Voice Transcription and Summarization Application

Contents

---

1. Abstract

This document describes the high-level design for a web application that helps users easily document and understand their spoken conversations, like meetings or personal notes. The application works by recording audio directly through the user's web browser, converting that speech into text right there on the user's computer (client-side). This text transcript is then sent to a backend server, which uses a powerful language model (like Gemini) to create a clear, well-structured summary. Finally, the application shows both the original full text and the neat summary back to the user on the webpage.

2. Introduction

Why this High-Level Design Document?

The main goal of this document is to give a clear blueprint of the application before we start building it in detail. It helps everyone involved understand how the different parts of the application will work together. Think of it as a plan that helps us spot potential problems early and serves as a guide during development.

This HLD will:

- Outline the main pieces of the application and how they connect.
- Describe what the user will see and interact with.
- Mention the software and services needed.
- Briefly touch on things like how well it should perform.
- List important design choices.

Scope

This document covers the overall structure of the system. This includes:

- How the application is broken down (frontend, backend).
- How information flows through the system (from voice to summary).
- The main technologies we plan to use.

It uses straightforward terms so that anyone involved in the project can understand the plan. It does *not* go into deep coding details.

3. General Description

Product Perspective

This application is a web-based tool designed for anyone who needs to quickly capture spoken information and get a summary of it. It aims to be simpler and more direct than manually typing notes or trying to remember everything said in a long conversation or meeting.

Problem Statement

People often participate in meetings or have conversations where important details are discussed. Remembering everything accurately or spending time manually transcribing and summarizing later is difficult and time-consuming. Users need a way to automatically get a text version of what was said and a concise summary to quickly grasp the main points. A similar tool that exists is NotePen.

Proposed Solution

Our solution is a web application with the following steps:

1. The user clicks a button on the webpage to start recording audio using their computer's microphone.
2. As the user speaks, JavaScript code running *in their browser* uses a speech-to-text library to convert the voice into text in real-time or shortly after recording stops.
3. Once the transcription is ready, the user can trigger sending this text to our backend server (built with Flask).
4. The backend server takes the text and sends it to an external LLM service (Gemini 2.5) via its API, asking for a summary.
5. The LLM processes the text and sends back a formatted summary.
6. Our backend receives the summary and sends it back to the user's browser.
7. The webpage then displays both the full, raw transcription and the nicely formatted summary for the user to read, copy, or use as needed.

Further Improvements

Looking ahead, we could add features like:

- Saving transcripts and summaries for later access.
- Allowing users to edit the transcript before summarizing.
- Supporting different languages.
- Integrating with calendar apps to automatically title meeting notes.
- Offering different summary formats (e.g., bullet points, paragraphs).

4. Technical Requirements

The application must be able to:

- Access the user's microphone (with their permission).
- Perform speech-to-text conversion within the web browser.
- Send the resulting text securely to the backend.
- Communicate with the chosen LLM (Gemini 2.5) API from the backend.
- Receive and process the summary from the LLM.
- Send the transcript and summary back to the frontend.
- Display both text outputs clearly on the webpage.
- Handle a reasonable duration of audio input (limited by browser/LLM capabilities).

5. Data Requirements

The system primarily handles two types of data:

- **Audio Data:** Captured temporarily by the browser for transcription. Not stored long-term in this version.
- **Text Data:**
    - Raw transcript generated client-side.
    - Formatted summary generated by the LLM.
    - This text data flows from client to backend, to LLM, back to backend, and finally back to the client for display. No database storage is planned for the initial version.
-

6. Tools Used

- **Frontend:** HTML, CSS, JavaScript. Potentially using libraries like Bootstrap for styling. A browser-based JavaScript Speech-to-Text library will be used for transcription.
- **Backend:** Python programming language using the Flask framework.
- **Summarization Service:** Google's Gemini 2.5 API.
- **Version Control:** Git / GitHub (assumed for code management).
- **Development Environment:** Standard text editors/IDEs (like VS Code, PyCharm).

7. Hardware Requirements

- **User:** A computer (desktop or laptop) with a modern web browser (like Chrome, Firefox, Edge) capable of running JavaScript and accessing the microphone. A working microphone (built-in or external). An internet connection is needed for the summarization step.

- **Server:** A standard web server environment capable of running the Python/Flask application and making outbound internet calls to the LLM API.

8. Constraints

- **Transcription Accuracy:** Depends heavily on the quality of the user's microphone, background noise, clarity of speech, and the capabilities of the chosen client-side JavaScript library.
- **Summarization Quality:** Depends on the effectiveness of the Gemini LLM for the given text and the quality of the input transcript.
- **Client-Side Performance:** Running transcription in the browser might use significant CPU/memory on the user's computer, potentially affecting performance on older machines or during very long recordings.
- **LLM Limitations:** The Gemini API might have usage limits (requests per minute, token limits per request) or potential costs in the future, even if currently free.
- **Internet Dependency:** Summarization requires an active internet connection to reach the backend and the LLM API. The transcription part works offline within the browser.

9. Assumptions

- Users will grant microphone permission to the web application.
- Users have browsers that support the necessary Web APIs (Web Audio API, potentially Web Speech API or the chosen library's requirements).
- The chosen client-side transcription library provides acceptable accuracy for typical use cases.
- The Gemini 2.5 API will be available, accessible from our backend, and perform reliably.
- Users understand that the quality of the output depends on the quality of the input audio.

10. Design Details

Process Flow

1. **User Action:** User navigates to the application webpage and clicks a "Start Recording" button.
2. **Audio Capture:** The browser accesses the microphone (after permission) and starts capturing audio.
3. **Client-Side Transcription:** A JavaScript library processes the audio stream (either live or after recording stops) and generates a text transcript. This happens entirely within the user's browser.

4. **Send to Backend:** User clicks a "Summarize" button. The JavaScript code sends the full transcript text to the Flask backend API endpoint.
5. **LLM Interaction:** The Flask backend makes an API call to the Gemini 2.5 service, sending the transcript and requesting a summary.
6. **Receive Summary:** The backend waits for the Gemini API to respond with the summarized text.
7. **Send to Frontend:** The Flask backend sends the received summary (and perhaps the original transcript again for consistency) back to the user's browser.
8. **Display Results:** JavaScript on the frontend receives the data and updates the webpage to show both the raw transcription and the formatted summary clearly.

Deployment Process

- The **Backend** (Flask application) will be deployed onto a cloud platform Google Cloud App Engine.
- The **Frontend** (HTML, CSS, JS files) can be served directly by the Flask application or hosted separately on a static file hosting service (like Netlify, Vercel, AWS S3/CloudFront).

11. Performance

- **Transcription:** Should ideally feel responsive, with text appearing shortly after speech. Performance depends on the JS library and user's hardware.
- **Summarization:** Latency will depend mainly on the response time of the Gemini API and the length of the transcript. The backend should handle requests efficiently.
- **UI Responsiveness:** The frontend interface should remain interactive and not freeze during recording or while waiting for the summary.

12. Reusability

- The backend Flask API, if designed well, could potentially be reused by other clients (e.g., a mobile app version) in the future.
- Frontend JavaScript functions for recording and interacting with the backend could be structured into reusable modules.

13. Application Compatibility

- **Frontend:** Aim for compatibility with recent versions of major web browsers (Chrome, Firefox, Safari, Edge) that support the required Web APIs.

- **Backend:** Standard Python/Flask application, compatible with common Linux/Windows server environments.

14. Resource Utilization

- **Client-Side:** Transcription will consume CPU and memory on the user's machine. This is a key consideration – we are offloading this work from the server.
- **Backend Server:** Will consume CPU, memory, and network bandwidth primarily when communicating with the LLM API and handling user requests. Usage is expected to be relatively low per user, scaling with the number of concurrent users requesting summaries.

15. Deployment

The application will be deployed to a suitable web hosting environment. This involves setting up the backend server process and making the frontend files accessible via a URL. Continuous integration/continuous deployment (CI/CD) practices might be used later for easier updates.

16. KPIs (Key Performance Indicators)

We can measure the success and health of the application using indicators like:

- **Successful Task Completion Rate:** Percentage of users who start recording and successfully receive both a transcript and a summary.
- **Summarization Latency:** Average time taken from clicking "Summarize" to displaying the summary.
- **Application Uptime:** Percentage of time the backend service is available.
- **Error Rates:** Frequency of errors logged on the backend (e.g., LLM API failures) and frontend (e.g., transcription failures).
- **User Feedback (if collected):** Subjective measure of satisfaction with transcript accuracy and summary quality.

17. Conclusion

This document outlines the high-level design for the Voice Transcription and Summarization Application. It proposes a system using client-side transcription for efficiency and privacy, coupled with a powerful backend LLM (Gemini 2.5) for high-quality summarization. The chosen technology stack (Flask, JavaScript) provides a good balance for building this application. This HLD serves as a foundational plan for the development process.