תמונה שמכילה גופן, גרפיקה, טקסט, עיצוב גרפי

התיאור נוצר באופן אוטומטי

Software Engineering Department  
Braude College

Capstone Project Phase B – 61999

**Global Communication System**

**24-2-R-21**

**Advisors:**

**Dr. Dan Lemberg,** [**lembergdan@braude.ac.il**](mailto:lembergdan@braude.ac.il)

**Mrs. Elena Kramer,** [**elenak@braude.ac.il**](mailto:elenak@braude.ac.il)

**Students:**

**Razi Mograbi,** [**razi.mograbi@e.braude.ac.il**](mailto:lembergdan@braude.ac.il)

**Dolev Seri,** [**Dolev.Seri@e.braude.ac.il**](mailto:lembergdan@braude.ac.il)

**GitHub Repository:** [**https://github.com/razimograbi/RaziDolev\_Shlav\_B.git**](https://github.com/razimograbi/RaziDolev_Shlav_B.git)

**Table Of contents**

Contents

[Abstract 4](#_Toc189513993)

[1. Introduction 4](#_Toc189513994)

[2. Architecture Overview 5](#_Toc189513995)

[2.1 Architecture Diagram 5](#_Toc189513996)

[2.2 Package Diagram 7](#_Toc189513997)

[2.2.1 Client Package (Android) 7](#_Toc189513998)

[2.2.2 Server Package (FastAPI) 7](#_Toc189513999)

[2.2.3 Firebase Package 8](#_Toc189514000)

[2.2.4 AI Services Package 8](#_Toc189514001)

[2.2.5 Relationships 8](#_Toc189514002)

[2.3 Activity Diagram 9](#_Toc189514003)

[3. Development Process 9](#_Toc189514004)

[3.1 Android Mobile App development 9](#_Toc189514005)

[3.2 VoIP server development 10](#_Toc189514006)

[3.2.1 WebSocket Architecture 10](#_Toc189514007)

[3.2.2 Redis Integration 10](#_Toc189514008)

[3.2.3 Speech and Translation Pipeline 10](#_Toc189514009)

[3.2.4 Concurrency and Scalability 11](#_Toc189514010)

[3.3 Firebase integration 11](#_Toc189514011)

[3.4 Testing and Improvements 11](#_Toc189514012)

[3.4.1 Unit Testing 11](#_Toc189514013)

[3.4.2 Integration Testing 11](#_Toc189514014)

[3.4.3 Performance Measurements 12](#_Toc189514015)

[3.5 Finalization and Deployment 13](#_Toc189514016)

[4. Tools Used During Development 13](#_Toc189514017)

[5. Challenges and Solutions 14](#_Toc189514018)

[6. Results and Conclusions 16](#_Toc189514019)

[7. Lesson Learned 16](#_Toc189514020)

[8. Did We Meet the Project Goals? 17](#_Toc189514021)

[8.1 Lack of iOS Support: 17](#_Toc189514022)

[8.2 Latency Issues: 17](#_Toc189514023)

[8.3 Fine-Tuning xTTS: 18](#_Toc189514024)

[9. User Guild 18](#_Toc189514025)

[10. Maintenance guide 23](#_Toc189514026)

[10.1 Server: 23](#_Toc189514027)

[10.2 Android App: 26](#_Toc189514028)

# Abstract

As the world becomes more connected, people increasingly need real-time communication across different languages, whether for business negotiations, global teamwork, or simple everyday interactions. However, traditional solutions—such as text-based translators or human interpreters can introduce delays and disrupt the natural flow of conversation. This often leads to misunderstandings and reduced efficiency, especially during live, two-way exchanges. Our project addresses this challenge by using advanced AI to build an Android-based voice calling system for two-person calls. The system captures, transcribes, and translates spoken language in near real time, storing conversation context to improve translation accuracy. Where supported, voice cloning preserves each speaker’s original voice, for unsupported languages, we fallback to OpenAI TTS for robotic audio output. By enabling near real-time multilingual communication on everyday devices, this work demonstrates how next-generation AI can reduce language barriers in both casual and specialized settings. Ultimately, our approach offers a practical glimpse into a future where language differences no longer stand in the way of meaningful global interaction.

# Introduction

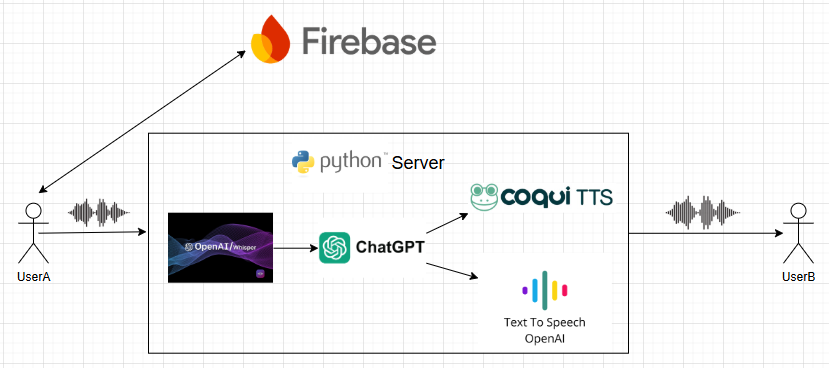
Today, there are nearly 8 billion people in the world, which has greatly increased the demand for trade and communication among diverse populations. Also, with the rise of the Internet, our global society is more interconnected than ever, making the ability to communicate across different languages increasingly vital. From sealing major international business deals to simplifying diplomatic talks between countries or simply catching up with loved ones overseas, effective cross-language communication is crucial.

However, current translation methods often fail to meet the demands of our fast-paced world. They tend to be slow, prone to inaccuracy, and struggle with capturing the emotional nuances of conversations. As a result, context is frequently lost, and subtle meanings in a speaker’s message can be overlooked. In many cases, businesses resort to hiring multiple translators to communicate with foreign partners, leading to greater costs and inefficiencies. Consequently, there is a growing need for solutions that can provide a more natural and accurate experience for individuals conversing in different languages—one that preserves the essence of what is being said.

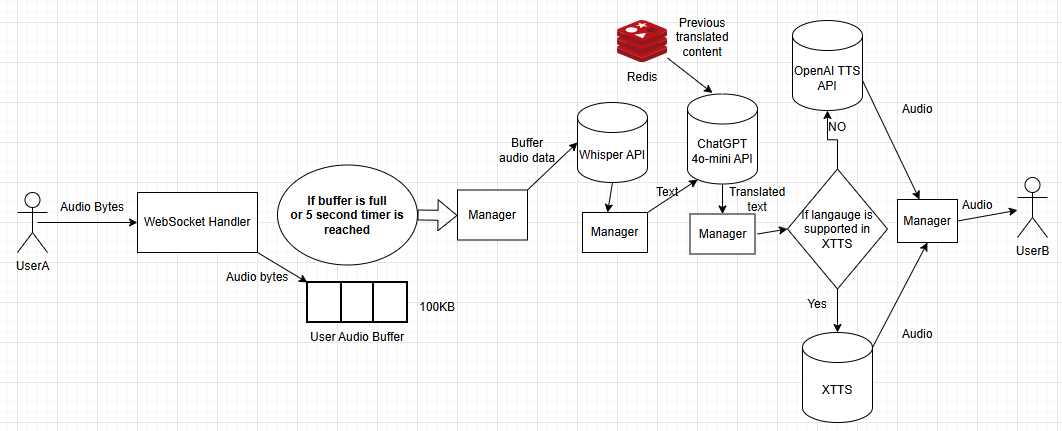
To address these challenges, our project applies advanced AI technologies to develop a communication platform that allows people to speak in their native languages, with each participant hearing the other’s voice translated into their own tongue. In addition to translating speech in near real time, the system preserves unique vocal characteristics to make the conversation feel more natural. This new approach has the potential to revolutionize how people worldwide interact, breaking down language barriers and fostering more seamless cooperation and understanding.

# Architecture Overview

## 2.1 Architecture Diagram

****

The system enables near real-time, multilingual voice communication for diverse users, including consumers, professionals, and military personnel. It consists of four key components: an Android client (Java), a Python server (FastAPI/Starlette), a Redis cache, and AI services (OpenAI, XTTS). The Android client captures and sends audio to the server via WebSocket, which manages connections, processes data, and stores metadata in Redis for reliable delivery. Audio chunks are buffered and sent to Whisper for speech-to-text, then to ChatGPT 4o-mini for translation. Translated text is converted to speech using XTTS (with voice cloning) or OpenAI TTS, depending on language support.

****

**Data Flow Summary**

In practical terms, both users establish a WebSocket connection with the server. When one user initiates a call, the other user is notified and can either accept/reject the call, once the call begins (call accepted), audio is continuously captured on the Android client. Each audio chunk is wrapped with metadata (such as a sequence number and length) and sent to the server in real time. After processing and translation through Whisper, ChatGPT 4o-mini, and the appropriate TTS service, the transformed audio—along with updated metadata—is sent back through the server to the receiving Android client. The client then uses this metadata to correctly configure playback and maintain a coherent conversation flow.

**Key Features and Constraints**

Real-Time Focus: The system is designed for near real-time performance, making latency a critical factor.

Two-Participant Limit: Currently supports one-on-one calls rather than group or conference calls.

FastAPI for Concurrency: FastAPI and Starlette were chosen to handle concurrent WebSocket connections.

Context Preservation: Redis storage ensures audio chunks are processed in order, helping the AI maintain context for more accurate translation and natural-sounding speech.

Overall, this architecture aims to deliver a smooth, intuitive experience for users requiring quick, reliable translation and voice communication, all under the hood of a Python server and specialized AI-driven services.

## Package Diagram

תמונה שמכילה טקסט, מספר, צילום מסך, קו

התיאור נוצר באופן אוטומטי

### 2.2.1 Client Package (Android)

**Purpose:** Manages the user interactions on Android devices, including UI, audio capture, WebSocket communication, and Firebase operations.

**Modules:**

* **Authentication & Profile:** Handles sign-up/login via SignUpUI and LoginUI and records a short audio clip using the RecordingModule.
* **Communication:** Uses WebSocketClient for socket connections and MetadataManager to structure metadata (sequence number, sample rate, etc.) for audio chunks.
* **Audio Playback:** Features AudioPlayer for configuring playback settings (sample rate, bit depth) and playing processed audio.
* **Firebase Interaction:** Includes FirebaseAuthManager for authentication and FirebaseUserSaver for storing user profiles (embeddings, language preferences).

### 2.2.2 Server Package (FastAPI)

**Purpose:** Acts as the core processing layer, managing WebSocket connections, sign-up requests, and AI-driven audio processing.

**Modules:**

* WebSocketManager: Facilitates real-time communication with clients, routing audio chunks and metadata.
* **Sign-Up Module:** SignUpHandler receives user details and audio recordings, forwarding them to EmbeddingService for generating speaker embeddings.
* **Audio Processing Module:** AudioChunkHandler validates and sequences incoming PCM data.

AudioPipeline orchestrates speech-to-text (Whisper), translation (ChatGPT), and text-to-speech (XTTS/OpenAI TTS).

* **Call Management Module:** CallTracker monitors active calls, and ParticipantManager manages participant states.

### 2.2.3 Firebase Package

**Purpose:** Handles user data storage and authentication.

**Components:**

* **User Database:** Stores profiles (names, emails, password hashes, languages, embeddings).
* **Authentication Module:** Manages sign-up, login, token generation, and validation.

**Usage:** The Android client interacts directly with Firebase for authentication and profile storage. The server may optionally update Firebase with new embeddings.

### 2.2.4 AI Services Package

**Purpose:** Provides external/internal AI capabilities for audio and text processing.

**Services:**

* **WhisperService:** Converts buffered audio chunks to text.
* **TranslationService:** Translates text using ChatGPT or similar NLP APIs.
* **TextToSpeechService:** Converts translated text to audio using XTTS (voice cloning) or OpenAI TTS.
* **EmbeddingService:** Generates speaker embeddings from short audio samples during sign-up.

### 2.2.5 Relationships

The Client Package → Server Package relationship covers both sign-up (transmitting user info and short WAV audio for embeddings) and real-time voice calls (sending audio chunks and receiving processed audio). Simultaneously, the Client Package → Firebase Package link reflects how the Android client leverages Firebase for user authentication, and profile storage, including storing speaker embeddings. Meanwhile, the Server Package → AI Services Package connection underlines the FastAPI server’s reliance on AI endpoints for speech-to-text, translation, text-to-speech, and embedding generation. Optionally, the Server Package → Firebase Package relationship arises if the server itself stores or updates user profiles (including newly generated embeddings) rather than delegating all data writes to the client.

## Activity Diagram

**תמונה שמכילה צילום מסך, תרשים, קו, טקסט

התיאור נוצר באופן אוטומטי**

# Development Process

To develop GlobalCommunication, we began by identifying our core requirements and determining how to translate them into code. Our first step was setting up a VoIP server capable of transmitting audio between two users, ensuring a functional communication backbone. Alongside this, we designed a basic Android UI, allowing us to test the interaction between the app and the VoIP system. This foundational setup enabled us to validate the core functionality before expanding to more advanced features.

Given our small team of two and limited time, we developed the mobile app and backend in parallel, ensuring steady progress while integrating both components efficiently.

The core development process included:

* 1. Android Mobile App development

We built the main activity and Call Dashboard Activity screens that interact with the WebSocket to stream audio to the server. Additionally, we implemented login, sign-up, and audio recording screens, ensuring each feature was developed step by step and thoroughly tested before moving to the next. While the UI remains simple and functional, it was designed with usability in mind, ensuring a smooth experience for users interacting with the app’s core features.

* 1. VoIP server development

The VoIP server was developed in parallel with the mobile app. We selected Python FastAPI for its asynchronous capabilities and native WebSocket support, which allows for low-latency, bidirectional audio transmission between clients. Below are the key design elements:

### 3.2.1 WebSocket Architecture

* Upon connection, each client establishes a persistent WebSocket connection with the server.
* The server spawns an asynchronous task for each incoming stream to handle audio data in real time. This event-driven approach prevents blocking and maximizes responsiveness.

### 3.2.2 Redis Integration

* **Metadata Management**: Each arriving audio chunk is preceded by a JSON-formatted metadata message (including sequence number, sampling rate, and length). This metadata is stored in a Redis hash for quick lookups and data validation.
* **Audio Chunk Ordering**: The server appends raw audio data to an ordered list (or queue) keyed by the call/session ID in Redis. This ensures chunks remain correctly sequenced, even if network fluctuations cause them to arrive out of order.
* **Buffering**: Each user session maintains a 100KB in-memory buffer. When the buffer is full or five seconds have elapsed—whichever occurs first—the server flushes the accumulated audio to the speech processing pipeline.

### 3.2.3 Speech and Translation Pipeline

* **Whisper (Speech-to-Text)**: Once flushed, the concatenated audio chunk is sent to Whisper for transcription.
* **ChatGPT 4o-mini (Translation)**: The transcribed text is passed to ChatGPT 4o-mini for translation into the target language.
* **TTS (Text-to-Speech)**:
  + If the target language is supported by **XTTS** (6.6GB parameters), the text is converted back to speech with voice cloning.
  + Otherwise, **OpenAI TTS** is used for languages not supported by XTTS.
* The resulting audio is returned to the server, which sends it back to the recipient over the WebSocket with new metadata followed by the raw audio data.

### 3.2.4 Concurrency and Scalability

* FastAPI’s asynchronous model allows for multiple simultaneous calls without significant blocking.
* Redis transactions avoid race conditions by locking critical sections when updating shared structures (e.g., call status, chunk ordering).
  1. Firebase integration

At the same time, we integrated Firebase into our project, leveraging its built-in Android support to manage authentication and data storage without the need for a separate backend server. This allowed us to seamlessly store and retrieve user-related data across different parts of the app, ensuring smooth integration with other features.

* 1. Testing and Improvements

Throughout development, the VoIP server underwent rigorous testing to validate its stability, performance, and functional correctness. We combined **unit tests**, **integration tests**, and **functional tests** to ensure each component—WebSocket handling, Redis data management, audio chunk ordering, and AI integration—performed as expected.

### 3.4.1 Unit Testing

* We wrote unit tests for smaller server functions, such as validating sequence numbers, parsing metadata, and handling Redis read/write operations.
* The function that converts MP3 files into raw PCM 16 kHz data received particular attention, ensuring consistent audio encoding quality.

### 3.4.2 Integration Testing

* Full end-to-end tests confirmed the seamless functionality of the pipeline. Audio captured on an Android device was sent to the server, where it was buffered and processed using Whisper for transcription, ChatGPT 4o-mini for translation, and the appropriate TTS service for speech synthesis before being routed back to the recipient.
* These tests were repeated under various network conditions to verify that chunk ordering and buffer logic remained robust against packet loss or reordering.

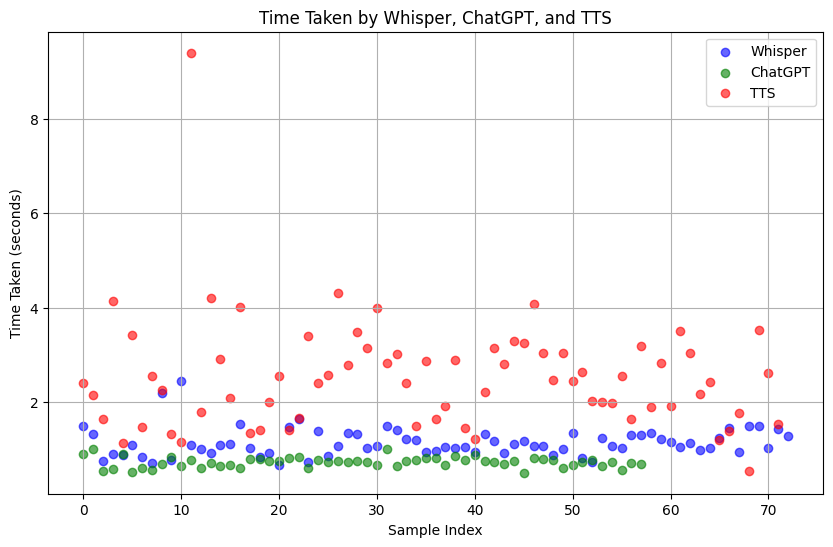
### 3.4.3 Performance Measurements

* We evaluated overall latency by measuring the time from when a user spoke to when the transcribed/translated audio played on the recipient’s device.
* The **XTTS** model, at 6.6 GB parameters, was tested on both CPU and an NVIDIA GTX 1080 GPU, yielding the following results for a one-minute conversation Between a person speaking Arabic and the other speaking English (average sentence length ~20 characters):

|  |  |  |  |
| --- | --- | --- | --- |
| Component | Without XTTS (s) | XTTS (CPU)(s) | XTTS (GPU)(s) |
| Whisper | 1.16 | 1.14 | 1.15 |
| ChatGPT4o-mini | 0.73 | 0.7 | 0.78 |
| TTS/XTTS | 2.58 (OpenAI TTS) | 7.5 (XTTS CPU) | 3.5 (XTTS GPU) |
| Notes | - | CPU processing was significantly slower due to the large model size. | - |

The performance comparison table highlights the average processing times for Whisper, ChatGPT, and TTS across different configurations. Without XTTS, OpenAI TTS achieves **a total** processing time of 4.47 seconds, whereas XTTS on a CPU significantly increases latency to 9.34 seconds due to its large model size. Using an NVIDIA GTX 1080 GPU reduces XTTS processing time to 5.43 seconds.

Also here is the sample points from the same conversation but without using the XTTS only the open AI TTS :



* 1. Finalization and Deployment

As development neared completion, we focused on final optimizations and ensuring stability across all features. We refined the app’s performance, tested key interactions, and addressed any remaining issues.

# Tools Used During Development

* **Android Studio** – The primary IDE used for developing and testing the Android app, providing an integrated environment for coding, debugging, and UI design.
* **Google Firebase** – A serverless backend solution that we used for authentication and data storage, allowing us to manage user data without maintaining our own backend server.
* **GitHub** – A version control system used to track changes, collaborate efficiently, and maintain different versions of the project throughout development.
* **FastAPI** – A high-performance, asynchronous web framework used to build the VoIP server, enabling efficient WebSocket communication and integration with AI services.
* **Redis** – An in-memory data structure store used for managing metadata, sequencing audio chunks, and caching temporary data to ensure low-latency processing.
* **Coqui XTTS** – A multilingual zero-shot text-to-speech model with **6.6 billion parameters**, enabling high-fidelity speech synthesis while preserving the speaker’s voice characteristics. Due to its size, testing was conducted on both **CPU and NVIDIA GTX 1080 GPU**, revealing significant performance differences in processing time.
* **Whisper (OpenAI)** – An automatic speech recognition (ASR) model used for transcribing audio into text with high accuracy across multiple languages.
* **ChatGPT 4o-mini** – A lightweight AI model used for translating transcribed text into the recipient’s language before speech synthesis.
* **OpenAI TTS** – A text-to-speech model used as an alternative when XTTS does not support the recipient’s language, offering clear and natural speech output without voice cloning.

# Challenges and Solutions

* **Learning Curve in Android Development**: As first-time Android developers, we struggled with understanding the project structure, managing Gradle dependencies, and navigating Android Studio. Debugging was also difficult, as we relied only on Logcat without knowing how to filter logs effectively.
* **Solution:** We improved by experimenting, reading documentation, and troubleshooting Gradle issues. Additionally, using ChatGPT helped us quickly understand concepts, debug errors, and find solutions more efficiently, making the learning process much smoother.
* **Managing Server Load Efficiently:** To avoid putting excessive load on the VoIP server, we needed to find an alternative solution for handling user management and authentication.
* **Solution**: Initially, we planned to build a separate Node.js server for user management while using Firebase as a database. However, we later discovered that Firebase could be integrated directly into our Android app, allowing us to handle both user authentication and data storage without maintaining an additional server. This approach simplified development and reduced server load.
* **Formatting the User Audio Sample:** When users sign up, they provide a 7-8 second audio sample for xTTS. However, xTTS only accepts WAV format, while Android records audio in a different format, requiring conversion.
* **Solution:** To resolve this, we created a custom recording class that captures the user's audio and builds a WAV file, including the necessary header and formatting. This allowed us to seamlessly generate compatible audio samples without relying on external conversion tools.
* **WebSocket Stability, Packet Fragmentation, and Packet Loss:** Network instability caused packet loss and occasional fragmentation of audio data, disrupting the conversation flow.
* **Solution**: Using the knowledge that we acquired from the Computer Networks course (61765), We introduced a sequence-based retransmission and acknowledgment mechanism to detect missing or fragmented packets. When a gap was identified, the server requests a resend from the sender.
* **Limited Language Coverage in XTTS:** The XTTS model does not support all languages (e.g., Hebrew), restricting voice cloning capabilities for certain user pairs.
* **Solution**: We developed a dynamic TTS selection mechanism that automatically defaults to OpenAI TTS whenever XTTS lacks support for the target language. This fallback ensures that users can still communicate effectively, even if voice cloning is unavailable.
* **Handling Large Audio Buffers Efficiently:** Continuous audio streaming generated large buffers, increasing both memory usage and the potential for processing delays.
* **Solution**: We implemented a rolling buffer system combined with an automatic flush mechanism and a Redis TTL (Time to Live). Once a buffer reached a specified size or time threshold, it was promptly flushed to the speech-processing pipeline, freeing up resources and preventing excessive memory consumption. This approach balanced real-time performance with system stability.

**Deployment:** Deploying the server on Google Cloud Platform (GCP) was one of the toughest parts of the project. With zero prior experience in GCP, server hosting, or ML model deployment, we faced numerous issues—configuring the environment, fixing dependencies, and repeatedly testing different setups. The biggest challenge was getting the XTTS model to run efficiently on limited hardware. It took about a week of trial and error, but in the end, we gained valuable experience in cloud deployment.

# Results and Conclusions

Overall, we successfully achieved most of our primary project goals—developing an Android app for multilingual speech-to-speech translation with voice cloning for supported languages. While the delay was significant, this was anticipated in Phase A. To address this, we tested GPU acceleration on parts of our application, demonstrating that the delay could be reduced. However, for cost reasons, our final deployment used a CPU-only virtual machine, resulting in significantly higher latency, as detailed in the development process chapter.

**decision making process:**

Our approach involved researching and evaluating various options to determine the best fit for our project. In some cases, cost was the deciding factor due to our tight budget. In others, efficiency and implementation time were the primary considerations, as we worked within a limited timeframe and needed the app to perform efficiently, given its low-latency requirements.

# Lesson Learned

As a team, we believe our development process was generally efficient. However, there were a few areas where improvements could have been made:

* 1. While our overall communication was effective, particularly when working on the backend and frontend separately, better coordination regarding the frontend’s requirements and the data sent by the server could have saved us time during development. Improved alignment in this area would have helped streamline the integration process.
  2. We made design decisions for the app (screens, user flow, etc.) on the fly instead of dedicating time upfront to plan its layout and functionalities from both the server and user experience perspectives. As a result, some screens had to be modified after they were already implemented, leading to unnecessary rework and time loss. A more structured design phase could have prevented this.
  3. Initially, we planned for the VoIP server to handle user management, but we later realized this could negatively impact communication performance. Midway through development, we decided to create a separate user management server, which resulted in additional time spent adjusting our architecture. Fortunately, Firebase's direct integration with Android made the transition somewhat easier, but better upfront planning would have improved work distribution and overall time management.

# Did We Meet the Project Goals?

Our app, Global Communication, successfully met most of our primary goals by implementing a speech-to-speech translation system with voice cloning. However, we fell short in a few areas:

* 1. Lack of iOS Support:

Initially, we planned to use Flutter, which supports both Android and iOS, but we later decided on Java. While most processing happens on the server, Java, as a native language for Android, provides better efficiency in handling UI updates and real-time audio streaming, reducing potential overhead. Additionally, Android has a larger market share, making it a more strategic choice over iOS. Moreover, we had more experience with Java, and given our time constraints, it was the more practical option for faster development.

* 1. Latency Issues:

While we anticipated some delay, our initial goal was to keep latency under 5 seconds, which could have been achieved with GPU acceleration. However, due to cost constraints, we had to deploy the server on a CPU-only virtual machine, which significantly increased the delay. Although our tests showed that GPU processing could improve performance, the high cost made it impractical for deployment**.**

* 1. Fine-Tuning xTTS:

Our initial plan was to fine-tune xTTS for each user to achieve better voice output. However, this proved to be a significant challenge, as it required dynamically creating datasets on the fly. Collecting high-quality voice samples in an uncontrolled environment made it difficult to ensure reliable training data. Timewise, we couldn't fit this into our development schedule, and given its complexity, it could be considered a separate project on its own.

# User Guild

**Welcome to Global communication**

**System Requirements:**

* Android Version: requires Android 8.0 (API Level 26) or higher
* Storage: At least 100 MB of free space required
* Headphones with Microphone: Required for calls to work properly
* Internet Connection: Required for the app services

**Installation**

* Go to the project repository on GitHub: <https://github.com/razimograbi/RaziDolev_Shlav_B.git>
* Download the APK file.
* Install the APK on your phone:
  + Open the downloaded file.
  + If prompted, allow installation from Unknown Sources in your phone settings.
  + Follow the on-screen instructions to complete the installation.
  + Start the app

**Login Screen**

**תמונה שמכילה טקסט, צילום מסך, גופן, עיצוב

התיאור נוצר באופן אוטומטי**

תמונה שמכילה טקסט, צילום מסך, גופן, עיצוב

התיאור נוצר באופן אוטומטי

This is the login screen where you can access your account or sign up for a new one.

When opening the app, you may see a "Headphones Required" message.  
This is because the app requires headphones with a microphone for clear communication. Press ok to continue.

* **Enter Your Email** – Type in the email address you registered with.
* **Enter Your Password** – Input your password.
* **LOG IN** – Click "LOG IN" to proceed to the Dashboard Screen.
* **SIGN UP** – Click "SIGN UP" to create a new account and proceed to the Sign-Up Screen.

**Signup screen**

**תמונה שמכילה טקסט, צילום מסך, תצוגה, תוכנה

התיאור נוצר באופן אוטומטי**תמונה שמכילה טקסט, צילום מסך, תוכנה, מערכת הפעלה

התיאור נוצר באופן אוטומטי

This is the Signup screen where you create new account.

* **Enter Your Email** – Provide a valid email address.
* **Enter Your Profile Name** – Choose a display name for your account.
* **Set Your Password** – Create a password.
* **Confirm Password** – Re-enter the password to ensure it matches.
* **Select Your Preferred Language** – Choose from the dropdown list.
* **CONTINUE** – Click "CONTINUE" Proceed to the Audio Recording Screen to record your voice for voice cloning.

**Audio Recording Screen**

תמונה שמכילה טקסט, צילום מסך, גופן, דף אינטרנט

התיאור נוצר באופן אוטומטיתמונה שמכילה טקסט, צילום מסך, עיצוב

התיאור נוצר באופן אוטומטי

This is the Audio Recording Screen, where you create a voice recording for voice cloning purposes. Make sure to provide a high-quality recording, as it will affect the accuracy of your voice cloning.

* **Microphone Permission** – Select "While using the app" or "Only this time" when prompted to allow microphone access. This is required for recording your voice.
* **Read the Given Passage** –Speak clearly and naturally as you read the text on the screen. The text is in English, but you can record in any language as long as the audio quality is good.
* **START RECORDING** – Click "START RECORDING" to begin capturing your voice. The recording will automatically stop after 8 seconds.
* **PLAY RECORDING** – Click "PLAY RECORDING" to listen to your recorded audio.
  + After listening to your recording, click "START RECORDING" again if the quality is not good enough.
* **SUBMIT** – Click "SUBMIT" to confirm and proceed with voice cloning. Once submitted, your signup is complete, and you will be redirected to the Login Screen.

**Dashboard Screen**תמונה שמכילה טקסט, צילום מסך, תוכנה, דף אינטרנט

התיאור נוצר באופן אוטומטי

**תמונה שמכילה טקסט, צילום מסך, תוכנה, דף אינטרנט

התיאור נוצר באופן אוטומטיתמונה שמכילה טקסט, צילום מסך, תוכנה, דף אינטרנט

התיאור נוצר באופן אוטומטי**

This is the Dashboard Screen, where you can manage your friends, see who is online and choose whom you want to call.

* **REFRESH** – Click "REFRESH" to update the list of online friends.
* **Enter Friend’s Email** – Type a friend’s email and click "ADD FRIEND" to add them.
* **Friend List** – If friends are online, they will appear in the list.
* **Click a friend’s name** to select who you want to call.
* **Long-click a friend’s name** to remove them from your friend list.
* **PROCEED** – Click "PROCEED" to go to the Call Center Screen.
  + If no friend is selected, you will enter the call screen and wait for an incoming call.
* **LOGOUT** – Click "LOGOUT" to return to the login screen.

**Call Center Screen**

**תמונה שמכילה טקסט, צילום מסך, גופן, עיצוב

התיאור נוצר באופן אוטומטיתמונה שמכילה טקסט, צילום מסך, גופן, עיצוב

התיאור נוצר באופן אוטומטיתמונה שמכילה טקסט, צילום מסך, כרטיס ביקור, עיצוב

התיאור נוצר באופן אוטומטי**

The Call Center Screen is where you can make and receive calls. If no friend was selected in the Dashboard Screen, you will wait for an incoming call. When a call is received, you can accept or reject it.

* **CALL** – Click "CALL" to initiate a call with a selected friend.
* **HANG UP** – Click "HANG UP" to end an ongoing call.
* **Incoming Call Notification** – If you receive a call, you will see a prompt with the caller’s email.
  + Click **"ACCEPT"** to answer the call.
  + Click **"REJECT"** to decline the call.
* **BACK** – Click "BACK" to return to the Dashboard Screen.

# 10. Maintenance guide

## 10.1 Server:

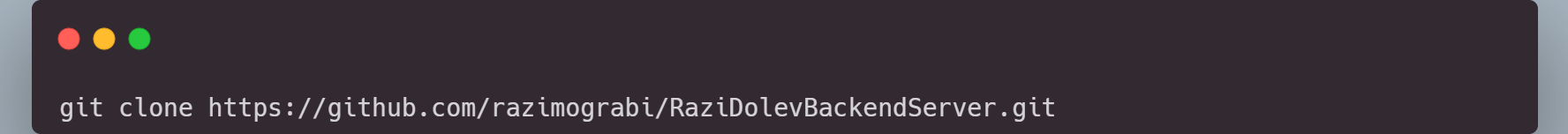
**Operating Environment**

1. **Operating System**
   * **Recommended**: Linux for server deployments.
   * While Windows and macOS may work, Linux provides better performance and stability for FastAPI, Redis, and GPU drivers.
2. **Python**
   * **Version**: Python 3.10+ is required for compatibility with FastAPI, AI libraries, and other dependencies.
3. **GPU for XTTS**
   * Because XTTS is required in production, a **NVIDIA GTX 1080 (or better)** is recommended.
4. **Dependencies**
   * **Redis**: Used for caching, metadata, and audio chunk ordering.
   * **Firebase**: Used for authentication and some data persistence.
   * No additional SQL databases or data stores (e.g., MySQL, PostgreSQL) are used.
5. **Installation Instructions**

The following steps outline the **recommended** process for installing and configuring the project’s **dedicated software** (the server code).

**Local Environment (Non-Docker)**

To begin, ensure Python 3.10+ and pip are installed on your system, along with FFmpeg for audio processing. Next, clone the project’s GitHub repository:





You’ll also need Redis and Redis-server—install it and start the service:

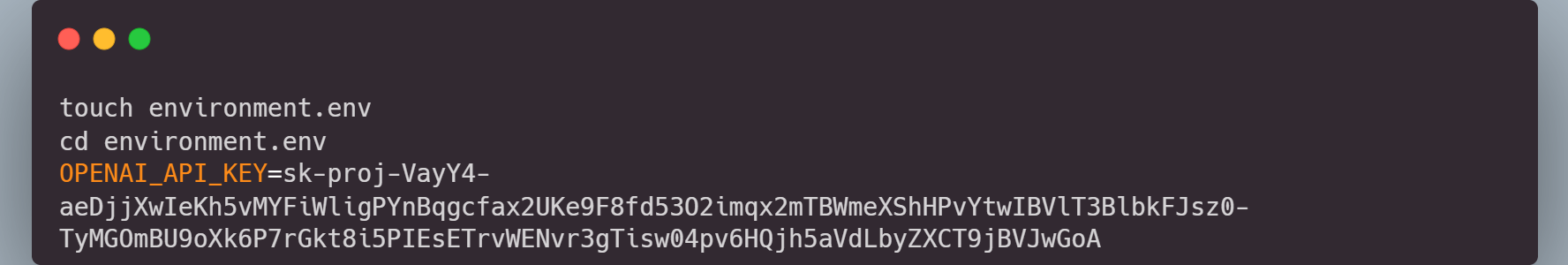








Once Redis is running, create an environment.env file in the project directory to store the OpenAI API key:



For convenience here is the OpenAI Key:

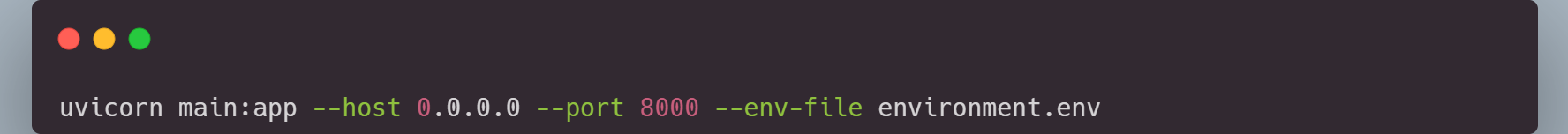
OPENAI\_API\_KEY=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

For the XTTS voice cloning feature, create a dedicated folder called “XTTS\_Packages” and clone the coqui/XTTS-v2 model from Hugging Face:

A black screen with white text

Description automatically generated

When these steps are complete, simply launch the server with Uvicorn, specifying both the host/port and the environment file:



uvicorn main:app --host 0.0.0.0 --port 8000 --env-file environment.env

Congratulations maintainer you now can run our VoIP server correctly on your local machine.

1. **Server File Explanations**

**main.py**

main.py is the central entry point of the application. It initializes the FastAPI server, sets up API routes and WebSocket endpoints, and manages near real-time audio call logic—including storing/transcribing audio chunks and sending TTS responses. It also handles connection tracking and Redis state management, ensuring the application can serve both HTTP requests and live WebSocket connections.

**user.py**

user.py defines the User class, which encapsulates all key information about a connected user—such as their WebSocket connection, unique ID, profile details, language preference, and voice embeddings. It also includes a language mapper for XTTS support and a helper print method for debugging user data.

**audioAccumulator.py**

audioAccumulator.py is responsible for accumulating raw PCM audio data and determining when enough has been collected for transcription. It uses an asynchronous lock to handle concurrent tasks safely, monitoring both data size and time thresholds. Once a threshold is met, it returns the accumulated audio, clears the buffer, and continues gathering new chunks.

**externalAPIs.py**

externalAPIs.py contains classes that interact with external AI services:

* **OpenAI Whisper** for speech-to-text transcription
* **OpenAI ChatGPT** for text translation
* **OpenAI TTS (with gTTS fallback)** for text-to-speech synthesis

Each class handles its service-specific logic, such as audio format conversions, API calls, and returning processed text or audio.

## 10.2 Android App:

* 1. **Operating Environment**

to maintain and update the Android app, ensure you have the following setup:

* **Android Studio (Recommended IDE)** – Other IDEs can be used but require external tools.
* **Gradle Version: 7.3.3** – Managed via Gradle Wrapper and should be set up automatically.
* **JDK Version: Java 11** (11.0.25)
* **Minimum SDK:** Android 8.0 (API Level 26)

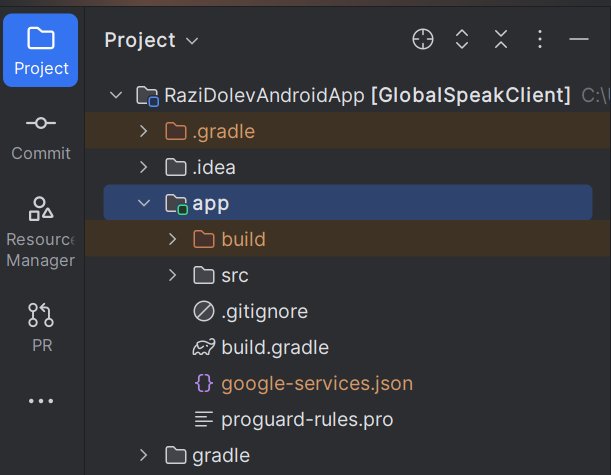
1. **Installation Instructions:**
2. clone the project’s GitHub repository: <https://github.com/razimograbi/RaziDolevAndroidApp.git>

* Open the project in Android Studio
* Sync Gradle
* Check JDK version (11)
* Dependencies will install automatically

1. Add A FireBase configuration file.

The app uses Firebase for user management. To link the app to your Firebase project, you need to add your own Firebase configuration file.

Steps to Get Your Firebase Config File:

* Follow this guide: <https://firebase.google.com/docs/android/setup>
* Click Download google-services.json to obtain your Firebase Android config file.
* Move the google-services.json file into your app-level root directory (app/ in Project View)

This is how the authentication and firestore database schema looks like:

תמונה שמכילה טקסט, גופן, מספר, קו

התיאור נוצר באופן אוטומטי

**תמונה שמכילה טקסט, גופן, צילום מסך, אלגברה

התיאור נוצר באופן אוטומטי**

Current signup method are:  
תמונה שמכילה טקסט, גופן, צילום מסך

התיאור נוצר באופן אוטומטי

* 1. **Modifying and expanding app features:**

Adding logic/modifying screens:

* You can modify/add Activity Java Files (.java) in com.example.globalspeakclient/
* If interacting with Firebase, use FirestoreService or AuthService where all the methods that interact with firebase exists.

תמונה שמכילה טקסט, צילום מסך, גופן

התיאור נוצר באופן אוטומטי

Adding/ modifying Screens Design:

* You can modify/add XML Layout Files in res/layout/
* Customize colors in res/values/colors.xml
* Modify text strings in res/values/strings.xml
* Update themes in res/values/themes.xml
* Apply styles using Material Components (e.g., com.google.android.material.button.MaterialButton)

תמונה שמכילה טקסט, גופן, צילום מסך, עיצוב

התיאור נוצר באופן אוטומטיתמונה שמכילה טקסט, גופן, צילום מסך, עיצוב

התיאור נוצר באופן אוטומטי

To run the app locally you need to add the following endpoints to the Main CallDashboard and AudioRecording Activities.

תמונה שמכילה טקסט, צילום מסך, גופן

התיאור נוצר באופן אוטומטי

תמונה שמכילה טקסט, צילום מסך, גופן, קו

התיאור נוצר באופן אוטומטיTo run the app on remote server:

Please do not try to test the application on the emulator as it does not support some of the features of the app. Run it on real device for testing.