Low-Level Design: Voice Transcription and Summarization Application

1. Introduction

- **1.1 Purpose:** This Low-Level Design (LLD) document provides detailed implementation specifics for the Voice Transcription and Summarization Application, building upon the concepts outlined in the High-Level Design (HLD). It is intended for developers, outlining the structure of the frontend and backend components, their interactions, and key code segments involved in the core functionality.
- **1.2 Scope:** This LLD covers the detailed design of the client-side JavaScript for voice capture and transcription, the Flask backend structure for handling requests and interacting with the Gemini API, and the specific implementation details of the summarization module. It includes essential code snippets to illustrate the core logic.

2. Detailed Design & Component Breakdown

The application consists of three main parts: Frontend (HTML/CSS/JS), Backend (Flask), and the Summarization Service interaction layer.

- **2.1 Frontend (Client-Side)**
    - **UI Elements (index.html):**
        - A main <form> (id="response-form") to manage user input and submission.
        - A <textarea> (id="transcript") to display the real-time and final transcript.
        - A <textarea> (id="summary", readonly) to hold the raw summary text returned from the backend.
        - A (id="summary-output") to display the formatted (Markdown rendered) summary.
        - A <button> (id="toggle-btn") to start/stop voice recording.
        - A <button> (id="submit-btn", type="submit") to send the transcript to the backend for summarization.
    -
    - **Core Logic (JavaScript in index.html):**
        - **Speech Recognition:** Utilizes the browser's Web Speech API (window.SpeechRecognition or webkitSpeechRecognition).
            - Initializes the recognizer with settings like continuous=true and interimResults=true.
            - The #toggle-btn click handler starts (recognizer.start()) or stops (recognizer.stop()) the recognition process and updates button text/styling.
            - The onresult event handler processes speech results: iterates through results, appends final transcripts to finalTranscript, captures interim results, and updates the #transcript textarea value dynamically.

```
// Key logic for handling speech recognition results
    recognizer.onresult = function(event) {
        let interimTranscript = "";
        for (let i = event.resultIndex; i < event.results.length; i++) {
            let transcript = event.results[i][0].transcript;
            if (event.results[i].isFinal) {
                finalTranscript += transcript;
            } else {
                interimTranscript += transcript;
            }
        }
        textarea.value = finalTranscript + interimTranscript;
    };
```

- **Data Submission:** The #submit-btn triggers a standard HTML form POST request. The content of the #transcript textarea is sent as form data with the key transcript to the backend's / endpoint.
- **Summary Display:** After the form submission and page reload (as handled by Flask's render_template), the returned summary variable populates the #summary textarea.

Markdown Rendering: A separate script uses marked.js library on DOMContentLoaded to take the content of the (hidden or read-only) #summary textarea, parse it as Markdown, and display the formatted HTML result in the #summary-output div.

```
// Key logic for rendering Markdown summary
    document.addEventListener('DOMContentLoaded', function() {
        const summaryTextarea = document.getElementById('summary');
        const summaryOutputDiv = document.getElementById('summary-output');

        if (summaryTextarea && summaryTextarea.value.trim() !== '') {
            const markdownContent = summaryTextarea.value;
            marked.setOptions({
              breaks: true
            });
            const renderedHtml = marked.parse(markdownContent);
            summaryOutputDiv.innerHTML = renderedHtml;
        }
    });
```

- **2.2 Backend (Server-Side - Flask)**
    - **Application Setup (app.py):**
        - A simple Flask application instance is created.
        - It imports the MyAssistant class for summarization logic.
    -
    - **API Endpoint (app.py):**
        - A single route / is defined, accepting both GET and POST methods.
        - GET requests simply render the index.html template.
        - POST requests handle the form submission:
            - Extracts the transcript text from the form data (request.form['transcript']).
            - Instantiates the MyAssistant.
            - Calls the assistant.generate(transcript) method to get the summary.
            - Re-renders the index.html template, passing the obtained summary (and potentially the original transcript) back to the frontend for display.
            -

```
# Key logic for Flask route handling
@app.route('/', methods=['GET', 'POST'])
def index():
    Assistant = MyAssistant()
    transcript = ''
    summary = ''
    if request.method == 'POST':
        transcript = request.form['transcript']
        print(transcript)
        summary = Assistant.generate(transcript)
    # Pass transcript back too if needed for redisplay in textarea
    # return render_template('index.html', transcript=transcript, summary=summary)
    return render_template('index.html', summary=summary)
```

- **2.3 Summarization Service (MyAssistant class in my_assistant.py)**
    - **Initialization & Dependencies:** Uses google-genai library and python-dotenv to manage the API key.
    - **Core Method (generate):**
        - Takes the raw transcript string as input.
        - Initializes the genai.Client using the API key loaded from environment variables (os.getenv("Gemini_api_key")).

- Specifies the Gemini model to use (gemini-2.0-flash).
- Constructs the contents payload, placing the transcript within the 'user' role part.
- Defines the GenerateContentConfig, crucially setting response_mime_type="text/plain" and providing the detailed system instruction prompt to guide the LLM on the desired summary format (headings, bullet points, style, length).
- Makes the API call using client.models.generate_content.
- Returns the response.text containing the generated summary.

```
# Key logic within MyAssistant.generate
    class MyAssistant:
      def generate(self, transcript: str) -> str:
        client = genai.Client(
          api_key=os.getenv("Gemini_api_key"),
        )

        model = "gemini-2.0-flash"
        contents = [
          # ... transcript formatted as user content ...
           types.Content(
            role="user",
            parts=[types.Part.from_text(text=transcript)],
          ),
        ]
        generate_content_config = types.GenerateContentConfig(
          response_mime_type="text/plain",
          system_instruction=[
            # ... detailed system prompt ...
             types.Part.from_text(text="""You are a professional summarization
    assistant...""")
          ],
        )

        response = client.models.generate_content(
          model=model,
          contents=contents,
          config=generate_content_config,
        )
        print(response.text) # Optional: server-side logging
        return response.text
```

3. Data Flow Sequence

1. **User Action:** User clicks "Record Voice" (#toggle-btn).
2. **Audio Capture & Transcription (Client):** Browser captures audio; JavaScript (recognizer.onresult) processes audio chunks via Web Speech API, updating the #transcript textarea.
3. **User Action:** User clicks "Stop Recording" (#toggle-btn, second click) and then "Summarize" (#submit-btn).
4. **Request (Client -> Server):** Browser sends a POST request to / with the #transcript content in the form data.
5. **Backend Processing (Server):** Flask route (app.py) receives the transcript. It instantiates MyAssistant and calls generate().
6. **LLM Interaction (Server -> Gemini):** MyAssistant sends the transcript and prompt to the Gemini API via the google-genai library.
7. **LLM Response (Gemini -> Server):** Gemini API processes the request and returns the generated summary text.
8. **Backend Response (Server -> Client):** MyAssistant returns the summary to the Flask route. Flask renders index.html, embedding the summary text within the template.
9. **Display (Client):** Browser receives the rendered HTML. The summary populates the #summary textarea. The DOMContentLoaded script runs, reads the summary, uses marked.js to convert it to HTML, and displays it in #summary-output.

4. Deployment Considerations

- **Backend:** The Flask application (app.py and my_assistant.py with dependencies) will be deployed to a platform like Google Cloud App Engine as described in the HLD. Configuration will include setting the Gemini_api_key environment variable.
- **Frontend:** The index.html, style.css, and any client-side JavaScript libraries (like marked.min.js if not using CDN) will be served. Flask's static folder mechanism is suitable for this, simplifying deployment as a single unit.

5. Conclusion

This LLD provides the specific implementation details for the Voice Transcription and Summarization application. It clarifies the roles of the frontend JavaScript (handling speech recognition and display), the Flask backend (request routing and orchestration), and the MyAssistant class (interfacing with the Gemini API for summarization). The provided code snippets highlight the essential logic within each component, guiding the development process based on the HLD.