Speech Based Code Editor Code Review Summary:

We focused on the three modules of our system for the code review: the speech-to-text, the text-to-code, and the front-end. Each module plays a critical role in ensuring the speech-based code editor operates well. Our goal for the code review was to assess the functionality, performance, and modularity of each component, identify potential bottlenecks, and work on the integration of the front end to the back end.

The speech-to-text module is responsible for accurately transcribing user input, and its effectiveness directly impacts the subsequent steps in the workflow. The text-to-code module interprets transcribed text, using a call to an LLM, which requires precision and adherence to coding standards. Finally, the front-end module provides the user interface and experience that helps connect all three modules, ensuring usability and accessibility. By examining these modules, we aimed to refine our implementation, address any areas of technical debt, and align the overall design with the system requirements. For each team that worked on a specific module, we had these members swap with another module to review their code, so we can focus on the integration of all three modules from a different perspective. This was useful because we are currently at the stage of integrating all modules. Looking at the other parts of the code helped us understand what issues might arise during this integration and what dependencies to rectify.

Front-end Review

1. **startOAuth**
    a. **Description**:
    This function initiates the GitHub OAuth process by generating an authorization URL containing the client ID and redirect URI. It then opens the URL in the user's default browser, allowing them to log in to GitHub and authorize access to the application.
    b. **Evaluation of Usefulness**:
    The function works as intended, successfully opening the GitHub authorization page. It enables seamless integration with GitHub's OAuth flow, ensuring that users can log in securely.
    c. **Issue 1: Hardcoded client ID**
    Explanation: The client ID is hardcoded in the function, exposing sensitive information in the source code. This could lead to misuse if the code is leaked or accessed by unauthorized individuals.
    How to fix: Move the client ID to an environment variable or configuration file and retrieve it dynamically in the code.

d. **Issue 2: Lack of error handling**
   Explanation: If the `vscode.env.openExternal` function fails to open the browser (e.g., due to network issues or system restrictions), the function provides no feedback or recovery options, potentially leaving users unable to proceed.
   How to fix: Add a try-catch block to handle errors and display appropriate error messages to the user if the browser fails to open.

2. **showLoginPanel**
   a. **Description**:
      This function creates and displays a login page using a VS Code webview. It includes an interactive interface for GitHub login and listens for user commands (e.g., login, sign-out). It dynamically updates the content of the webview based on the user's actions and login state.
   b. **Evaluation of Usefulness**:
      The function effectively creates a login interface and handles user interactions as expected. It successfully facilitates GitHub login and displays the appropriate content.
   c. **Issue 1: No user feedback during authentication**
      Explanation: While the user is authenticating, the panel does not provide feedback (e.g., "Logging in..."). This may confuse users if the process takes time or encounters issues.
      How to fix: Add a loading message or spinner to the webview during the authentication process.
   d. **Issue 2: Static HTML content**
      Explanation: The webview's HTML content is static and lacks flexibility for updates or localization. Any changes to the UI require modifying the HTML string in the function.
      How to fix: Use a templating engine or external HTML file to dynamically generate the content, allowing for easier updates and better maintainability.
3. **startRecording**
   a. **Description**:
      This function initializes a microphone input stream, processes audio data in real-time, and sends the audio data to the webview as a Base64-encoded string. It is essential for capturing user speech for transcription.
   b. **Evaluation of Usefulness**:
      The function works well for initiating audio recording and transferring audio data to the webview. It supports real-time speech-to-text functionality effectively.
   c. **Issue 1: No error handling for microphone access**
      Explanation: If the user denies microphone access or the system lacks a microphone, the function fails silently, leaving users unsure of the issue.
      How to fix: Add error handling to detect and notify users if microphone access is denied or unavailable.
   d. **Issue 2: Inefficient audio data processing**
      Explanation: The function processes and sends audio data in small chunks, which could increase overhead and reduce performance, especially for long

recordings.
How to fix: Implement buffering to process larger chunks of audio data before sending them to the webview, reducing the frequency of data transfers.

4. **ConnectWebSocket**
    a. **Description**:
    This function connects to a web socket on your local computer.
    b. **Evaluation of Usefulness**:
    The function works well for connecting to a specific socket (4001) and has sufficient error handling.
    c. **Issue 1: Should work for any socket**
    Explanation: This method should be more general and take in a specific port number and then connect to a socket on that port instead of being hard coded to 4001.
    How to fix: Add a parameter that represents the port number and fix the logic to connect to that port.
    d. **Issue 2: Failing to Reconnect**
    Explanation: If there is a failure to connect on a specific web socket (ie the port is being used by some other process) the function should just run a new server on a new port and connect to that. This is crucial because if the web socket does not work, everything breaks.
    e. **How to fix**: In the case that I cannot connect to the web socket, I should create a new server on a different port that is unused and try to connect to that one.

Speech to Text Review

Speech to text module:
(Note: comments here are mainly made on the file https://github.com/evdalal/speech-code-editor/blob/6a5463c4dc4bc5cac276767bdac8d5eb3c81b66d/SpeechToText/transcriber/audioTranscriber.py. This file may be subject to refactoring later.)

1. **transcribe():**
    a. **Description:**
    The function is used to transcribe captured audio. It automatically starts recording when voice activity is detected if not manually started recording. It automatically stops recording when no voice activity is detected if not manually stopped. It processes the recorded audio to generate a transcription.
    b. **Evaluation of Usefulness**
    Yes, the function is useful for hands-free transcription in scenarios where users cannot or prefer not to manually control recording. However, its effectiveness is contingent on proper voice activity detection (VAD) and accurate audio processing.
    c. **Issue 1's title: Voice Activity Detection Fails in Noisy Environments**

- **Explanation:**
  In environments with high background noise, the function may incorrectly detect voice activity due to noise or fail to detect a low-volume speaker. This results in unintended recordings or missed speech, reducing the system's reliability.
- **How to fix:**
  1. Integrate advanced VAD algorithms that distinguish between human voice and background noise, such as WebRTC VAD or machine learning models.
  2. Allow users to adjust sensitivity settings for voice detection based on their environment.

d. **Issue 2's title: Transcription Accuracy Decreases for Accents or Poor Audio Quality**
   - **Explanation:**
     The function relies on transcription models that may not be optimized for diverse accents or poor-quality audio inputs, leading to inaccurate or incomplete transcriptions. This undermines the function's utility for users with non-standard accents or suboptimal recording conditions.
   - **How to fix:**
     Introduce noise reduction and speech enhancement preprocessing steps before sending the audio to the transcription engine.

2. process_audio_chunk():

a. **Description:**
   The function handles incoming audio chunks, stores them in a buffer. And then sends the data to an audio queue when the buffer is full. It supports handling both raw bytes and NumPy arrays, converting stereo audio to mono, resampling to a 16kHz sample rate, and ensuring data is in the correct int16 format before buffering.

b. **Evaluation of Usefulness**
   This function provides the input data format matches the expectations and works as expectation.

c. **Issue 1: Thread safety for shared resources**
   - **Explanation:**
     The function accesses self.buffer and self.audio_queue, which may be shared across threads. If this function is called concurrently in a multi-threaded context, race conditions could occur, leading to corrupted buffer data or lost audio chunks.
   - **How to fix:**
     Provide clear text suggestions for fixing the bug or include the corrected code snippet, if applicable.

d. **Issue 2's title:**
   - **Explanation:** Briefly explain the issue, why it occurs, and the potential impact it has on the system.

- **How to fix:** Use thread-safe mechanisms like locks from threading module to ensure exclusive access to shared resources:

```
# Add a lock as a class attribute
self.lock = Lock()

# Use the lock when modifying the buffer
with self.lock:
    self.buffer += chunk
    while len(self.buffer) >= buffer_limit:
        data_to_process = self.buffer[:buffer_limit]
        self.buffer = self.buffer[buffer_limit:]
        self.audio_queue.put(data_to_process)
```

3. **_recording_worker (within class AudioTranscriber)**
   a. **Description:**
   The `_recording_worker` method in the `AudioTranscriber` class is a core component responsible for monitoring audio input and controlling recording states based on voice activity.
   b. **Strengths:**
      - **Comprehensive Functionality**:
        The method handles various tasks, including monitoring audio input, managing state transitions, handling buffer overflow, and initiating transcription. This makes it the backbone of the transcription workflow.
      - **Voice Activity Integration**:
        The combination of voice activity detection (via Silero/WebRTC) and automatic start/stop mechanisms enhances the usability of the transcription system.
      - **Early Transcription Feature**:
        Allowing early transcription during silent periods provides flexibility and improves the user experience, especially in scenarios with intermittent speech.
      - **Robust Error Handling**:
        The use of try-except blocks ensures that the function can gracefully handle unexpected errors, preventing complete crashes in the system.
   c. **Issues:**
      - **High Coupling**
        The function could be refactored so that some functionalities are handled in smaller, well-defined helper methods. For example, we may consider adding functions: `_manage_recording_state()`,responsible for managing recording start/stop logic;
        `_detect_and_handle_silence()`,responsible for handling silence-related logic.
      - **Limited Scalability for High Data Rates**
        Handling large audio queues with manual buffer overflow checks (`self.audio_queue.qsize() > self.max_allowed_latency`)

can lead to delays and data loss under high data rates. We may want to implement circular buffers or integrate a dedicated audio streaming library that supports high-throughput scenarios.

- **Manual management of timing variables**
  Variables like `last_processing_check_time`, `last_buffer_message_time`, and `silence_start_time_after_speech` are managed manually, increasing the risk of errors in timing logic. We may want to use a central timer utility or library (e.g., `datetime.timedelta` or a state machine) to manage time-related logic more reliably and clearly.

Text to Code Review

1. **get_conversation_messages**
   a. **Description:**
   This method is part of the text-to-code service. Its purpose is to query the Firebase database for the conversation ID and user ID provided in the user's request, retrieving the user's past conversation history. The retrieved history is then sent to the Llama model, allowing it to understand the user's previous interactions each time. Since LLMs are stateless, this approach enables the model to simulate a form of memory. It enhances its contextual understanding.

   b. Evaluation of Usefulness
   In order to review whether the function can work as expected, a unit test called test_get_conversation_messages was implemented. The test validates the HTTP response status code and prints the query results from Firebase realtime database to the console. The results indicate that the query works as expected, returning the correct data with an HTTP status code of 200. The function's basic functionality is confirmed to be working.

   c. Issue 1: Inefficient query approach
   - **Explanation:** The current approach retrieves all user history records from the database and then filters the data in the Python code. This increases query time, adds unnecessary network data transfer, and enlarges the code's cache space. All these drawbacks make current method an inefficient querying method.
   - **How to fix:** Improved the design of the tree-structured key-value data in Firebase Realtime Database. Under the `conversation_history` node, each unique conversation ID is now directly stored as a child node. The new query method in Python constructs a direct path to the specified conversation ID, referencing the `history` node under it. This approach not

only optimizes the query's time complexity but also reduced the memory used for each query within python.

d. **Issue 2: Unnecessary query with user id**
   - **Explanation:** Using only the conversation ID is sufficient to retrieve the corresponding conversation history data. Adding an additional filter based on the user ID is unnecessary.
   - **How to fix:** Directly query user history data only with conversation id.

1. update_data_to_firebase:

a. **Description:**
Whenever a user sends a new request command from the frontend, a new entry should be added to the corresponding conversation ID in the Firebase Realtime Database to record the user's current command and the associated code file.

b. **Evaluation of Usefulness**
A unit test is designed to verify whether this function correctly inserts new conversation records into the database, including the user instructions and the user code file in the json format:
{
“1”: “Line 1 code”,
“2”: “Line 2 code”
}
The test results, as shown in the Firebase database, confirm that line number such as '1' is stored as a child node in the database.

c. **Get wrong code file format when querying user history:**
   - **Explanation:**
     Storing the user's code file in the database with line numbers as keys and the corresponding code as values can cause issues when querying data. When retrieving a user's history, Firebase Realtime Database treats incrementing numerical keys as an array due to its internal behavior. As a result, the query returns a Python list instead of the expected JSON key-value pair data.
   - **How to fix:**
     Modify the original approach by first converting the JSON file into a string and then storing this string as a value in the database. When querying, simply use Python's JSON library to convert the string back into the desired JSON format. Storing data as a string ensures that the format of the user's code files remains consistent.