# **Real-time Speech Processor**

This project provides a real-time speech processing application that transcribes audio streams and files, detects predefined keywords, and provides instant alerts. It leverages Django for the backend API, Django Channels for WebSocket communication, Celery for asynchronous task processing, and Redis as the message broker and channel layer. OpenAI's Whisper API is used for audio transcription.

## **Table of Contents**

1. [Architecture Overview](https://www.google.com/search?q=%231-architecture-overview)
2. [Setup Instructions](https://www.google.com/search?q=%232-setup-instructions)
   * [Prerequisites](https://www.google.com/search?q=%23prerequisites)
   * [Installation](https://www.google.com/search?q=%23installation)
   * [Environment Variables](https://www.google.com/search?q=%23environment-variables)
   * [Database Setup](https://www.google.com/search?q=%23database-setup)
   * [Running Components](https://www.google.com/search?q=%23running-components)
     + [Redis](https://www.google.com/search?q=%23redis)
     + [Daphne (ASGI Server)](https://www.google.com/search?q=%23daphne-asgi-server)
     + [Celery Worker](https://www.google.com/search?q=%23celery-worker)
     + [Django Development Server (for Frontend & API)](https://www.google.com/search?q=%23django-development-server-for-frontend--api)
3. [Design Decisions and Trade-offs](https://www.google.com/search?q=%233-design-decisions-and-trade-offs)
4. [System Architecture Diagram](https://www.google.com/search?q=%234-system-architecture-diagram)
5. [Database Schema](https://www.google.com/search?q=%235-database-schema)
6. [Performance Considerations](https://www.google.com/search?q=%236-performance-considerations)
7. [Future Enhancement Ideas](https://www.google.com/search?q=%237-future-enhancement-ideas)

## **1. Architecture Overview**

The application follows a microservices-like architecture, integrating several key components to handle real-time audio processing:

* **Frontend (HTML, CSS, JavaScript):** A simple web interface for starting/stopping audio recording, uploading audio files, and displaying live transcriptions and keyword alerts. It communicates with the backend via REST API for keyword management and WebSockets for audio streaming and real-time updates.
* **Django Backend (Python):**
  + **REST API:** Manages keywords and handles audio file uploads.
  + **Django Channels:** Extends Django to handle WebSockets. It acts as the bridge between the WebSocket client (frontend) and the asynchronous task queue (Celery via Redis).
* **Daphne:** An HTTP, HTTP/2 and WebSocket protocol server for Django Channels. It serves the ASGI application, handling WebSocket connections.
* **Celery:** An asynchronous task queue that offloads long-running operations (like audio transcription and keyword processing) from the main web server.
* **Redis:** Serves as both the message broker for Celery (to queue tasks) and the channel layer for Django Channels (for real-time pub/sub communication between Daphne and Celery).
* **OpenAI API (Whisper):** An external service used by Celery tasks to perform high-quality speech-to-text transcription.

The flow for real-time audio is:

Frontend (Audio Chunks) --WebSocket--> Daphne --Channels/Redis--> Celery (Transcription Task) --OpenAI API--> Celery (Process Transcription Task) --Channels/Redis--> Daphne --WebSocket--> Frontend (Transcription/Alerts)

The flow for audio file upload is:

Frontend (File) --REST API--> Django Backend --Celery/Redis--> Celery (Transcription Task) --OpenAI API--> Celery (Process Transcription Task) --Channels/Redis--> Daphne --WebSocket--> Frontend (Transcription/Alerts)

## **2. Setup Instructions**

### **Prerequisites**

* Python 3.10+
* pip (Python package installer)
* Redis Server (running locally or accessible via network)

### **Installation**

1. **Clone the repository:**  
   git clone <repository\_url>  
   cd placer\_ai\_project\_new
2. **Create and activate a Python virtual environment:**  
   python -m venv venv  
   .\venv\Scripts\activate # On Windows  
   source venv/bin/activate # On macOS/Linux
3. **Install project dependencies:**  
   pip install -r requirements.txt

### **Environment Variables**

Create a .env file in the backend directory (where manage.py is located) and populate it with the following:

DJANGO\_SECRET\_KEY=your\_django\_secret\_key\_here # Generate a strong, random key  
DJANGO\_DEBUG=True # Set to False in production  
OPENAI\_API\_KEY=YOUR\_NEW\_ALL\_PERMISSIONS\_API\_KEY\_HERE # Replace with your OpenAI API Key  
REDIS\_URL=redis://localhost:6379/0 # Adjust if Redis is on a different host/port  
OPENAI\_ORGANIZATION\_ID=org-YOUR\_ORGANIZATION\_ID\_HERE # Optional, if you have an OpenAI organization ID  
OPENAI\_PROJECT\_ID=proj-YOUR\_PROJECT\_ID\_HERE # Optional, if you have an OpenAI project ID

**Important:** Replace placeholder values with your actual keys and IDs. For DJANGO\_SECRET\_KEY, you can generate one using Python:

import os  
import secrets  
print(secrets.token\_urlsafe(50))

### **Database Setup**

Apply Django database migrations to create the necessary tables for Keywords and Transcriptions:

cd backend  
python manage.py migrate

### **Running Components**

You need to run three core components for the application to function: **Redis**, **Daphne**, and **Celery**. The Django development server is also needed to serve the frontend and handle REST API calls.

**Ensure you are in the backend directory (where manage.py is located) and your virtual environment is activated for all Python commands below.**

#### **Redis**

Start your Redis server. If you installed it locally, you can usually find it in your system's services or by running redis-server from your installation directory.

* To monitor Redis activity (optional, but useful for debugging):  
  Open a new terminal and run:  
  redis-cli MONITOR

#### **Daphne (ASGI Server)**

Daphne handles the WebSocket connections.

* Open a new terminal, activate your virtual environment, navigate to the backend directory, and run:  
  daphne backend.asgi:application -p 8000  
    
  This will start Daphne listening on 127.0.0.1:8000.

#### **Celery Worker**

Celery processes the audio transcription and keyword detection tasks asynchronously.

* Open a new terminal, activate your virtual environment, navigate to the backend directory, and run:  
  python.exe -m celery -A placer\_ai\_project worker --loglevel=info --pool=solo

#### **Django Development Server (for Frontend & API)**

This server serves your index.html and handles the REST API endpoints.

* Open a new terminal, activate your virtual environment, navigate to the backend directory, and run:  
  python manage.py runserver  
    
  This will typically start the server on http://127.0.0.1:8000/.

**Once all three components (Redis, Daphne, Celery) are running, and the Django development server is active, you can open your browser to http://localhost:8000/ to use the application.**

## **3. Design Decisions and Trade-offs**

* **Django + Django REST Framework (DRF):**
  + **Decision:** Chosen for rapid backend API development, robust ORM, and a mature ecosystem.
  + **Trade-off:** Can be overkill for very simple APIs, but provides a solid foundation for future expansion.
* **Django Channels + Daphne + Redis:**
  + **Decision:** Essential for real-time, bidirectional WebSocket communication, enabling live transcription and alerts. Redis acts as a scalable channel layer for pub/sub.
  + **Trade-off:** Adds complexity to the deployment and development environment compared to a purely HTTP-based application.
* **Celery + Redis:**
  + **Decision:** Decouples long-running tasks (like OpenAI API calls for transcription) from the main web server, preventing UI freezes and improving responsiveness. Redis is a fast and reliable message broker.
  + **Trade-off:** Introduces an additional component to manage and monitor.
* **OpenAI Whisper API:**
  + **Decision:** Provides highly accurate and robust speech-to-text transcription with minimal effort.
  + **Trade-off:** External dependency, incurs costs per usage, and requires an API key.
* **gevent Pool for Celery:**
  + **Decision:** gevent is an excellent choice for I/O-bound tasks (like making external API calls to OpenAI), allowing the Celery worker to handle many concurrent requests efficiently without using multiple heavy OS threads.
  + **Trade-off:** Requires careful monkey-patching at startup (gevent.monkey.patch\_all()) to ensure all standard library I/O operations are non-blocking. Incorrect patching can lead to subtle bugs or performance issues.
* **In-memory Audio Buffering in Consumer:**
  + **Decision:** Collects small audio chunks from the client into a larger buffer before sending to Celery, reducing the number of API calls to OpenAI and improving transcription quality (as Whisper performs better on longer segments).
  + **Trade-off:** Introduces a slight delay in transcription updates (e.g., 1 second buffer means 1-second delay).
* **Temporary File Storage for Uploads:**
  + **Decision:** Handles larger audio file uploads by saving them to disk before passing the path to Celery, avoiding large in-memory transfers between Django and Celery.
  + **Trade-off:** Requires disk I/O and cleanup logic. For very high-volume production, cloud storage (S3) would be preferred.
* **Frontend (HTML/JS/Tailwind):**
  + **Decision:** A lightweight and direct approach for the UI, allowing quick prototyping and responsiveness with Tailwind CSS.
  + **Trade-off:** Lacks the component-based structure and state management benefits of modern JavaScript frameworks (like React, Vue, Angular), which could become a limitation for more complex UIs.

## **4. System Architecture Diagram**

+-------------------+ +-------------------+ +-----------------+  
| | | | | |  
| Frontend | | Daphne | | Celery |  
| (HTML, JS, CSS) | | (ASGI Server) | | (Worker) |  
| | | | | |  
+--------+----------+ +---------+---------+ +--------+--------+  
 | REST API (Keywords, Upload) | Tasks (Transcribe, Process)  
 | WebSocket (Audio Stream, Alerts) |  
 | |  
 | |  
 | |  
+--------v----------+ +--------v----------+ +--------v--------+  
| | | | | |  
| Django Backend |<----->| Redis |<----->| OpenAI API |  
| (DRF, Channels) | | (Broker, Channel) | | (Whisper) |  
| | | | | |  
+--------+----------+ +-------------------+ +-----------------+  
 |  
 | Database (SQLite/PostgreSQL)  
 |  
+--------v----------+  
| |  
| Database |  
| (SQLite for dev) |  
| |  
+-------------------+

## **5. Database Schema**

The project uses two main models: Keyword and Transcription.

### **Keyword Model**

Stores keywords for detection and associated talking points.

| **Field** | **Type** | **Description** |
| --- | --- | --- |
| id | AutoField | Primary key. |
| word | CharField | The keyword to detect (max 100 chars, unique). |
| talking\_point | TextField | Optional suggestion for the keyword. |
| is\_active | BooleanField | Whether the keyword is active (default: True). |
| created\_at | DateTimeField | Timestamp of creation (auto\_now\_add). |
| updated\_at | DateTimeField | Timestamp of last update (auto\_now). |

### **Transcription Model**

Stores transcribed text segments for historical reference.

| **Field** | **Type** | **Description** |
| --- | --- | --- |
| id | AutoField | Primary key. |
| session\_id | CharField | Identifier for the audio session (max 255 chars, indexed). |
| speaker\_type | CharField | Type of speaker ('prospect', 'agent', 'unknown'). |
| text | TextField | The transcribed text. |
| timestamp | DateTimeField | Timestamp of transcription (auto\_now\_add). |

## **6. Performance Considerations**

* **Asynchronous Tasks:** Celery offloads CPU-intensive and I/O-bound tasks (OpenAI API calls) from the main Django process, ensuring the web server remains responsive.
* **WebSocket for Real-time:** WebSockets provide a persistent, low-latency connection for live audio streaming and immediate delivery of transcriptions and alerts, avoiding the overhead of repeated HTTP requests.
* **Audio Buffering:** The AudioConsumer buffers audio chunks for 1 second before sending them to Celery. This reduces the frequency of Celery task invocations and OpenAI API calls, which can improve transcription quality and reduce costs.
* **Redis as Broker/Channel Layer:** Redis is an in-memory data store, providing very fast message queuing for Celery and efficient pub/sub capabilities for Django Channels
* **Scalability:** The architecture is designed to be horizontally scalable. Multiple Daphne instances can be run behind a load balancer, and multiple Celery workers can be added to handle increased task load. Redis can also be clustered for high availability.

## **7. Future Enhancement Ideas**

* **User Authentication and Authorization:** Implement user accounts to manage personal keyword lists and view session histories.
* **Frontend Framework:** Migrate the frontend to a modern JavaScript framework (e.g., React, Vue, Angular) for better component management, state handling, and a more dynamic user experience.
* **Speaker Diarization:** Integrate a speaker diarization model (e.g., from OpenAI, Hugging Face) to identify and label different speakers in the conversation automatically.
* **Customizable Transcription Models:** Allow users to select different transcription models or fine-tune models for specific accents/domains.
* **Persistent Audio Storage:** Instead of temporary local files, integrate with cloud storage (e.g., AWS S3, Google Cloud Storage) for storing uploaded audio files.
* **Advanced Keyword Management:**
  + Sentiment analysis around keywords.
  + Keyword categories.
  + Thresholds for alert sensitivity.
* **Real-time Analytics Dashboard:** Visualize transcription data, keyword frequency, and speaker talk time.
* **Error Reporting:** Integrate with error monitoring tools like Sentry for better visibility into production issues.
* **Celery Beat:** Implement Celery Beat for scheduled tasks, e.g., daily reports or cleanup of old data.
* **Frontend UI/UX Improvements:** Enhance the user interface with better visual feedback for recording status, loading indicators, and alert animations.
* **Dockerization:** Containerize the application components (Django, Daphne, Celery, Redis) using Docker and Docker Compose for easier deployment and environment consistency.