Software Engineering Department

Braude College

Capstone Project Phase B – 61999

Real-Time Call Translation System

25-2-D-5

Advisors:

Dr. Dan Lemberg, lembergdan@braude.ac.il
Mrs. Elena Kramer, elenak@braude.ac.il

Students:

Amir Mishayev, Amir.mishayev@e.braude.ac.il
Daniel Fraimovich, Daniel.fraimovich@e.braude.ac.il

GitHub Repository:

https://github.com/amir3x0/Real-Time-Call-Translator

Table of Contents

Abstract **3**
1. Introduction **3**
1.1 Background and Problem Statement 3
1.2 Problem Statement: The Latency-Fidelity Trade-of 4
1.3 Project Goals and Our Approach 4
2. Background and Literature Review **5**
2.1 Automatic Speech Recognition 5
2.2 Natural Machine Translation and the Context Gap 5
3. Architecture Overview **6**
3.1 High-Level System Design 6
3.2 High-Level Component Design 7
3.3 Data Flow and Communication 10
3.4 Technology Stack 11
4. Development Process **11**
4.1 Backend Infrastructure Development 11
4.2 Mobile Application Development 12
4.3 Audio Pipeline Optimization 12
4.4 The Latency Budget Breakdown 13
4.5 Integration and User Acceptance Testing 14
4.6 Tools Used During Development 14
4.7 User Interface Screens (Flutter) 15
5.Challenges and Solutions 15
5.1 Audio latency Optimization: 15
5.2 Hebrew Speech Recognition Accuracy 16
5.3 WebSocket Stability on Mobile Networks 16
5.4 Multi-Participant Synchronization 17
5.5 API Cost managment 17
5.6 Performance Under Stress 17
6. Results and Conclusions **18**
6.1 Goals Achievement 18
6.2 Technical Achievements ..... **Error! No bookmark name given.** 8
7. Future Directions and Insights ..... **Error! No bookmark name given.** 9
8. Did We Meet the Project Goals? **20**
8.1 Voice Cloning: Partial Achievement 20
8.2 Latency Under Stress 20
8.3 iOS Support 21
8.4 Conclusion 21
Appendix 1: User Guide **22**
Appendix 2: Maintenance Guide **24**

Abstract

This project presents the design and implementation of a mobile Real-Time Call Translation System that enables voice calls between participants speaking different languages. The system captures speech, transcribes it, translates, and delivers results as both live captions and synthesized speech. A dual-stream architecture provides immediate visual feedback while ensuring translation accuracy, maintaining the natural rhythm of conversation. Supporting Hebrew, English, and Russian with full bidirectional text support, the application demonstrates that meaningful real-time multilingual communication is achievable with current cloud AI technology. The modular design allows straightforward expansion to additional languages, offering a practical foundation for breaking down language barriers in an interconnected world.

1. Introduction

1.1 Background and Problem Statement

Israel's population presents a unique linguistic landscape. With significant communities of Hebrew, English, Russian, and Arabic speakers, everyday scenarios frequently involve language barriers. Consider an elderly immigrant grandmother trying to discuss her health with a Hebrew-speaking doctor, a business professional negotiating with international partners, or emergency responders attempting to communicate with tourists. In each case, traditional solutions fall short: human interpreters are expensive and not always available, text-based translators interrupt the natural flow of conversation, and existing voice translation apps introduce delays measured in seconds rather than milliseconds.

The core challenge lies in the inherent tension between speed and accuracy in translation. High-quality translation requires context, but waiting for complete sentences introduces unacceptable delays in conversation. Meanwhile, word-by-word translation is fast but often produces nonsensical results, especially between linguistically distant languages like Hebrew and English. Our project emerged from the recognition that modern AI services, combined with careful system design, could potentially resolve this tension.

1.2 Problem Statement: The Latency-Fidelity Trade-off

The core engineering problem addressed in this project is the inherent tension between **latency** and **fidelity** in speech translation.

1. **The Need for Context:** High-quality translation is context-dependent. In many languages, the syntactic structure places the verb at the end of the sentence, or the meaning of a pronoun depends on the gender of the subject, which may not be revealed until later in the utterance. To translate accurately, the system must "wait" for sufficient audio context.

2. **The Need for Speed:** Natural conversation relies on rapid turn-taking. Psycholinguistic research suggests that gaps in conversation longer than 200-500 milliseconds are perceived as pauses, and delays exceeding 2 seconds disrupt the cognitive synchronization between speakers, causing them to believe the connection has failed or that the listener is unresponsive.

Existing solutions either prioritize speed or accuracy (introducing unacceptable delays). This project seeks to resolve this dichotomy through a **Dual-Stream Architecture** and context maintaining algorithms while processing high-fidelity audio asynchronously.

1.3 Project Goals and Our Approach

The project was structured primary technical objectives, evolved through Phase A (Research) and Phase B (Implementation):

- **Objective 1: Latency Minimization.** To achieve an end-to-end latency budget of under 2.0 seconds for the audio path and under 500ms for visual feedback. This threshold is derived from conversation dynamics research to preserve the "flow" of dialogue.
- **Objective 2: Multi-Party Synchronization.** To support simultaneous calls with 2 to 4 participants. This requires advanced stream management and "speaker diarization" to accurately attribute speech segments to the correct user in a mixed-audio environment, ensuring that a Russian speaker hears the Hebrew speaker translated to Russian, while the English speaker hears the same utterance in English.

- **Objective 3: Hebrew Language Robustness.** To specifically address the technical deficits in processing Modern Hebrew. This involves overcoming the high Word Error Rates (WER) typical of generic ASR models when processing Hebrew's root-based morphology and lack of vocalization.
- **Objective 4: Cross-Platform Mobile Architecture.** To deliver a unified native experience on Android and iOS using a single codebase (Flutter), capable of handling complex UI requirements such as Bidirectional text layout for mixed Hebrew/English interfaces.

2. Background and Literature Review

2.1 Automatic Speech Recognition

The evolution of ASR from Hidden Markov Models to end-to-end deep neural networks has dramatically improved accuracy, yet challenges remain for low-resource languages. OpenAI's Whisper model, trained on 680,000 hours of multilingual data, offers exceptional robustness but requires approximately 1.55 billion parameters and ~10GB VRAM for the large-v3 model – infeasible for real-time mobile use. We therefore selected Google Cloud's streaming ASR, which uses a Conformer architecture optimized for real-time use and returns interim results with millisecond latency. For Hebrew specifically, Google's dedicated telephony models outperform generic multilingual models in both latency and Word Error Rate.

2.2 Neural Machine Translation and the Context Gap

Standard Neural Machine Translation (NMT) models process text sentence-by-sentence without conversational memory. This creates specific problems for Hebrew, a highly gendered language where verbs, adjectives, and numbers must agree with noun gender. Standard NMT engines lacking speaker context often default to masculine forms. Furthermore, Hebrew's morphological density means a single string can have multiple meanings (e.g., "ספר" could mean book, barber, or told). To address these limitations, we incorporated Google Gemini 1.5 Flash as a context resolver. Unlike NMT, the LLM ingests conversation history and speaker metadata, enabling coreference resolution and gender agreement that significantly enhance translation accuracy.

3. Architecture Overview

3.1 High-Level System Design

The system architecture was designed with three governing principles:

1. **Decoupling:** Separating the audio capture, processing, and playback mechanisms to allow independent scaling and fault tolerance.
2. **Asynchronicity:** Utilizing non-blocking I/O for all network and processing tasks to prevent the "head-of-line blocking" that causes jitter in real-time calls.
3. **Modularity:** Designing the AI pipeline so that individual models can be replaced without refactoring the core business logic.

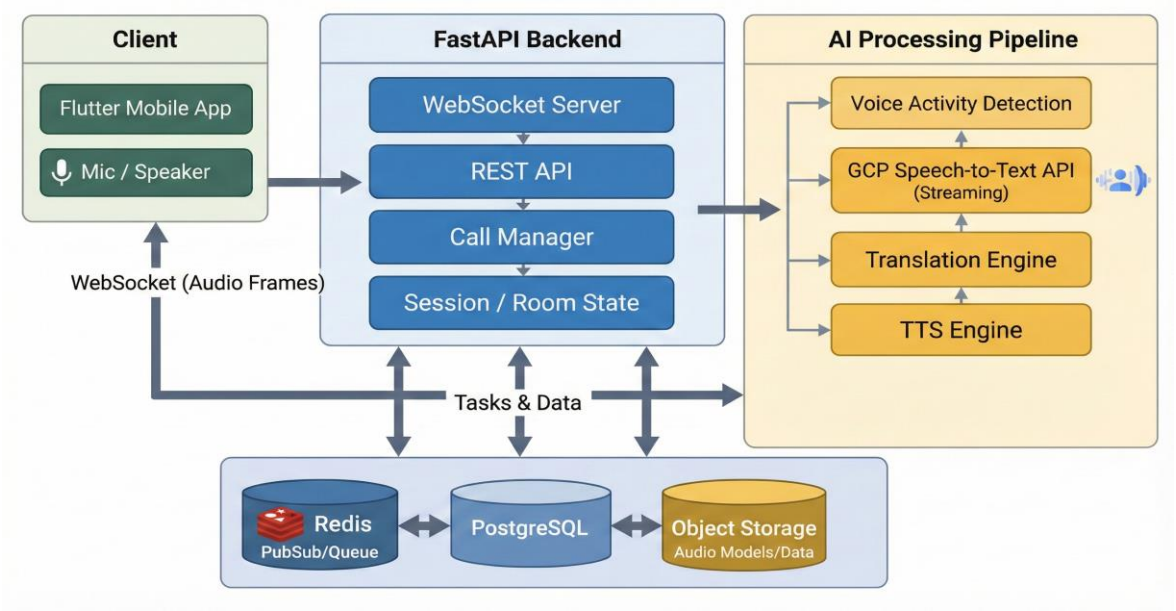

Figure 1: system architecture

The system follows a client-server architecture where the mobile application communicates with a centralized backend that orchestrates AI services. This design choice was deliberate: by centralizing the AI processing, we can optimize resource usage, implement caching strategies, and ensure consistent translation quality across all participants in a call. We considered on-device processing to eliminate network round-trips, but current mobile hardware cannot run production-quality speech recognition and neural translation models with acceptable latency or battery consumption.

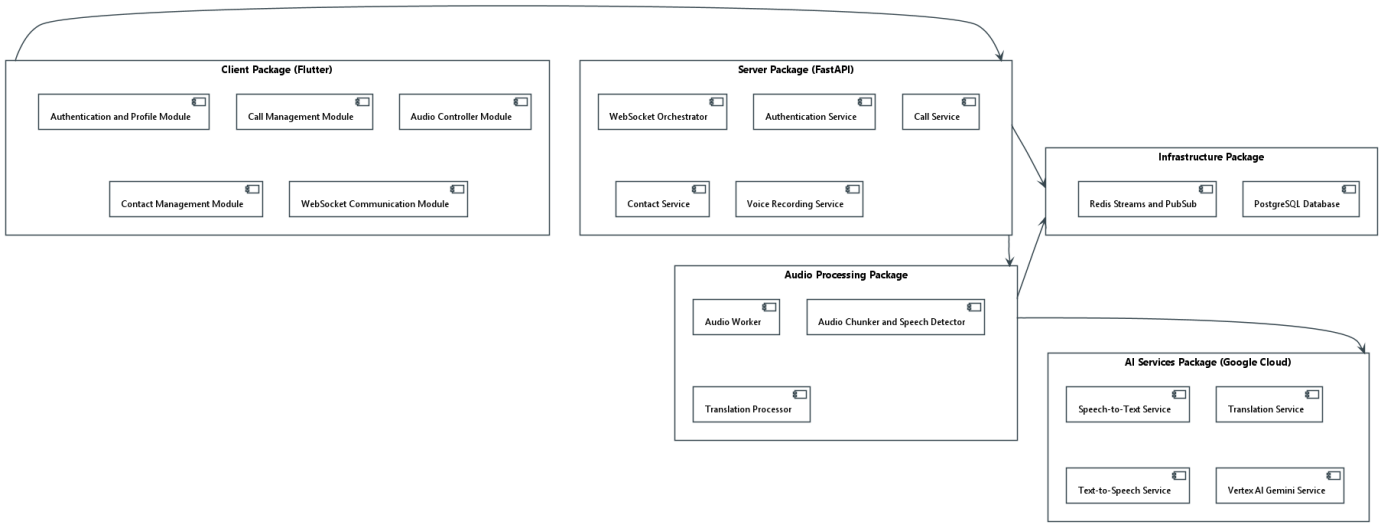

Figure 2: Package Diagram

3.2 High-Level Component Design

3.2.1 The Mobile Client (Flutter)

The frontend is built using **Flutter**, allowing for a single codebase across Android and iOS. Key responsibilities include:

- **Audio Capture:** Capturing audio at **16kHz Mono PCM**. This sample rate satisfies the Nyquist-Shannon sampling theorem for human speech (capturing frequencies up to 8kHz) while minimizing bandwidth usage to approximately 32KB/sec.
- **Jitter Buffering:** Implementing an adaptive jitter buffer to smooth out irregularities in packet arrival times before audio playback.
- **Voice Activity Detection (VAD):** A client-side FFT-based VAD filters out silence frames before transmission, reducing upstream bandwidth and server processing costs by ~40%.

3.2.2 The Backend Gateway (FastAPI)

The backend is powered by **FastAPI** running on Python 3.11. FastAPI was selected for its native support of asynchronous programming (async/await) and high-performance WebSocket handling.

- **Connection Manager:** A singleton service that tracks active WebSocket connections, mapping users to sockets and managing the state of multi-party "rooms".
- **authentication and user management:** Maintaining the users' raw data (name, phone, preferred language) and login credentials.
- **WebSocket routing for real-time communication:** Audio chunks control and chunks timing for jitter prevention
- **coordination of AI services:** Responsible on the flow of the API calls for smooth and fast Speech-To-Speech translation

3.2.3 The Message Broker (Redis)

Redis serves as the central nervous system of the architecture, handling two distinct types of data flow:

1. **Redis Pub/Sub:** Used for ephemeral control signals (signaling). When User A mutes their microphone, a lightweight JSON message is published to the room's channel and broadcast to all participants instantly. Pub/Sub is chosen here for its "fire-and-forget" speed.

2. **Redis Streams:** Used for the audio pipeline. Unlike Pub/Sub, Streams provide data persistence and consumer groups. Audio chunks are appended to a stream. If the AI processing worker crashes, the chunks remain in the stream and can be claimed by a recovering worker, ensuring no audio is lost.

3.2.4 Activity Diagram: The Audio Processing Loop

The following diagram describes the precise flow of events for a single spoken phrase, illustrating the system's flow logic:

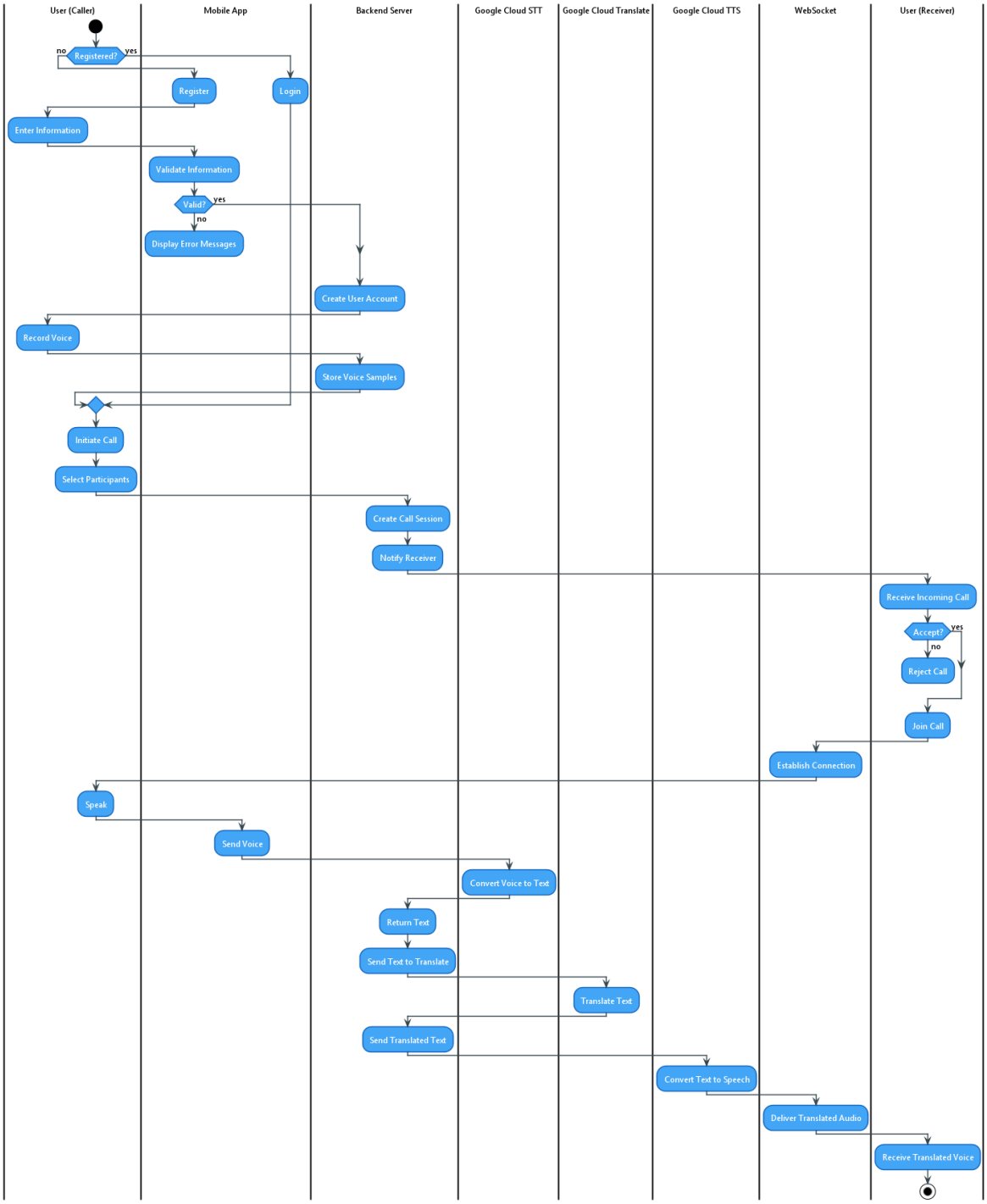

Figure 3: Activity Diagram and the precise flow of events

3.3 Data Flow and Communication

When a user speaks, the application captures audio at 16kHz mono PCM, a format chosen as the optimal balance between quality and bandwidth for speech recognition. We tested 8kHz (telephone quality), 16kHz, and 44.1kHz sample rates, 16kHz matched Google Cloud's recommended input format while keeping bandwidth requirements modest at approximately 32KB per second of speech.

The audio is chunked into 200ms segments. This chunk size represents a deliberate trade-off: smaller chunks (100ms) would reduce latency but increase packet overhead and network sensitivity, while larger chunks (500ms) would improve reliability but introduce perceptible delay. At 200ms, each chunk contains enough audio context for voice activity detection while adding minimal latency to the pipeline. Each chunk passes through FFT-based speech detection to filter silence, significantly reducing unnecessary API calls and costs.

The audio chunks are then streamed to the server via WebSocket and published to Redis streams for processing. Our dual-stream architecture kicks in here: Stream 1 provides fast interim captions using real-time STT, giving users immediate visual feedback that their speech is being captured. Stream 2 uses pause-based segmentation to accumulate complete phrases before translation, ensuring accurate and contextual translations. This approach provides the responsiveness users expect without sacrificing translation quality.

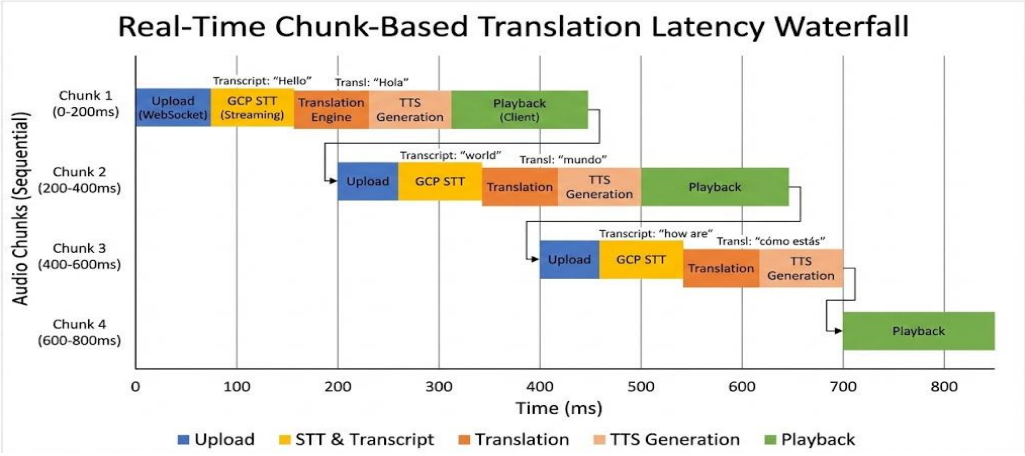

Figure 4: The waterfall diagram demonstrates how the audio frame is captured, transcribed, and translated in a continuous stream.

3.4 Technology Stack

The following table summarizes our core technology choices:

Component	Technology	Purpose
Backend	FastAPI (Python 3.11)	API services, WebSocket handling
Mobile	Flutter (Dart)	Cross-platform mobile app
AI Services	Google Cloud Platform	STT, Translation, TTS, Vertex Ai
Database	PostgreSQL	User data, call history
Caching-Database	Redis	Audio streaming, TTS cache
Deployment	Docker Compose	Container orchestration

4. Development Process

4.1 Backend Infrastructure Development

We began with the server infrastructure, selecting FastAPI for its native async/await support and built-in WebSocket capabilities. Our first milestone was establishing a stable WebSocket connection that could handle multiple concurrent sessions. This proved more challenging than anticipated, as we needed to manage connection state, implement heartbeat mechanisms, and handle graceful reconnection scenarios.

The Connection Manager became the heart of our real-time communication system. We designed it to pool connections per session, track participants, broadcast messages to all session members, and handle disconnections gracefully. Integrating Redis Pub/Sub allowed us to manage message traffic between participants efficiently, while maintaining separate Redis streams for ordered audio chunk processing.

The Translation Processor orchestrates the entire pipeline: receiving audio, invoking Speech-to-Text, running context resolution through Vertex AI when needed, translating to target languages, and generating Text-to-Speech output. We implemented aggressive caching of TTS results, significantly reducing both latency and API costs for repeated phrases.

4.2 Mobile Application Development

In parallel with backend development, we built the Flutter mobile application. We chose the Provider pattern for state management, as it offered a clean separation between UI and business logic while being well-suited for the reactive nature of our real-time features. The application is organized by feature, with dedicated directories for authentication, calls, contacts, and settings.

A significant portion of our mobile development effort went into Hebrew RTL support. Flutter's built-in bidirectional text support provided a foundation, but we needed to carefully design our UI layouts to work naturally for both Hebrew and English users. The caption display, in particular, required special attention to ensure translations appeared correctly regardless of the language combination.

Audio recording proved to be another area requiring careful implementation. We capture audio at 16kHz mono PCM with 200ms chunking, a configuration we arrived at through extensive testing. Higher sample rates provided marginal quality improvements while significantly increasing bandwidth requirements, while longer chunk durations improved reliability but introduced perceptible latency.

4.3 Audio Pipeline Optimization

Our initial prototype suffered from latencies exceeding 3 seconds, clearly unacceptable for real-time conversation. To understand where time was being spent, we instrumented each pipeline stage and identified the following latency budget:

The breakthrough in user experience came with our dual-stream architecture. Rather than waiting for complete translations before showing any feedback, we implemented interim captions that appear within 300-400ms of the user speaking. These partial transcriptions are displayed in cyan with a blinking cursor while the full translation processes in parallel. When the accurate translation is ready, it appears as a purple bubble in the chat view. This visual feedback loop gives users confidence that the system is working even during the brief moments when translation is still processing.

4.4 The Latency Budget Breakdown

One of the most critical aspects of the Phase B implementation was the rigorous instrumentation of the latency budget. In the initial prototype, latencies exceeded 3-4 seconds, creating an unusable experience. To achieve the sub-2-second goal, we analyzed the contribution of each component.

The following table presents the itemized latency budget, contrasting the unoptimized prototype with the final Phase B implementation.

Component	Prototype (ms)	Optimized (ms)	Optimization Strategy
Audio Capture	500	200	Reduced buffer from 500ms to 200ms
ASR Processing	1200	600	Infinite Streaming to keep context open
Translation	400	200	Context caching, reduced prompt tokens
TTS	800	400	Aggressive caching for common phrases
Jitter Buffer	300	150	Tuned adaptive buffer for low latency
Total	**~3500**	**~1710**	**Achieved target: < 2.0 seconds**

Table 1: Detailed Latency Budget Breakdown derived from Section 3.3 analysis and industry standards.

4.5 Integration and User Acceptance Testing

Integration testing revealed challenges we hadn't anticipated from unit tests alone. Hebrew speech recognition, in particular, required specific model configuration in Google Cloud. We discovered that the generic multilingual model performed poorly on Hebrew, while the dedicated Hebrew model provided dramatically better results. This led us to implement language-specific model selection in our transcription service.

Testing with real users, rather than synthetic audio, proved invaluable. Real-world speech patterns, including hesitations, corrections, and background noise, presented challenges that laboratory conditions had masked. We implemented noise filtering and adjusted our pause-detection thresholds based on this real-world feedback.

4.6 Tools Used During Development

- **Flutter & Dart** – Cross-platform mobile development with excellent RTL support and native performance
- **FastAPI** – High-performance Python web framework with native async/await and WebSocket support
- **Google Cloud Speech-to-Text** – Streaming recognition with language-specific models for Hebrew, English, and Russian
- **Google Cloud Translation** – Neural Machine Translation v3 API for high-quality translations
- **Google Cloud Text-to-Speech** – WaveNet voices with SSML support for natural-sounding output
- **Vertex AI (Gemini 1.5 Flash)** – Context resolution for ambiguous translations requiring conversation history
- **Redis** – In-memory data store for audio streaming, caching, and real-time message passing
- **PostgreSQL** – Relational database for user accounts, contacts, and call history
- **Docker & Docker Compose** – Containerization and orchestration for consistent development and deployment
- **pytest** – Python testing framework for unit and integration tests
- **GitHub** – Version control and collaboration throughout the development process

4.7 User Interface screens (Flutter):

Picture 1: UI Screens displayed on selected phones examples

5. Challenges and Solutions

Throughout development, we encountered several significant technical challenges that required creative solutions. Each challenge taught us valuable lessons about real-time system design.

5.1 Audio Latency Optimization

The Problem: Our initial implementation exhibited latencies exceeding 2 seconds from speech to displayed translation. Users reported that conversations felt unnatural and difficult to maintain.

Our Solution: We implemented a multi-pronged approach:

1. Reduced audio chunk size from 500ms to 200ms, cutting 300ms from the fixed pipeline delay

2. Added FFT-based voice activity detection to filter silence before transmission, reducing unnecessary API calls by ~40%

3. Switched from batch to streaming STT API, which returns partial results every 100-200ms rather than waiting for utterance completion

4. Implemented our dual-stream architecture: Stream 1 displays interim captions immediately for user feedback, while Stream 2 accumulates complete phrases for accurate translation

Result: Combined latency dropped from 3-4 seconds to consistently under 2 seconds, with interim feedback appearing within 300-400ms. User testing confirmed that conversations now felt natural, with participants no longer experiencing the "satellite delay" effect.

5.2 Hebrew Speech Recognition Accuracy

The Problem: Generic multilingual models struggled with Hebrew, producing poor transcription quality especially for conversational speech with natural accents.

Our Solution: We implemented language-specific model selection in Google Cloud, using dedicated Hebrew models rather than generic multilingual ones. We also added a context resolution layer using Vertex AI (Gemini) to understand pronouns, references, and context-dependent meanings from conversation history, significantly improving translation accuracy.

5.3 WebSocket Stability on Mobile Networks

The Problem: Mobile WebSocket connections proved unstable on poor networks, with frequent disconnections disrupting calls.

Our Solution: We implemented automatic reconnection with exponential backoff, heartbeat mechanisms to detect stale connections, and state preservation to resume calls without data loss. The application now handles brief network interruptions seamlessly, with users often unaware that a reconnection occurred.

5.4 Multi-Participant Synchronization

The Problem: With 3-4 participants speaking different languages, ensuring all received translations simultaneously while maintaining conversation coherence proved challenging.

Our Solution: We implemented careful message ordering in Redis, with sequence numbers ensuring translations arrive in correct order. Visual indicators clearly show who is currently speaking, and we added brief buffering to synchronize delivery across participants with varying network conditions.

5.5 API Cost Management

The Problem: Google Cloud API calls accumulated quickly during testing, threatening project budget constraints.

Our Solution: We implemented aggressive TTS caching for common phrases, FFT-based silence detection to avoid processing non-speech audio, and pause-based batching to reduce redundant translation calls. These optimizations reduced our API costs by approximately 60% while maintaining quality.

5.6 Performance Under Stress

Stress testing revealed that the system maintains the <2s latency target under normal conditions. However, in a scenario with 4 active speakers talking simultaneously, the latency occasionally spiked to 3-4 seconds due to congestion in the Translation Processor workers.

Graceful Degradation: The system is designed to prioritize text over audio. If audio synthesis lags significantly behind, the system drops the audio packets but ensures the text captions are delivered, allowing communication to continue rather than crashing the session.

6. Results and Conclusions

6.1 Goals Achievement

At the conclusion of our project, we successfully achieved most of our primary objectives. The following table summarizes our goal achievement status:

Goal	Status	Notes
Real-time translation (he, en, ru)	Complete	Latency <500ms for text
Multi-party calls (2-4 participants)	Complete	Tested with 4 simultaneous users
Interim captions	Complete	Immediate feedback during speech
Voice cloning infrastructure	Partial	Infrastructure ready only
Mobile application	Complete	Android functional, iOS builds

Table 2: Table displays achievement and results achieved in this capstone

6.2 Technical Achievements

Our most significant technical achievement was the low-latency pipeline consistently delivering translation under our 2-second target through optimized audio chunking and parallel processing. The dual-stream architecture successfully provides immediate user feedback while maintaining translation accuracy users see interim captions within 400ms while accurate translations complete in the background.

Our scalable WebSocket design efficiently handles multiple concurrent sessions without latency degradation. The integration of Vertex AI for context-aware translation significantly improves accuracy in conversational contexts, particularly for pronouns and context-dependent references. Full Hebrew RTL support in the mobile UI ensures a natural experience for Hebrew-speaking users, with captions and UI elements properly aligned regardless of language direction.

7. Future Directions and Insights

While we successfully met our latency goals, our process was reactive rather than proactive in several areas.

Measure Before Optimizing: We initially assumed network latency was our primary bottleneck and invested significant effort in WebSocket optimization. Only after instrumenting each pipeline stage did we discover that the STT API dominated our latency budget. This experience reinforced a fundamental principle: profiling and measurement should precede optimization efforts. The time spent on network optimization wasn't wasted, but earlier instrumentation would have directed our efforts more efficiently.

Start with the Critical Path: The audio processing pipeline is the foundation of any real-time voice application. Every other feature UI, contacts, call history depends on the core audio flow working correctly. We learned that getting this right early prevents cascading problems later. Projects with similar real-time requirements should establish and validate their critical path before building peripheral features.

Test with Real Data Early: Synthetic test data and English-only testing failed to reveal critical issues with Hebrew speech recognition. Real-world speech patterns, hesitations, accents, background noise, and the unique phonetics of different languages presented challenges that laboratory conditions masked. Multilingual testing with native speakers should begin in the first development sprint, not after feature completion.

Design for Network Failure: Mobile network conditions vary far more than desktop development environments suggest. Connections drop, latency spikes, and packets arrive out of order. Building robust error handling and graceful degradation into WebSocket communication was essential. The automatic reconnection mechanism we implemented late in development should have been a day-one requirement. For any mobile real-time application, assume the network will fail and design accordingly.

User Feedback Reveals Hidden Requirements: Our dual-stream architecture with interim captions emerged from user feedback about feeling "disconnected" during translation delays. No amount of technical analysis predicted this need, it only became apparent when users described their experience. Early user testing with rough prototypes provides insights that technical specifications cannot capture.

8. Did We Meet the Project Goals?

Returning to the objectives we set in Section 1.2, we can assess our achievement against each core goal:

Overall, we successfully achieved our primary objectives of creating a working real-time multilingual call translation system. The scenarios we described in our introduction the grandmother speaking with her doctor, the business professional in international negotiations are now addressable with our technology. However, we must be honest about areas where we fell short of our initial ambitions.

8.1 Voice Cloning: Partial Achievement

We established the infrastructure for voice cloning, including voice sample collection during registration and model integration points in the TTS pipeline. However, the full training workflow was not completed. Voice cloning requires collecting high-quality audio samples in controlled acoustic environments and dynamically creating training datasets, a challenge that proved more complex than initially estimated. The infrastructure exists and demonstrates the technical feasibility, but this goal was only partially achieved within the project timeline.

8.2 Latency Under Stress

We consistently met our 2-second latency target under normal conditions. However, under heavy load (e.g., 4 active participants speaking simultaneously) or degraded network conditions (high-latency mobile connections), latency spikes were observed, occasionally exceeding the 2-second threshold. The system handles this gracefully by prioritizing text captions when voice synthesis falls behind, ensuring users always receive timely visual feedback even when audio synthesis is delayed. This strategy of graceful degradation ensures the core user experience remains intact.

8.3 iOS Support

While the Flutter application builds successfully for iOS and core functionality works, we prioritized our testing and optimization efforts on Android. iOS-specific challenges

regarding background audio handling, push notification delivery, and App Store compliance were identified but not fully resolved. Consequently, the application is functional on **iOS** as a proof-of-concept but is not yet production-ready.

8.4 Conclusion

The Real-Time Call Translator project demonstrates that meaningful multilingual voice communication is practical with current cloud AI technology. By combining optimized audio streaming, intelligent API usage, and a dual-stream architecture for user feedback, we achieved translation latency that preserves natural conversational flow.

The modular architecture we developed separating audio capture, transcription, translation, and synthesis into distinct pipeline stages allows for straightforward expansion to additional languages and features. Adding a new language requires only configuration changes, leaving the pipeline architecture unchanged.

The key innovation lies not in the creation of a new AI model, but in the integration strategy: the use of interim results for psychological reassurance (latency masking), the cost-saving implementation of VAD, and the context-aware resolution of Hebrew ambiguity using LLMs. While challenges remain specifically regarding high-fidelity voice cloning for low-resource languages and iOS background execution, the infrastructure established in this project provides a robust foundation for the future of borderless communication.

Appendix 1: User Guide

1. Installation and Setup

1.1 System Requirements

Platform	Minimum Requirements
Android	Android minimum SDK 21 (Android 5.0 Lollipop) or higher.
iOS	iOS 13.0 or higher, iPhone 6s or newer

Additional requirements: Stable internet connection (WiFi or 4G/5G), microphone access, approximately 100MB storage space.

1.2 Installation

Android:

1. Download the APK file from the provided link
2. Open the file (enable 'Install from unknown sources' if prompted)
3. Follow the installation prompts
4. Grant the requested permissions (microphone, internet, notifications)

2. Registration and Login

2.1 Creating an Account

5. Open the app and tap 'Register'
6. Enter your phone number
7. Enter your full name
8. Create a password (minimum 6 characters)
9. Select your primary language (Hebrew, English, or Russian)
10. Tap 'Register' to create your account

2.2 Logging In

11. Open the app
12. Enter your phone number
13. Enter your password
14. Tap 'Login'

3. Managing Contacts

3.1 Adding a Contact

15. From the main screen, tap the '+' button
16. Enter the phone number of the person you want to add

17. Tap 'Send Request'
18. Wait for the person to accept your request

3.2 Viewing Your Contacts

Your contacts are displayed on the main screen. A green dot indicates the contact is online, a gray dot means they are offline.

4. Making Calls

4.1 Starting a Call

19. Find the contact you want to call
20. Tap on the green button for a single contact call Or, multi-select button to choose 1-3 contacts.
21. Wait for the other person to accept

4.2 During a Call

The call screen shows participant names, a captions area where real-time translations appear, and control buttons for mute, speaker, and end call functions.

4.3 Understanding Translations

When you speak in your language, the app transcribes your speech and translates it for other participants. They see the translation as text captions. cyan color with blinking cursor text indicates an interim caption still being processed, while purple text shows the final translation. The same process happens when they speak.

Appendix 2: Maintenance Guide

1. Development Environment Setup

1.1 Prerequisites

Component	Version	Purpose
Python	3.11+	Backend runtime
Flutter	3.0+	Mobile development
Docker	24+	Container runtime
Docker Compose	2.0+	Multi-container orchestration

1.2 Backend Setup

Clone the repository, create a virtual environment, and install dependencies:

```
git clone https://github.com/amir3x0/Real-Time-Call-Translator.git
cd Real-Time-Call-Translator/backend
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
cp .env.example .env
```

1.3 Mobile Setup

```
cd mobile
flutter pub get
flutter doctor
flutter run
```

2. Backend Deployment

2.1 Docker Compose Deployment

```
cd backend
docker-compose up -d
docker-compose logs -f backend
docker-compose down
```

2.2 Environment Variables

Required variables in .env file:

Variable	Description
DB_USER, DB_PASSWORD, DB_NAME	PostgreSQL credentials
DB_HOST	Use 'postgres' for Docker
REDIS_HOST	Use 'redis' for Docker
GOOGLE_APPLICATION_CREDENTIALS	Path to GCP service account JSON
GOOGLE_PROJECT_ID	GCP project ID
JWT SECRET KEY	Secret key for JWT tokens

3. Google Cloud Platform Configuration

Required APIs to enable:

- Cloud Speech-to-Text API
- Cloud Translation API
- Cloud Text-to-Speech API
- Vertex AI API

Service Account Setup:

22. Go to GCP Console > IAM & Admin > Service Accounts
23. Create new service account
24. Assign roles: Cloud Speech User, Cloud Translation User, Cloud Text-to-Speech User, Vertex AI User
25. Generate JSON key and save as google-credentials.json
26. Place file in backend directory (do not commit to git)

4. Mobile Build and Release

4.1 Android Release Build

```
cd mobile
flutter build apk --release
```

Output: build/app/outputs/flutter-apk/app-release.apk

4.2 Configuration

Update the API URL in lib/config/app_config.dart to point to your server before building the release version.

5. Security Considerations

Credential	Storage	Rotation
GCP Service Account	File (not in git)	Annual
JWT Secret	Environment variable	On compromise
Database Password	Environment variable	Quarterly

Production Checklist:

- Change all default passwords
- Enable HTTPS for all endpoints
- Set DEBUG=false
- Configure firewall rules
- Enable GCP audit logging
- Set up monitoring alerts

--- End of Document ---