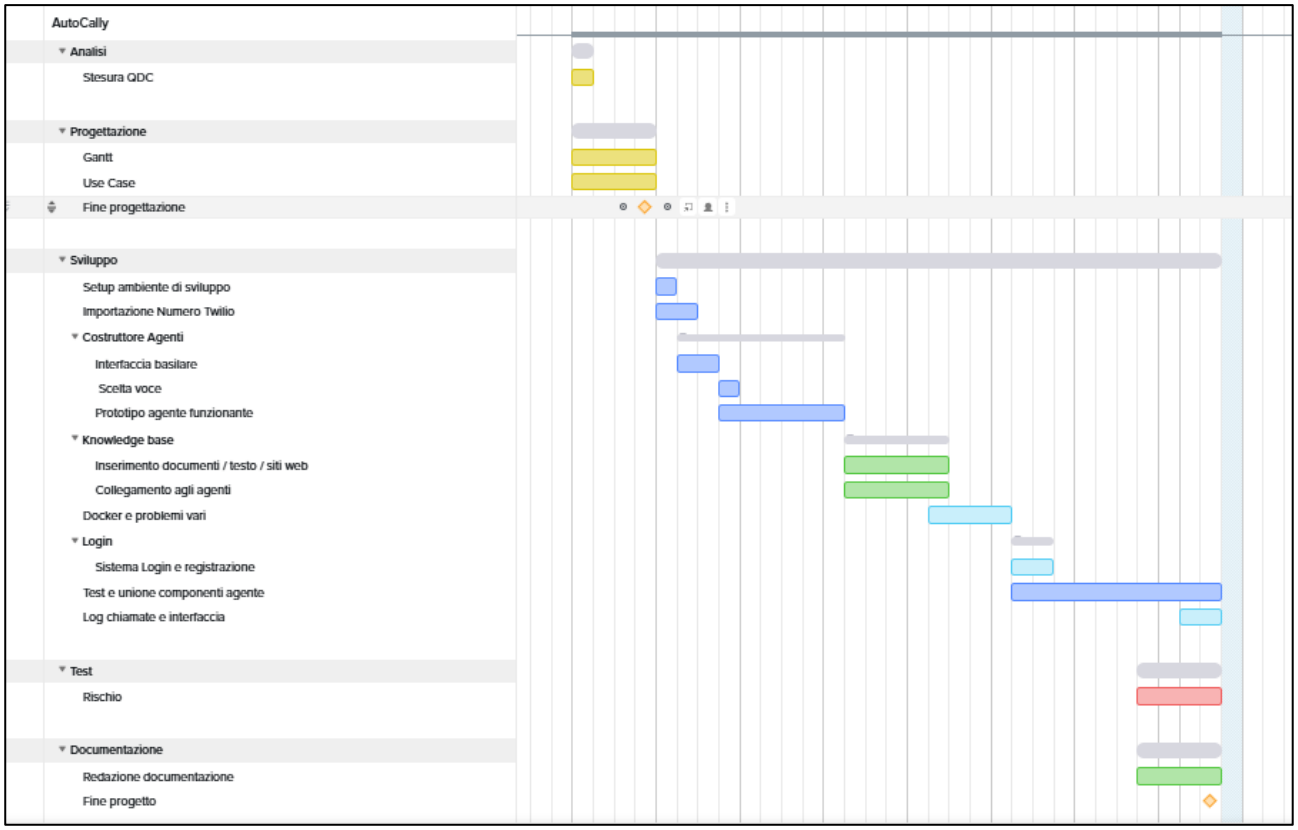
### 5.2 Risultati test

| ID Test | Requisito | Risultato |
|---|---|---|
| **TC-001** | REQ-001 | Passato |
| **TC-002** | REQ-001 | Passato |
| **TC-003** | REQ-002 | Passato |
| **TC-004** | REQ-002 | Passato |
| **TC-005** | REQ-002 | Passato |
| **TC-006** | REQ-002 | Passato |
| **TC-007** | REQ-002 | Passato |
| **TC-008** | REQ-002 | Passato |
| **TC-009** | REQ-002 | Passato |
| **TC-010** | REQ-002 | Passato |
| **TC-011** | REQ-002 | Passato |
| **TC-012** | REQ-004 | Passato |
| **TC-013** | REQ-004 | Passato |
| **TC-014** | REQ-005 | Passato |
| **TC-015** | REQ-005 | Passato |
| **TC-016** | REQ-005 | Passato |
| **TC-017** | REQ-005 | Passato |
| **TC-018** | REQ-005 | Passato |
| **TC-019** | REQ-005 | Passato |
| **TC-020** | REQ-005 | Passato |
| **TC-021** | REQ-005 | Passato |

### 5.3 Mancanze/limitazioni conosciute

Le chiamate telefoniche non funzionano, i pagamenti, login con Google non sono stati implementati per mancanza di tempo.
Mancano statistiche dei prezzi perché le chiamate telefoniche non sono state implementate completamente.
Mancano i tool, calendario, ecc, per mancanza di tempo.

# 6 Consuntivo




# 7 Conclusioni

## 7.1 Sviluppi futuri

Implementazione di tool, finire le chiamate, aggiungere la visulizzazione in tempo reale dei log delle chiamate dall'interfaccia, sistema di pagamenti, modo per l'admin di gestire le variabili di ambiente dall'interfaccia grafica, interfaccia per vedere la salute delle api.

## 7.2 Considerazioni personali

Sono molto soddisfatto del lavoro svolto su questo progetto, anche se non sono riuscito a completare tutti i requisiti previsti. Ho deciso di mettermi alla prova sperimentando con tecnologie e approcci che non avevo mai utilizzato prima, come l'uso di Docker, la gestione dell'audio e la programmazione asincrona.
Sono molto soddisfatto anche dell'architettura del progetto, anche di tutti i dettagli che non si vedono a prima vista ma che sono stati molto complessi da sviluppare per me.
Queste scelte mi hanno permesso di imparare molto, anche al di là degli obiettivi iniziali, e ritengo che il valore formativo dell'esperienza sia stato comunque molto alto.

## 8 Glossario

| Termine | Descrizione |
|---|---|
| DB | Database, un sistema organizzato per memorizzare, gestire e recuperare dati. |
| LLM | Large Language Model, un tipo di modello di intelligenza artificiale addestrato su grandi quantità di dati testuali per comprendere e generare linguaggio umano. Esempi includono GPT-3 e GPT-4. |
| Docker | Una piattaforma che permette di sviluppare, spedire e eseguire applicazioni in contenitori (containers), che sono ambienti isolati che eseguono il software in modo coerente. |
| Container Docker | Un'istanza di un'applicazione o servizio in esecuzione all'interno di un contenitore Docker, che garantisce un ambiente di esecuzione consistente e portabile tra sistemi diversi. |
| Linux Mint | Una distribuzione del sistema operativo Linux, basata su Ubuntu, che offre un'interfaccia utente facile da usare, particolarmente adatta ai principianti. |
| Frontend | La parte di un'applicazione web o software con cui l'utente interagisce direttamente. Include la progettazione e la struttura visiva del sito o dell'applicazione. |
| Nginx | Un server web e reverse proxy ad alte prestazioni, utilizzato per servire contenuti statici, fare bilanciamento del carico e gestire il traffico HTTP/HTTPS. |
| Proxy | Un server che funge da intermediario tra un client e un server. Viene usato per migliorare la sicurezza, monitorare il traffico, e bilanciare il carico. |
| Reverse proxy | Un tipo di proxy che inoltra le richieste dei client a uno o più server backend. Viene usato per gestire il traffico, proteggere i server backend, e bilanciare il carico. |
| API | Un insieme di regole che consente a software diversi di comunicare tra loro. Le API sono utilizzate per accedere a funzionalità o dati di un'applicazione esterna. |
| SSL | Un protocollo di sicurezza che crittografa la comunicazione tra un client (come un browser) e un server per garantire la privacy e l'integrità dei dati durante la trasmissione. Ora, SSL è in gran parte sostituito dal TLS (Transport Layer Security), ma il termine "SSL" è ancora comune. |

# 9 Indice delle figure



# 10 Bibliografia

## 10.1 Sitografia

- https://chatgpt.com
- https://stackoverflow.com/questions/44785585/how-can-i-delete-all-local-docker-images
- https://developers.deepgram.com/home/introduction
- https://www.youtube.com/watch?v=J2sbC8X5Pp8&ab_channel=GregKamradt
- https://www.youtube.com/watch?v=Yf9JkKmeO8U
- https://www.youtube.com/watch?v=MsP6OvXrAWA
- https://github.com/cartesia-ai/docs