Politecnico di Torino

Politecnico di Torino

Master's Degree in Artificial Intelligence and Data Analytics

A.a. 2024/2025

Graduation Session July 2025

Bridging the Communication Gap: A Mobile App for Seamless Integration of Sign Language in Real-Time Video Communication

Supervisors:

Prof. SARAH AZIMI

Dott. CORRADO DE SIO

Candidate:

DAVIDE NATALE

Summary

Bridging the communication gap between deaf and hearing individuals remains a significant challenge, particularly within specific domains such as education, healthcare and public administration. Although sign languages are fully recognized as natural languages, their limited integration into mainstream communication tools contributes to the social and linguistic marginalization of deaf communities worldwide. Addressing this gap requires the development of inclusive, privacy-aware, and real-time technological solutions that facilitate seamless interaction between individuals, regardless of their different modalities of communication.
This thesis aims to contribute to this objective by building an Android mobile application that integrates an AI-powered bidirectional translation engine.
Specifically, the project involves the design and implementation of two distinct, interconnected modules.
The first module is a backend system composed of multiple microservices, each running in a Docker container. The core service is a Node.js-based web server responsible for handling user authentication, video calls and user data, which is stored in a relational PostgreSQL database. In particular, for video calls, the server manages both the signaling process–using push notifications–and the routing and forwarding of media streams among participants. Additionally, to support real-time sign language translation, the server communicates with two Python-based microservices that process media streams by executing the corresponding AI translation algorithms.
The second module is an Android mobile application built with Expo, which interacts with the backend via RESTful APIs. Users can use it to engage in video-based interactions similarly to popular video conferencing tools, but enhanced by real-time sign language translation features.
The result of this work is a fully functional prototype designed to provide a solid foundation for future enhancements and further development.

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Sarah Azimi and Corrado De Sio, as well as to Federico Buccellato, for their constant availability, insightful guidance, and invaluable support throughout the development of this project. I am especially thankful for the opportunity they gave me to work on such a meaningful and stimulating topic.
I am also deeply grateful to my family and friends, in particular my cousin Valentina and my friend Alessandro Carosetti, for their support and encouragement throughout my academic journey. Their presence and belief in me have been essential in reaching this important milestone.

Table of Contents

List of Tables

List of Figures

Acronyms

API
Application Programming Interface

AI
Artificial Intelligence

REST
Representational State Transfer

CNN
Convolutional Neural Network

JSON
JavaScript Object Notation

XML
Extensible Markup Language

JWT
JSON Web Token

FCM
Firebase Cloud Messaging

YAML
Yet Another Markup Language

ML
Machine Learning

ORM

Object-Relational Mapping

SQL

Structured Query Language

HTTP

Hypertext Transfer Protocol

TTL

Time-to-Live

SMTP

Simple Mail Transfer Protocol

HTML

Hyper Text Markup Language

TCP

Transmission Control Protocol

SFU

Selective Forwarding Unit

RTP

Real-time Transport Protocol

ACID

Atomicity, Consistency, Isolation, Durability

URL

Uniform Resource Locator

STT

Speech-to-Text

TTS

Text-to-Speech

SLP

Sign Language Processing

SLD

Sign Language Detection

SLI

Sign Language Identification

SLS

Sign Language Segmentation

SLR

Sign Language Recognition

SLT

Sign Language Translation

SLP

Sign Language Production

RNN

Recurrent Neural Network

ASLR

Automatic Sign Language Recognition

LIS

Italian Sign Language

ASL

American Sign Language

LSF

French Sign Language

CSLR

Continuous Sign Language Recognition

S2G2T

Sign2gloss2text

S2T

Sign2text

S2(G+T)

Sign2(gloss+text)

G2T

Gloss2text

OTP

One-Time Password

UI

User Interface

UX

User Experience

FAB

Floating Action Button

CRUD

Create, Read, Update, Delete

WHO

World Health Organization

Chapter 1

Introduction

1.1 Deaf People and Sign Language

Deafness is a condition that affects millions of individuals globally and encompasses a broad spectrum of hearing impairments, ranging from partial hearing loss to complete deafness. According to the World Health Organization (WHO), over 1.5 billion people experience some degree of hearing loss, with approximately 430 million requiring rehabilitation services, including access to hearing aids and communication technologies [1].

Deaf individuals are commonly classified into two main groups: those who are prelingually deaf, meaning they lost their hearing before acquiring spoken language, and those who are postlingually deaf, who lost their hearing after having developed language skills. These classifications have significant implications for communication preferences and language acquisition strategies. Moreover, within the deaf population, a distinction is commonly made between *Deaf* with a capital 'D' and *deaf* with a lowercase 'd', which reflects not just the degree of hearing loss but also cultural and linguistic identity. The term *Big-D Deaf* refers to individuals who identify as part of a distinct sociolinguistic community, with sign language as their primary mode of communication and a shared cultural heritage. Conversely, *small-d deaf* refers to individuals with hearing loss who may rely on spoken language, hearing devices, or lip-reading, and who typically do not participate in Deaf culture. For members of the Deaf community, sign language plays a central role in communication and identity.

Sign languages are fully developed natural languages with their own grammar, syntax, and vocabulary, and they are independent from the spoken languages of their respective countries [2]. However, they are often not widely understood or supported outside the Deaf community, creating barriers to full participation in society.

These communication challenges significantly affect the everyday lives of deaf individuals. Inadequate access to interpreters, limited awareness among hearing individuals, and insufficient adoption of assistive technologies frequently lead to systemic exclusion across domains such as education, healthcare, and public administration. As a result, deaf individuals—especially those who use sign language—often face limited educational and professional opportunities, restricted access to essential services, and social marginalization.
These limitations underscore the urgent need for inclusive technologies that can bridge the communication gap between deaf and hearing individuals. The development of digital tools that can facilitate bidirectional communication—by translating sign language into spoken or written language and vice versa—has the potential to significantly improve accessibility and promote inclusion. In this context, computer vision, natural language processing, and mobile computing are emerging as promising enablers of real-time communication support for the deaf community.

1.2 Motivation

Bridging the communication gap between deaf and hearing individuals remains a pressing challenge, particularly in key contexts such as education, healthcare and public administration. Language barriers and the lack of supportive technologies significantly limit access to educational, professional, and social opportunities, leading to a sense of isolation for deaf communities worldwide.
In the academic domain, for example, deaf students struggle to follow lectures and interact with professors and peers due to the lack of appropriate supporting technologies. The introduction of solutions that integrate the use of sign language could help to bridge the existing gap, enabling more active and inclusive participation in university life. In the healthcare sector, such solutions could facilitate communication between medical professionals and deaf patients, improving diagnostic accuracy and the overall quality of care. Similarly, in public services, they could simplify access to information and legal rights, making interactions more efficient and inclusive. In recent years, this issue has gained increasing importance, attracting growing attention from the scientific community and leading to a rising number of studies aimed at developing increasingly innovative solutions.

1.3 Goal

The goal of this thesis is to contribute to the field by designing and developing an Android mobile application that facilitates seamless communication between deaf and hearing individuals. To support this objective, a backend system was developed

to integrate a real-time communication platform, enabling efficient bidirectional interaction. This backend communicates with the mobile application to provide real-time sign language translation functionalities.
The result of this work is a fully functional prototype, intended to serve as a solid starting point for future improvements and further development.

1.4 Thesis structure

In the following chapters, I will describe in detail the work carried out in this thesis. I will begin by providing an overview of the related works and theoretical backgrounds, which are essential to understand the context of this project. This includes a review of the most relevant studies in the fields of sign language processing, pose estimation, and real-time communication technologies.
Next, I will present the methodology adopted for the design and development of the entire system. In particular, I will provide a detailed overview of the system architecture, focusing on the design decisions made during the development process of both the mobile application and the backend system. Particular attention will be given to the integration of the AI-based translation engine and the interaction between the various components of the system.
Finally, I will conclude this work by discussing the results achieved and their implications for the field. This will include potential future improvements and development opportunities aimed at enhancing the system's capabilities and addressing the challenges encountered in real-world scenarios, leading to more inclusive communication solutions for the deaf communities.

Chapter 2

Related Works

This chapter presents an overview of related works in the field of sign language processing, which serves as the foundation and starting point for the development of this project.

2.1 Early Approaches

Over the past two decades, a wide range of technological solutions has been developed and proposed to bridge the communication gap between deaf and hearing individuals. Early approaches primarily focused on text-based messaging or Speech-to-Text (STT) and Text-to-Speech (TTS) transcription systems, allowing deaf users to engage in conversation through manual typing or automated subtitles. These methods are often natively integrated into major operating systems; however, they are largely limited to one-directional communication and fail to capture the non-verbal expressiveness and emotional nuances inherent in human interaction [3]. More advanced platforms attempted to directly interpret or produce sign language through the use of wearable sensors, depth cameras, or computer vision techniques. For example, the SignAloud project, developed by researchers at the University of Washington, employs sensor-embedded gloves to recognize specific hand gestures and map them to spoken words [4]. Similarly, KinTrans provides real-time sign-to-voice translation by leveraging depth sensors and skeletal tracking data [5].
Despite the notable progress achieved, these solutions typically suffer from at least one of the following limitations:

- *unidirectional functionality*
- *dependency on specialized hardware*
- *limited support for different languages and dialects*

These constraints hinder their integration into real-time applications on mobile devices, significantly limiting their accessibility and scalability in everyday communication scenarios.
While such systems may be effective for delivering static content—such as in museums or public service announcements—their applicability remains constrained in dynamic, real-world interactions.

2.2 Deep Learning Approaches

The advent of Computer Vision and the introduction of Deep Learning algorithms have shifted the scientific community's attention toward this field, leading to the development of innovative new solutions.
In particular, the creation of dedicated datasets [6, 7] and the release of open-source frameworks—such as MediaPipe [8] and OpenPose [9], which enabled pose estimation—have facilitated the emergence of new specific tasks within the domain of Sign Language Processing (SLP).
These tools allow for the extraction of anonymized, low-dimensional representations of human motion while preserving essential gestural features, making them particularly suitable for applications where privacy and computational efficiency are critical.

2.2.1 Sign Language Detection and Identification

Sign Language Detection (SLD) is a binary classification task aimed at determining whether sign activity is present in a given video or video segment. It plays a critical role in identifying regions of interest within video content.
Potential applications include the automatic tagging and categorization of sign language videos, serving as an initial step toward auto-captioning, as well as functioning as a preliminary filter for subsequent tasks such as Sign Language Recognition (SLR) and Sign Language Translation (SLT).
Borg et al. [7] proposed a multi-layer RNN architecture composed of both CNN and RNN components for SLD. Their objective was to provide a method that could be applied to automatically initialize ASLR system, rather than relying on predefined assumptions regarding the video content.
Subsequently, Moryossef et al. [10] refined this technique for real-time detection by proposing a simple human optical-flow representation for video based on pose estimation to perform a binary classification per frame. They compared various possible inputs such as full-body pose estimation, partial pose estimation collected through OpenPose [9], ultimately achieving a more efficient system suitable for interactive and real-time applications.
While detection focuses on identifying the presence of signing activity within

video content, Sign Language Identification (SLI) aims to classify the specific sign language used—such as distinguishing between Italian Sign Language (LIS), American Sign Language (ASL), and French Sign Language (LSF).

2.2.2 Sign Language Segmentation

Sign Language Segmentation (SLS) involves detecting the temporal boundaries of signs or phrases in video data, effectively dividing the content into meaningful units. Renz et al. [11] proposed an automatic sign segmentation model capable of identifying the temporal boundaries between signs in continuous sign language. Their approach demonstrated the effectiveness of coupling robust 3D-CNN representations with an iterative 1D temporal CNN refinement module to produce accurate sign boundary predictions.
The objective of their work was to provide a method with the potential to significantly reduce annotation costs, thereby facilitating the creation of domain-specific datasets and promoting further research and development in the field of Sign Language Processing (SLP).

2.2.3 Sign Language Recognition

Sign Language Recognition (SLR) involves the detection and labeling of signs within videos. This task plays a crucial role in facilitating the integration of deaf individuals into society.
Based on the acquisition process, SLR systems can be classified into sensor-based and vision-based approaches, or alternatively as isolated and continuous [12]:

- **Isolated SLR:** This approach focuses on classifying short video segments under the assumption that each segment contains a single gloss. It typically uses physically attached sensors to track head, finger, and body motion. Compared to Continuous SLR, Isolated SLR tends to yield higher performance due to its reduced complexity.

- **Continuous SLR (CSLR):** This approach aims to recognize sequences of glosses in continuous, unsegmented videos. It commonly uses multiple cameras (or webcams) to capture gestures, making it more suitable for real-time applications.

Historically, research has focused more on isolated gloss recognition due to its lower complexity. Nevertheless, both Isolated and Continuous SLR remain highly challenging tasks, as they must comprise a wide range of gestures and facial expressions.

2.2.4 Sign Language Translation

Sign Language Translation (SLT) aims to convert sign language into spoken or written language. While Sign Language Recognition (SLR) focuses on recognizing signs as their corresponding glosses, SLT involves translating these recognized glosses into natural language text that can be understood by non-signers.
Glosses represent a transcription of signs that preserve grammatical and semantic elements specific to sign languages, such as tense, word order, spatial relationships, and directional cues. In some cases, glosses also include information about sign repetition or emphasis. As such, translating from gloss to natural language requires additional processing to generate grammatically correct and semantically coherent output. SLT approaches can be divided into four common categories [13]:

- **Sign2gloss2text (S2G2T):** This approach first recognizes the sign language video as gloss annotations, which are then translated into spoken language text.
- **Sign2text (S2T):** This end-to-end approach directly generates spoken language text from sign language video, bypassing the intermediate gloss representation.
- **Sign2(gloss+text) (S2(G+T)):** A multitask approach that jointly outputs both glosses and spoken language text, often leveraging gloss annotations as auxiliary supervision.
- **Gloss2text (G2T):** This approach focuses on translating gloss sequences into spoken language text, typically used to evaluate the translation quality independent of visual recognition.

Camgoz et al. [6] demonstrated that incorporating glosses—semantic labels aligned with the structure of sign language—can significantly improve translation performance. More recent Transformer-based models have further enhance this line of research by capturing long-range dependencies and context information, thereby enabling more accurate gloss-to-text and video-to-text translation [14, 15].

2.2.5 Sign Language Production

Sign Language Production (SLP), on the other hand, involves translating spoken language to a continuous stream of sign language video. Avatar-based systems such as JASigning, SiMAX, and PAULA have focused on producing sign language from either text or speech input using 3D character animations. These systems typically rely on formal sign transcription languages, such as HamNoSys or SiGML, to generate synthetic sign sequences [16, 17].

Saunders et al. [18] proposed Progressive Transformers, a novel Transformer-based architecture capable of translating discrete spoken language into continuous 3D sign pose sequences in an end-to-end fashion. Unlike previous approaches, which primarily focused on generating concatenated isolated signs, this model aims to produce fluid, continuous sign language sequences that more closely resemble natural signing.

2.3 Commercial Solutions

Several commercial applications have been developed to support communication for deaf and hard-of-hearing users, particularly in the context of phone or video calls. Among the most widely used are Ava [19] and Pedius [20], both of which leverage Speech-to-Text (STT) and Text-to-Speech (TTS) technologies to provide real-time transcription and voice synthesis. Ava also offers integration with video conferencing platforms such as Zoom, enabling live captioning during meetings.
Despite their utility, these tools present notable limitations when compared to more sign language-centric solutions.
First, they do not support natural sign language communication, relying exclusively on written text and voice, and therefore fail to convey the rich expressiveness, spatial structure, and non-verbal cues that are intrinsic to sign language.
Moreover, their functionality often varies across operating systems (e.g., Android vs. iOS), leading to inconsistencies in the user experience. Additionally, several advanced features—such as unlimited transcriptions or meeting integrations—are gated behind subscription plans, which may limit accessibility for some users.
In contrast, the solution presented in this work focuses on native support for sign language, offering a more inclusive and expressive communication channel without relying solely on text-based interaction.

Chapter 3

Background

This chapter provides a brief overview of the technologies used in the project associated with this thesis. It serves as a general introduction to the tools, frameworks and technologies adopted, in order to facilitate the understanding of the subsequent chapters.

3.1 Development Tools & Versioning

This category includes tools and standards that supported the overall development workflow, version control, and system configuration. These technologies facilitated collaborative coding, standardized data exchange formats, and ensured the reproducibility and deployability of the application environment across different systems. Each tool played a key role in the project's infrastructure and daily development lifecycle.

3.1.1 Git

Git is a distributed version control system widely used in software development to manage and track changes in source code. Its core functionalities include branching, merging, and support for non-linear development workflows, which allow developers to organize and maintain different versions of a code base efficiently.
GitHub is a cloud-based platform that provides hosting for Git repositories and extends its capabilities through features such as commit history visualization, issue tracking, and integration with continuous deployment tools.
In this project, Git and GitHub were employed to version and manage the source code of the various system components, allowing for a structured development process, clear documentation of progress over time, and safe experimentation through branch-based workflows.

3.1.2 JSON

JavaScript Object Notation (JSON) is a lightweight data-interchange format that is easy for humans to read and write, and straightforward for machines to parse and generate. It is based on a subset of the JavaScript programming language, and its syntax consists of key-value pairs and ordered lists, making it highly intuitive and widely compatible across programming environments.
Due to its simplicity, readability, and language independence, JSON has become the most widely adopted standard for data exchange in web applications, particularly in client-server communication through RESTful APIs. Unlike XML, JSON offers a more compact representation, reducing payload size and improving transmission efficiency over the network.
In this project, JSON was adopted as the primary format for structuring and transmitting data between the mobile client and the web server. Its hierarchical structure allowed for a clear and organized representation of user inputs and system responses, facilitating seamless integration between front-end and back-end components.

3.1.3 Swagger (OpenAPI)

Swagger—now part of the OpenAPI Specification—is a widely adopted framework for designing, documenting, and consuming RESTful APIs. The OpenAPI Specification provides a standardized, language-agnostic format for describing API endpoints, request and response schemas, authentication methods, and other relevant metadata. This enables both humans and machines to understand the capabilities of a web service without requiring access to source code or external documentation.
Tools in the Swagger ecosystem, such as Swagger UI and Swagger Editor, facilitate the generation of interactive and machine-readable API documentation, improving development efficiency and easing client-side integration.
In this project, the OpenAPI Specification was used to define and document the RESTful APIs exposed by the web server. This approach enabled the automatic generation of comprehensive and up-to-date API documentation, which may greatly support future developments and maintenance efforts on the application.

3.1.4 Docker

Docker is an open-source platform designed to automate the deployment, scaling, and management of applications through containerization. Containers encapsulate an application along with its dependencies, configurations, and environment, ensuring consistency across different systems and stages of development. This makes Docker particularly valuable in facilitating reproducibility, portability, and isolation

of software components.
Docker Compose is a complementary tool that enables the definition and orchestration of multi-container Docker applications using a declarative YAML configuration file. It simplifies the management of services, networks, and volumes, allowing developers to define complex application topologies and manage them with simple commands.
In this project, Docker was employed to containerize the backend components, ensuring environmental consistency and easing deployment across various development and testing environments. Docker Compose was used to orchestrate the different backend services, enabling coordinated startup, configuration, and interconnection of components with minimal manual intervention.

3.2 Frontend Technologies

The frontend technologies were responsible for implementing the user interface and enabling interaction between end users and the system. The tools in this category were used to design and build a responsive, cross-platform application capable of delivering a seamless user experience.

3.2.1 JavaScript

JavaScript is a high-level, interpreted programming language widely adopted in web and mobile development due to its flexibility, event-driven architecture, and broad compatibility across platforms. It supports functional and object-oriented programming paradigms and serves as the foundation for various frameworks dedicated to building user interfaces in modern applications, such as React and React Native.
TypeScript is a statically typed superset of JavaScript that compiles to plain JavaScript. It introduces type annotations, interfaces, and other advanced features that enhance code reliability, readability, and maintainability—especially in large-scale or long-term software projects.
Both languages are commonly utilized in modern development environments, with the selection between them typically depending on project-specific requirements and developer preferences.

3.2.2 React Native (Expo)

React Native is an open-source framework that enables the development of cross-platform mobile applications using JavaScript or TypeScript, following the React programming paradigm. It allows developers to define user interfaces through reusable components and to access native functionalities via a unified bridge

architecture. React Native inherits key principles from React—such as declarative syntax and component-based design—making it particularly effective for building responsive and maintainable mobile interfaces.
Expo is a framework built on top of React Native that simplifies the development process by providing a comprehensive toolkit and a set of pre-configured libraries, enabling the creation of mobile applications without requiring direct interaction with native code. It abstracts many of the complexities of mobile development, offering features such as live reloading, over-the-air updates and simplified configuration. Expo supports both JavaScript and TypeScript, giving developers the flexibility to adopt static typing where needed for better code quality and maintainability.
In this project, the mobile application was developed directly using Expo, taking advantage of its streamlined workflow, built-in functionalities, and full compatibility with the React Native ecosystem.

3.3 Backend & API Development

This category includes the technologies employed for implementing the backend logic, managing data persistence, and exposing functionalities through RESTful APIs. The components in this section enabled the system to process requests, handle user authentication and authorization, interact with databases, and manage asynchronous tasks. These tools formed the core of the server-side architecture.

3.3.1 Python

Python is a high-level, interpreted programming language known for its readability, extensive standard library, and strong support for multiple programming paradigms. Over the past decade, it has become the dominant language for scientific computing, data analysis, artificial intelligence, and machine learning, due to its vast ecosystem of libraries and frameworks specifically designed for these domains.
In the field of Artificial Intelligence (AI) and Machine Learning (ML), Python provides seamless integration with powerful tools such as TensorFlow, PyTorch, scikit-learn, and NumPy, allowing for efficient model development, training, and deployment. Its simplicity and expressive syntax enable rapid prototyping and facilitate collaboration among researchers and developers.
In this project, Python was used to implement the core backend components responsible for translating sign language into audio and vice versa. TensorFlow was employed for gesture recognition, leveraging Python's mature ecosystem to accelerate the integration of AI-driven functionalities within the system.

3.3.2 Node.js (Express)

Node.js is a runtime environment that allows the execution of JavaScript code outside the browser, built on the V8 JavaScript engine. It is particularly well-suited for building scalable, event-driven, and non-blocking I/O applications, making it a popular choice for server-side development. Its single-threaded architecture combined with asynchronous programming capabilities enables high-performance handling of concurrent operations.
Express is a minimalist and flexible web application framework for Node.js that simplifies the development of web servers and APIs. It offers a streamlined interface for routing, middleware integration, and HTTP request management.
In this project, Node.js was used in conjunction with Express to develop a web server responsible for exposing RESTful APIs consumed by the mobile client, enabling structured and efficient access to the system's functionalities.

3.3.3 Sequelize

Sequelize is a promise-based Object-Relational Mapping (ORM) library for Node.js that provides an abstraction layer over relational databases such as PostgreSQL, MySQL, and SQLite. An ORM is a programming technique that allows developers to interact with a database using object-oriented paradigms rather than writing raw SQL queries. It maps database tables to classes and records to objects, enabling more intuitive and maintainable data manipulation within application code.
The use of an ORM like Sequelize offers several advantages, including automated query generation, schema synchronization, model validation, and support for complex relationships between entities. It also enhances code readability, reduces the chances of SQL injection vulnerabilities, and promotes a more consistent development workflow.
In this project, Sequelize was employed to manage the application's interaction with a relational PostgreSQL database, facilitating the definition of data models and enabling efficient implementation of persistence logic in a structured and type-safe manner.

3.3.4 PostgreSQL

PostgreSQL is a powerful, open-source relational database management system known for its reliability, robustness, and support for advanced data types and operations. It adheres closely to SQL standards while also offering a wide range of modern features, such as full-text search, JSON support, and extensibility through user-defined functions and data types.
Designed with an emphasis on data integrity and transactional consistency, PostgreSQL is widely adopted in applications that require complex queries, strong

ACID compliance, and scalable performance. Its architecture supports concurrent access and efficient indexing strategies, making it suitable for both small-scale and enterprise-level deployments.
In this project, PostgreSQL was used as the primary data store to persist user data and application-specific information. Its structured schema, combined with strong query capabilities, ensured reliable data management and seamless integration with the Sequelize ORM for efficient database operations.

3.3.5 JWT

JSON Web Token (JWT) is an open standard that defines a compact and self-contained method for securely transmitting information between parties as a JSON object. It is commonly used for authentication and authorization in modern web and mobile applications due to its stateless nature and ease of integration with HTTP-based protocols.
Unlike traditional session-based authentication, which relies on server-side session storage, JWT enables stateless authentication by embedding user identity and authorization claims directly within the token. This makes it particularly suitable for mobile applications, where maintaining persistent sessions on the client side is not feasible.
In this project, JWT was adopted to implement the authentication mechanism for the mobile application. Upon successful login, the server issues an access token for authorizing API requests, along with a refresh token that enables the client to obtain new access tokens without requiring repeated logins. This mechanism ensures secure and seamless user sessions, while also supporting scalability and minimizing server-side storage dependencies.

3.3.6 Redis

Redis is an open-source, in-memory data structure store that supports a wide range of data types, including strings, hashes, lists, sets, and more. It is widely used as a caching layer, message broker, and key-value store due to its extremely low latency and high throughput, making it suitable for performance-critical applications.
One of Redis's key features is its support for automatic expiration of keys through Time-to-Live (TTL) settings. This allows developers to associate a limited lifespan with specific entries, enabling efficient management of temporary data such as session tokens, cache entries, and rate-limiting counters.
In this project, Redis was employed to implement a JWT token blacklist mechanism. When a user logs out or when a token needs to be invalidated, the corresponding token is stored in Redis with an expiration time matching its validity period. This approach leverages Redis's speed and TTL capabilities to efficiently manage

invalid tokens without impacting overall system performance or requiring persistent storage.

3.3.7 Nodemailer

Nodemailer is a module for Node.js applications that enables the programmatic sending of emails through various transport mechanisms, including SMTP, OAuth2, and third-party services. It provides a high-level API for composing and dispatching emails, with support for attachments, HTML content, and templating, making it a widely adopted tool for integrating email functionality into server-side applications. In this project, Nodemailer was employed in combination with Gmail's SMTP service to implement the automated delivery of transactional emails from the web server. These service messages included notifications such as successful registration confirmations and account deletion acknowledgments. The use of Nodemailer enabled seamless integration of email-based user communication while ensuring a reliable and configurable delivery mechanism.

3.3.8 Supabase

Supabase is an open-source backend-as-a-service platform that provides a suite of scalable services commonly required in modern applications, including authentication, real-time databases, and cloud storage. Built on top of PostgreSQL, it aims to offer a developer-friendly alternative to proprietary backend solutions, while maintaining flexibility and openness.
One of Supabase's core features is its cloud storage system, which allows secure and organized management of files such as images, documents, and media assets. It includes access control policies, public and signed URLs, and integration with authentication services to manage file-level permissions.
In this project, Supabase was used specifically for cloud storage of user profile images. By leveraging its secure and scalable storage service, the system was able to delegate file management to a dedicated cloud infrastructure, ensuring persistent availability, ease of retrieval, and integration with user profile management.

3.4 Real-time Communication & Notifications

Real-time communication and notification mechanisms were critical for enabling interactive features and timely user updates. This category includes technologies that facilitated bidirectional communication between clients and the server, push notifications, and media stream handling. Their integration allowed the system to support synchronous interactions and real-time data delivery, which were essential to the project's requirements.

3.4.1 WebSocket

WebSocket is a communication protocol that enables full-duplex, bidirectional communication channels over a single, long-lived TCP connection. Unlike traditional HTTP, which follows a request-response model, WebSocket allows servers and clients to exchange data in real time without the overhead of repeated handshakes, making it ideal for latency-sensitive applications such as live chats, online gaming, and video conferencing.
In this project, WebSocket was used by the web server to manage the signaling phase required for initiating video calls, in coordination with the Mediasoup library, which handled the media transmission. During this phase, signaling messages—such as transport parameters and connection metadata—were exchanged between the mobile client and the server to establish and configure the media transport channels. WebSocket provided the low-latency, persistent communication needed to support real-time call setup and coordination.

3.4.2 FCM

Firebase Cloud Messaging (FCM) is a cross-platform messaging solution provided by Google that enables reliable delivery of messages and push notifications to client applications. It supports both upstream and downstream messaging and is widely used in mobile and web development to engage users and facilitate real-time communication.
FCM offers features such as message targeting based on device tokens, user segments, or topics, as well as support for both notification and data payloads.
In this project, FCM was used to deliver push notifications to the mobile application, allowing the system to inform users of relevant events such as incoming video calls. Its reliability, scalability, and seamless integration with mobile platforms—in combination with WebSocket—made it a suitable choice for implementing real-time user notifications.

3.4.3 Mediasoup

Mediasoup is a modern, open-source Node.js library for building scalable and customizable real-time audio/video communication systems. It acts as a Selective Forwarding Unit (SFU), a media server architecture that receives media streams from clients and selectively forwards them to other participants without mixing them. This approach ensures low latency and efficient bandwidth usage, particularly in group call scenarios.
Mediasoup offers a high degree of flexibility by allowing developers to intercept, process, and re-inject media streams, enabling advanced features such as real-time analysis, filtering, or recording. Its design abstracts much of the complexity of

WebRTC while still providing full control over transport configuration and media flow.
In this project, Mediasoup was chosen for its simplicity and extensibility, allowing the implementation of a robust media server without the need to develop one from scratch. Its SFU architecture and stream manipulation capabilities were instrumental in enabling efficient and controllable video call management within the system.

3.5 Media Processing

This section focuses on the technologies used for audio and video processing within the system. Media processing tools were essential for encoding, decoding, and manipulating multimedia content, allowing for its transformation, compression, and transmission. These capabilities were fundamental for managing the media-intensive aspects of the project, particularly in the context of real-time communication.

3.5.1 FFmpeg

FFmpeg is a powerful, open-source multimedia framework used for processing audio and video data. It provides a rich set of tools for encoding, decoding, transcoding, muxing, demuxing, streaming, and filtering media in a wide variety of formats. Due to its flexibility and command-line interface, FFmpeg is widely adopted in both production environments and research applications involving media processing.
In this project, FFmpeg served as an intermediary component between the web server and Python-based microservices responsible for media translation. Specifically, it was used to capture RTP streams from Mediasoup, encode them into raw audio or video formats, and forward them to the Python-based translation engine. After processing, FFmpeg was again employed to re-encode the media into RTP streams and inject them back into a Mediasoup transport. This setup enabled the seamless integration of AI-based translation capabilities within live media flows, leveraging FFmpeg's efficiency and versatility in handling real-time multimedia data.

Chapter 4

Methodology

This chapter presents a detailed description of the design and development process of the proposed solution. It begins with an overview of the overall system architecture, followed by an in-depth analysis of each individual system component.

4.1 System Architecture

The system designed and developed in my project is composed of two distinct and interconnected modules. The overall system architecture is illustrated in Figure 4.1. The first module is the backend system, which was implemented using a microservices architecture. As shown in Figure 4.1, the core service is a Node.js-based web server responsible for handling user authentication and managing user data, which is stored in a relational PostgreSQL database. Additionally, a Redis in-memory data store is employed to manage the blacklisting of invalidated JWT tokens, ensuring secure and efficient session control. The mobile client and web server communicate by exchanging data in JSON format through RESTful APIs exposed by the backend.
Moreover, the web server is also responsible for managing video calls, handling both the signaling process and media stream routing. It communicates over TCP socket with two Python-base microservices, which execute AI translation algorithm to provide sign language translation features integrated in the video calls.
The second module is a mobile application designed to enable both deaf and hearing users to engage in video-based communication, helping to bridge the communication gap between them. In addition to video calls, the application also offers several auxiliary features, which will be described in more detail in the following sections. For this project, the application was developed to run on Android devices. However, the adopted framework allows for straightforward extension to iOS platforms, enabling future cross-platform development. In addition, particular attention was

paid to ensuring that the design of the mobile application prioritized accessibility, ease of use, and compliance with privacy regulations.

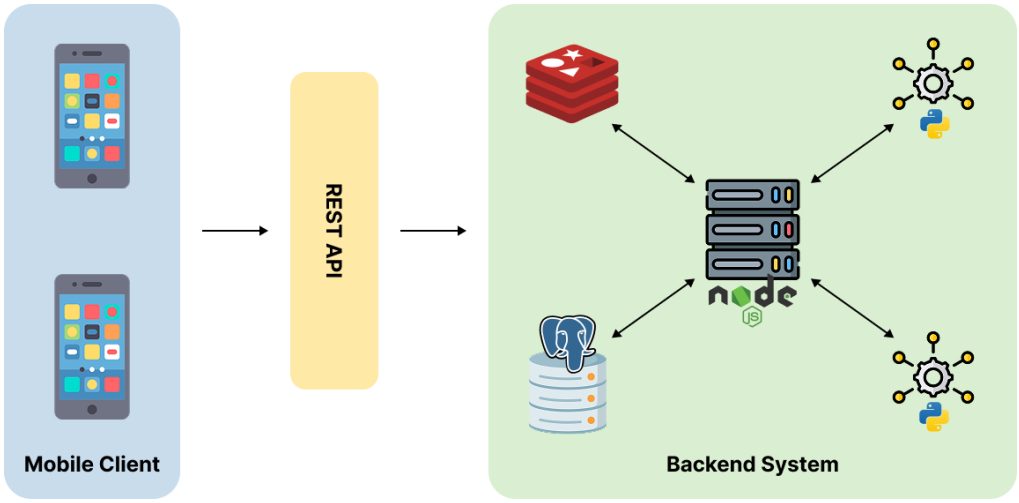


Figure 4.1: System Architecture.

4.2 Mobile Application

This project was conceived with the goal of bridging the communication gap between deaf and hearing individuals, enabling them to communicate through real-time video interactions supported by sign language translation features. Accordingly, this scenario will be adopted as the reference context throughout the chapter. Nevertheless, the long-term objective is to provide an initial example of a practical application that could be extended in future work to other sensitive domains, such as education and healthcare.

4.2.1 Functional Requirements

Having established the reference scenario, the functional requirements that the application must satisfy in order to provide the aforementioned features are as follows:

1. *Authentication:* Users must be able to register on the platform by providing a valid email address and a strong password. These credentials can then be used to authenticate the user during the login process.
2. *Credential Management:* After registration, users can change their password by providing a new one that meets the platform's security requirements.

Additionally, in case of a lost password, users can recover their credentials through an OTP-based verification process sent to their registered email address.

3. *Profile Management:* Users can edit their personal profile, including updating their personal information, email address, and phone number. Moreover, they must be able to permanently delete their account if it is no longer needed.

4. *Preferences Configuration:* Users must be able to configure their preferences regarding accessibility features, specifically the real-time sign language translation service, as well as the activation of push notifications. These notifications are used to alert users of incoming or missed video calls, even when the application is running in the background or has been terminated.

5. *Contacts Management:* Users must be able to create, edit, and delete their contacts, as well as view their associated personal information. All local changes must be continuously synchronized with the remote backend to ensure a consistent and seamless user experience across multiple devices.

6. *Video Call Management:* The application must support real-time video-based communication, with optional sign language translation features depending on the user's personal preferences. Additionally, users must be able to manage their recent call history by deleting entries that are no longer needed.

7. *Multi-Device Support:* The application must support the use of a single account across multiple devices. User data and recent activity (e.g., contacts, call history, and notifications) must be synchronized in real time through the backend infrastructure. This ensures a coherent and uninterrupted user experience regardless of the device being used.

4.2.2 Actors and Use Cases

Having introduced the reference scenario and the functional requirements of the application, it is now possible to identify the system's main actors and the roles they assume within it. In particular, the end users who interact with the mobile application are as follows:

- *Hearing user*
- *Deaf user*

Both users assume the same role within the application, differing only in the way they communicate: the deaf user relies on sign language, while the hearing user communicates through voice. Nevertheless, the features provided by the application

ensure a natural and seamless interaction between them.
The analysis of the application's functional requirements led to the identification of the fundamental actions needed to satisfy them. The diagram shown in Figure 4.2 illustrates the classification of the use cases along with their interrelationships. The principal use cases identified are:

- *Registration*
- *Login*
- *Password Recovery*
- *Profile Visualization*
- *Profile Editing*
- *Password Change*
- *Preferences Management*
- *Contacts Management*
- *Calls Management*
- *Video Call Initiation*
- *Incoming Call Response*
- *Logout*
- *Account Deletion*

These use cases correspond to the key functionalities of the application. They will be discussed in more detail in the following paragraphs. In particular, each use case will be described alongside graphical examples, illustrating what the user must do and how to interact with the application in order to perform the corresponding action.

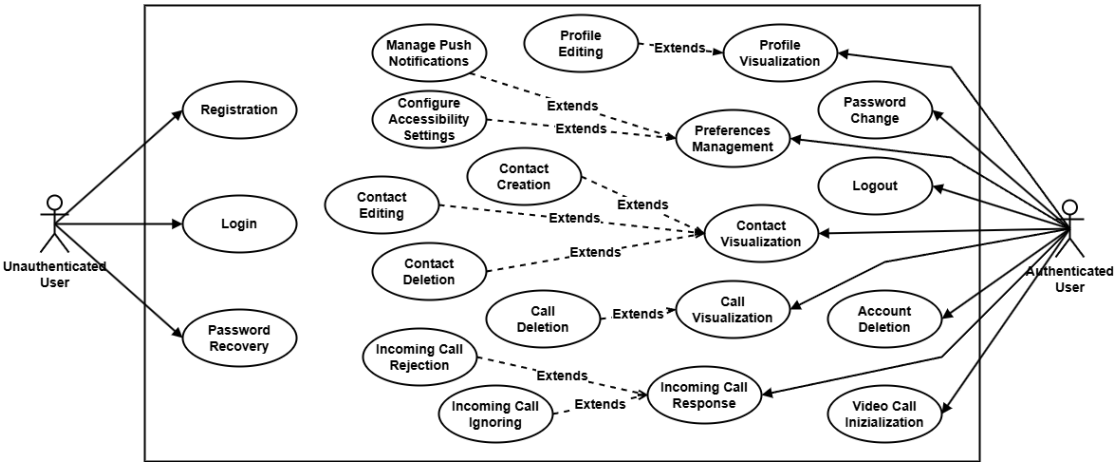


Figure 4.2: Use Cases Diagram.

4.2.3 User Interface Design

In relation to the identified use cases and functional requirements, it was also necessary during the design phase to define the overall structure of the application and to organize the information layout across the various screens, ensuring users could access all key functionalities intuitively.
In particular, this process was carried out using Figma, one of the most widely adopted platforms for User Interface (UI) and User Experience (UX) design. This tool enabled the creation of mockups for each application screen and allowed for full prototyping, making it possible to analyze the navigation flow in detail.
The diagram shown in Figure 4.3 illustrates the main application routes and the navigation flow between them.
The UI and navigation design were guided by established best practices aimed at creating a clean, intuitive interface that supports a smooth and natural user experience. Regarding the implementation, the navigation system was developed using the Expo Router library, part of the Expo framework.
Expo Router is a file-based routing solution for React Native applications, where each file within the `app` directory corresponds to a specific route. Its adoption enabled easier navigation management and ensured a cleaner and more organized project structure.
The following paragraphs provide an in-depth overview of the main application screens, in direct relation to the use cases introduced previously.

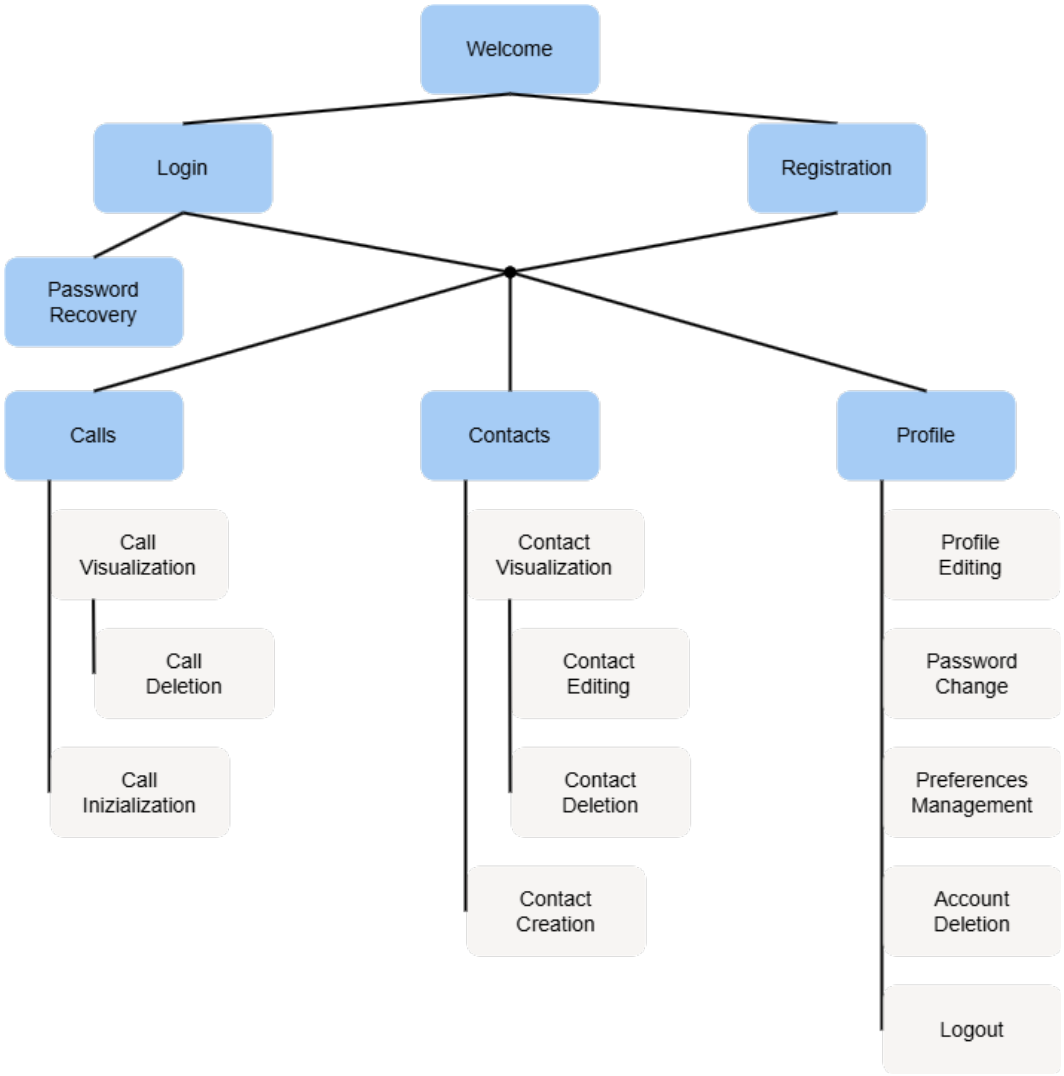
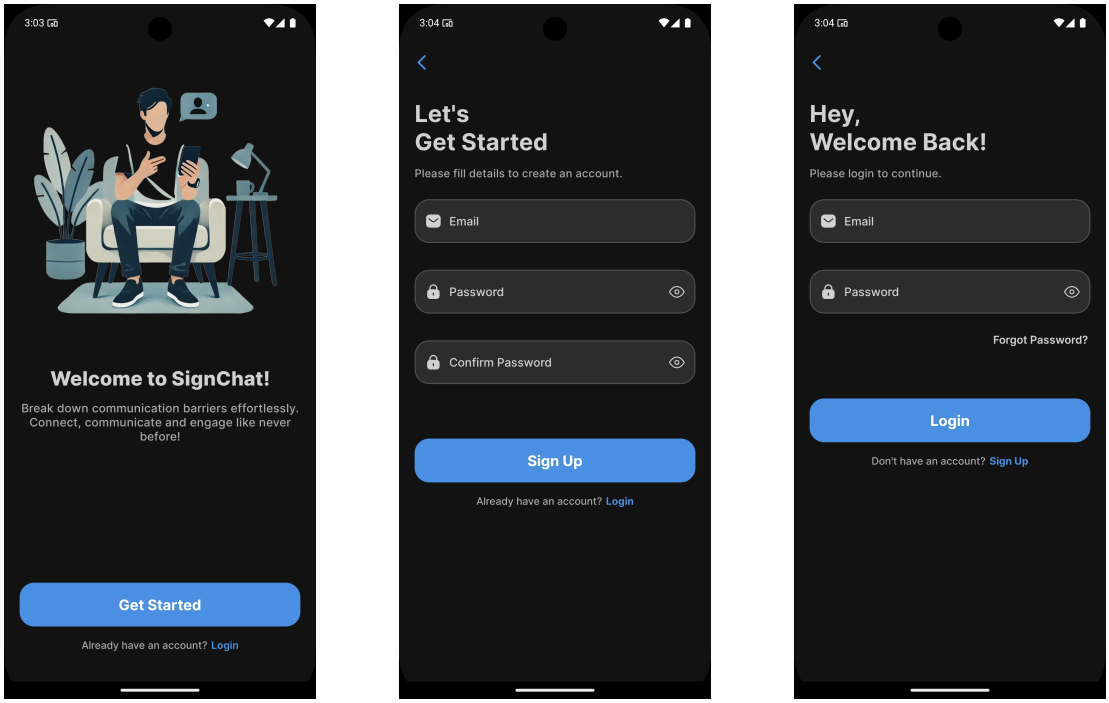


Figure 4.3: Application Structure.

Registration and Authentication

The entry point of the application is the *Welcome Screen* (Figure 4.4a), which provides a brief overview of application's features and presents two buttons allowing users to navigate either to the *Registration Screen* (Figure 4.4b) or the *Login Screen* (Figure 4.4c). On the *Registration Screen*, users can create a new account by entering a valid email address and a strong password. Conversely, the *Login Screen* allows users to authenticate by submitting previously registered credentials.
Both screens include input validation with contextual error messages to guide users in correcting invalid or incomplete data. Additionally, the password fields feature a

toggle button that allows users to show or hide the entered password, enhancing both usability and security.

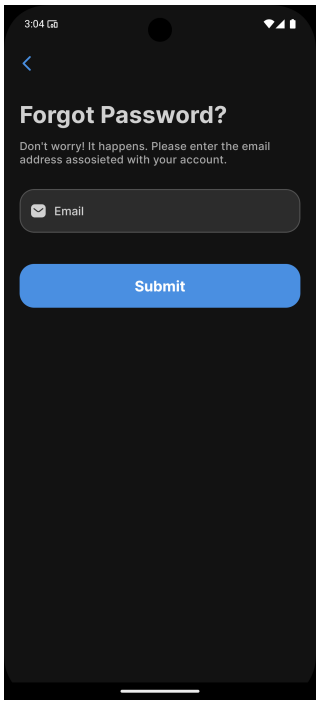


(a) Welcome **(b)** Registration **(c)** Login

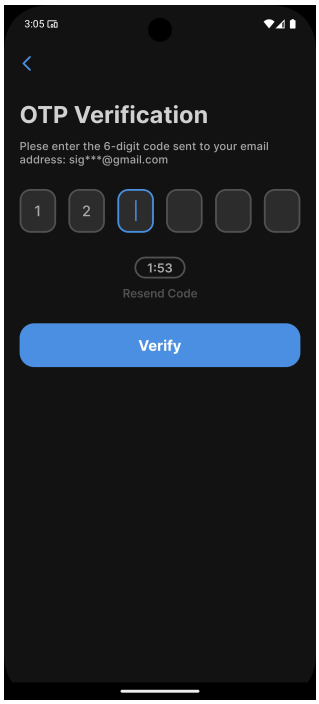
Figure 4.4: Registration and Authentication Screens.

Password Recovery

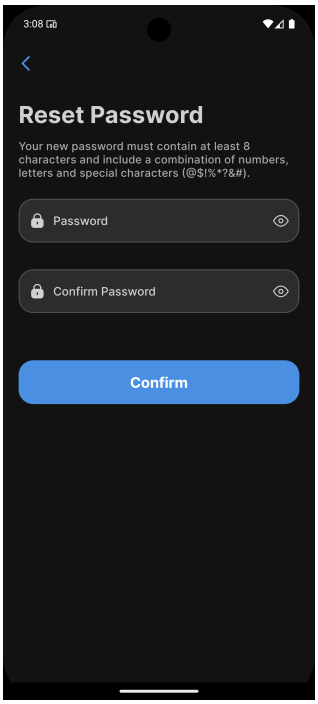
The *Login Screen* also includes a *Forgot Password* option, which, when pressed, redirects the user to the password recovery system. This allows users to reset their password in case of loss, thereby enhancing both the security and accessibility of their account. The navigation flow of password recovery system is shown in Figure 4.5. The process begins by prompting the user to enter the email address associated with their account. After performing the necessary checks, the backend verifies whether the email address is registered and, if so, sends an email containing a 6-digit OTP code. The user must enter this code to confirm ownership of the email address. Upon successful verification, the user is prompted to enter a new strong password. Once the entire process is completed, the application displays a confirmation message and provides a button that redirects the user to the *Login Screen*, where they can authenticate using their new credentials.



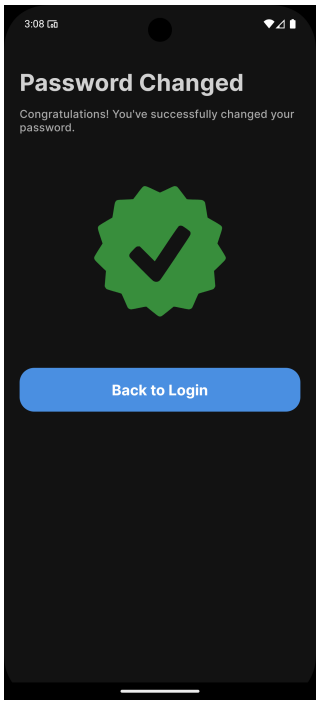
(a) Forgot Password



(b) OTP Verification



(c) Reset Password

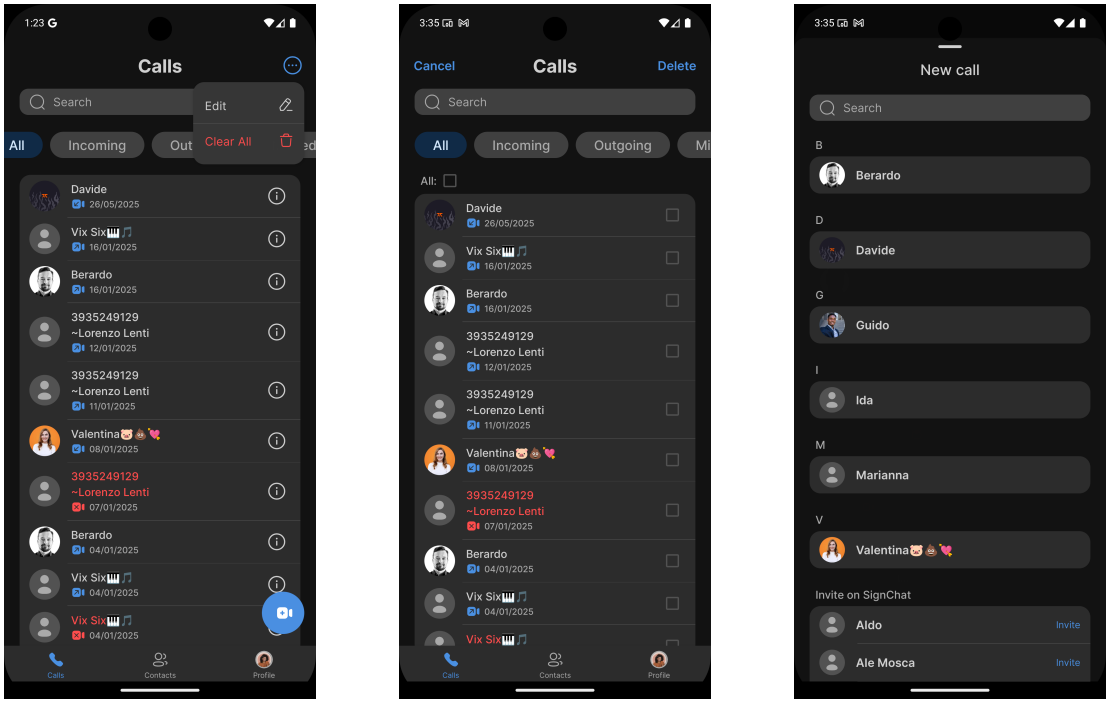


(d) Password Changed

Figure 4.5: Password Recovery Screens.

Calls Management

After authentication, the application adopts a tab-based structure, as shown in Figure 4.3. The first tab corresponds to the *Calls Screen* (Figure 4.6a), which displays the history of recent calls. In this screen, calls can be filtered by category—for example, incoming, outgoing, or missed—as well as through a search bar that allows users to filter calls by phone number or contact name.
Users can also personalize this list by clicking the options button located in the top-right corner, a menu appears allowing them to either clear the entire call history or selectively delete individual entries, as illustrated in Figure 4.6b.
Additionally, a Floating Action Button (FAB) positioned in the bottom-right corner opens a bottom sheet (Figure 4.6c) when clicked, allowing the user to select a contact to call from a list. This list can also be filtered using a search bar by phone number or contact name.



(a) Calls **(b)** Edit Calls **(c)** New Call

Figure 4.6: Calls Management Screens.

Moreover, for each call in the history list, an info icon is provided that navigates the user to the *Call Info Screen* (Figure 4.7), which summarizes all relevant information about the selected call. This screen also allows the user to initiate a new call with that contact, navigate to a dedicated screen containing the contact's

details, or remove the call from the history list.

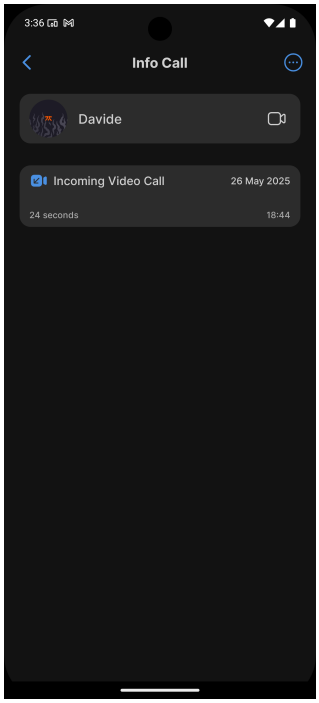
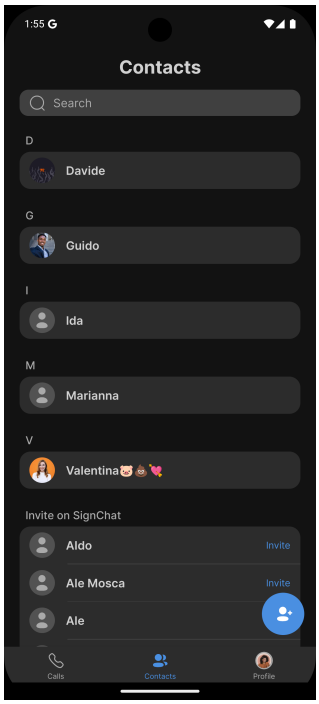


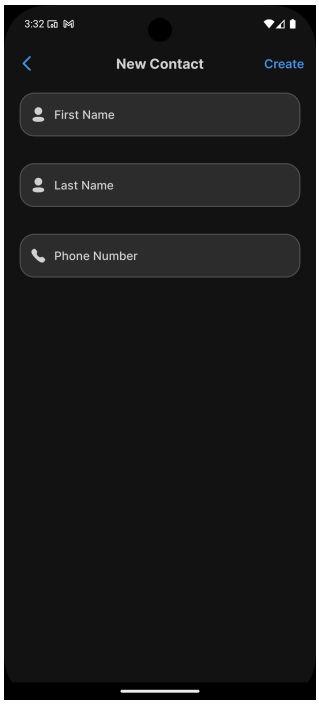
Figure 4.7: Info Call Screen.

Contacts Management

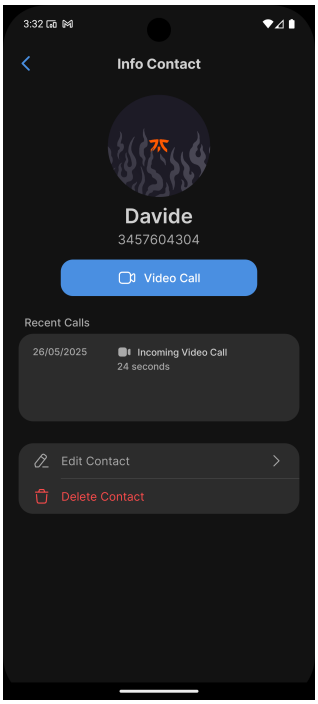
The second tab corresponds to the *Contacts Screen* (Figure 4.8a), which displays a list of contacts grouped alphabetically. A search bar allows users to filter contacts by phone number or name. Contacts who are not registered on the platform are grouped separately and can be easily invited with a single tap, which triggers the sending of a download link to the corresponding recipient.
Additionally, a FAB positioned in the bottom-right corner allows users to navigate to a dedicated screen for creating a new contact. The contact's information is then saved both locally and remotely. By clicking on a specific contact, users are redirected to the *Contact Info Screen* (Figure 4.8c), which displays all personal information related to the selected contact, along with a list of recent calls with them. Moreover, this screen allows users to easily initiate a call with the contact, edit the contact's information (Figure 4.8d), or delete the contact both locally and remotely.



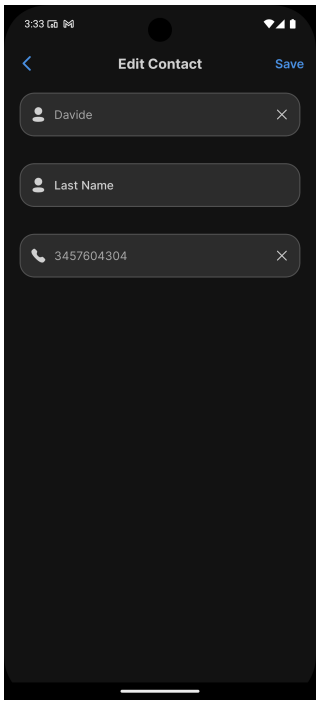
(a) Contacts



(b) New Contact



(c) Contact Info

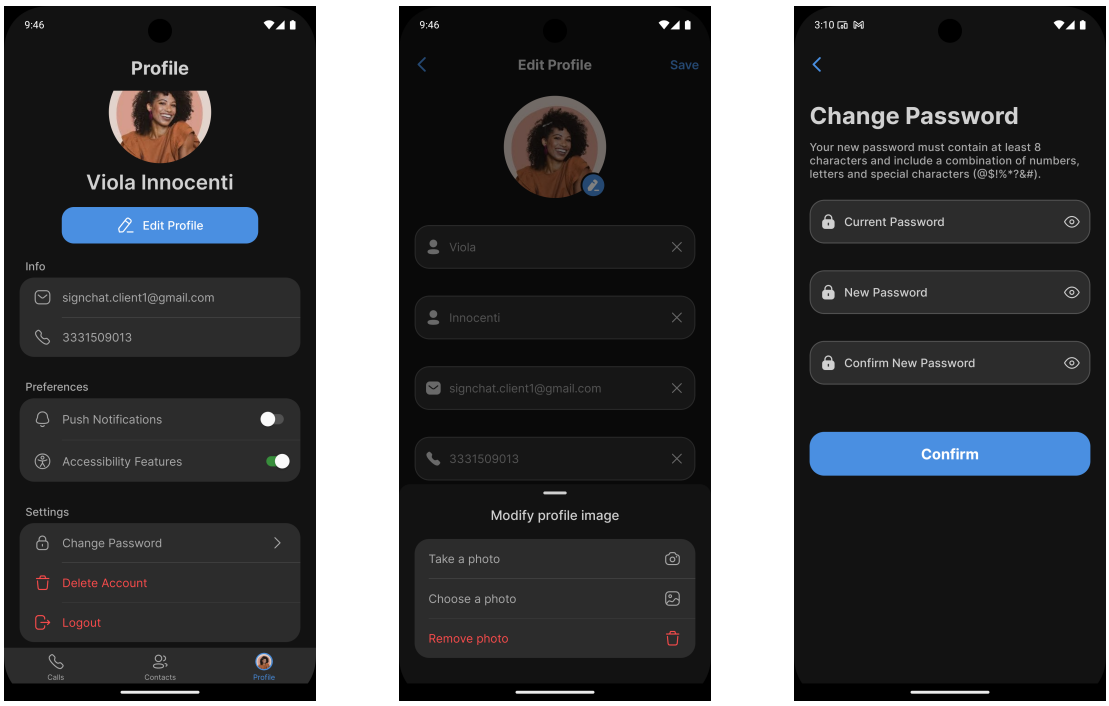


(d) Edit Contact

Figure 4.8: Contacts Management Screens.

Profile and Preferences Management

The third and final tab corresponds to the *Profile Screen* (Figure 4.9a), which displays the user's personal information, including full name, email, phone number, and profile image. By clicking the corresponding edit button, the user is redirected to a dedicated screen where they can modify their personal information (Figure 4.9b), including the profile image. In particular, an edit icon on the profile image opens a bottom sheet that allows users to choose whether to take a new photo using the camera, upload one from the gallery, or remove the existing image.
Additionally, the *Profile Screen* allows users to configure their preferences. Specifically, they can enable or disable push notifications—which are used to alert them of incoming or missed calls—as well as accessibility features, namely real-time sign language translation integrated into video calls.
Finally, users can change their password by accessing the appropriate screen (Figure 4.9c), where they must enter their current password along with a new, strong one. They can also choose to log out or delete their account entirely.

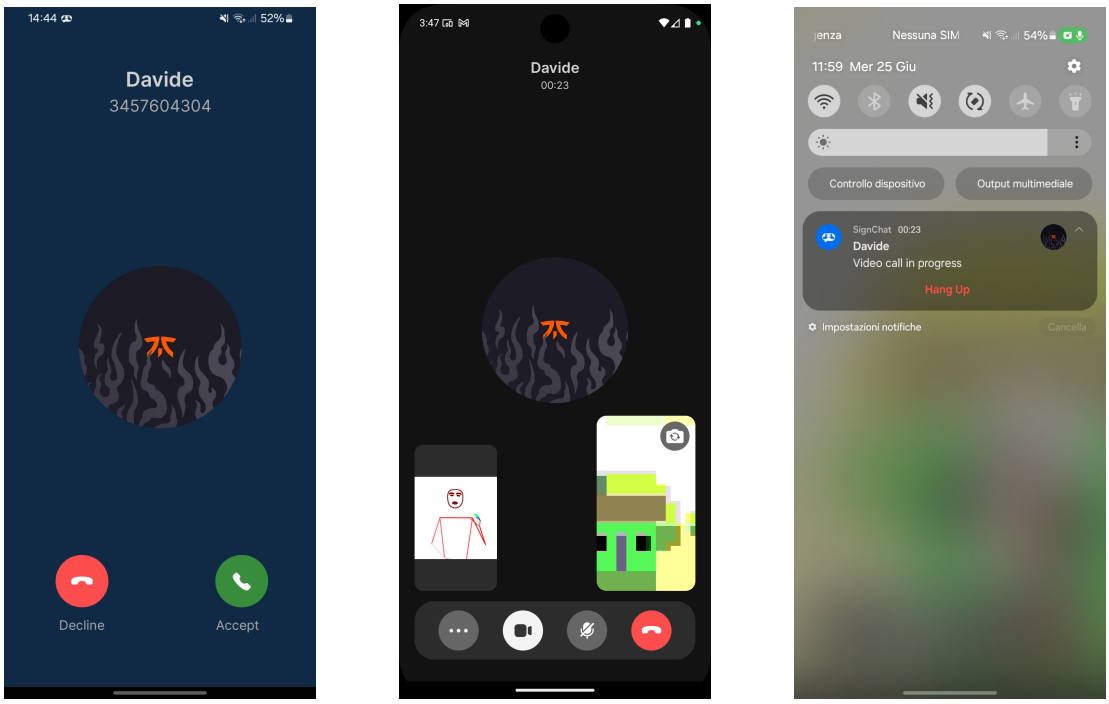


(a) Profile **(b)** Edit Profile **(c)** Password Change

Figure 4.9: Profile and Preferences Management Screens.

Incoming Call Response

When users receive an incoming call, the backend system sends a push notification to the target user, alerting them regardless of whether the application is in the background or has been terminated. The notification displays relevant information about the incoming call, such as the caller's name and profile image. Users can easily accept or decline the call directly from the notification, or tap it to open the application, which redirects them to the *Incoming Call Screen* (Figure 4.10a). This screen replicates the information and functionalities available in the incoming call notification.



(a) Incoming Call **(b)** Ongoing Call **(c)** Call Notification

Figure 4.10: Incoming Call Response Screens.

If the user chooses to accept the call, they are redirected to the *Ongoing Call Screen* (Figure 4.10b), which displays the caller's information along with a timer showing the duration of the call. This screen also includes a preview of the user's camera in the bottom-right corner, which can be dragged to any corner of the screen based on user preference. Additionally, a dedicated button allows users to switch between the front and rear cameras, while a control bar enables them to manage both camera and microphone by turning them on or off.
In the case of a video call between a deaf user and a hearing user, the real-time

translation feature provided by the backend system enables voice-to-sign and sign-to-voice conversion. Specifically, the hearing user's voice is translated into sign language and displayed to the deaf user in a dedicated box in the bottom-left corner, while the deaf user's video is processed to recognize sign language and translated into voice for the hearing user. Moreover, during a video call, an ongoing notification (Figure 4.10c) is displayed in the notification panel. It summarizes the key details of the current call and allows users to either return to the *Ongoing Call Screen* or hang up if needed.

4.3 Backend System

In order to enable the mobile application to support all the functionalities described in the previous section, it was necessary to design and develop a backend system capable of managing all relevant aspects through interaction with the mobile client, whose architecture was briefly introduced in Section 4.1.
The following paragraphs provide a more in-depth overview of each key functionality, analyzing the design decisions and implementation choices made throughout the development process.

4.3.1 Authentication

Authentication is the process of verifying the identity of a user attempting to access a resource. It also serves as a prerequisite for authorization, which determines which resources a user is permitted to access.
Since mobile applications cannot rely on traditional session-based authentication—where maintaining persistent sessions on the client side is not feasible—this project adopts the JWT standard to implement a stateless authentication mechanism. JWT embed the user's identity and authorization claims directly within a token, allowing for secure and scalable communication between the client and server. Upon successful login, the server generates and returns two tokens:

- an *access token*, which must be included in subsequent API requests to authorize the user,
- and a *refresh token*, which can be used to request a new access token once the original token expires.

This mechanism ensures a seamless and secure user experience by avoiding repeated logins. For security reasons, the access token is configured with a short expiration time, while the refresh token has a longer validity period but is used only when necessary. Importantly, only the access token is included in API requests,

while the refresh token is used exclusively to renew expired sessions.
Regarding token generation, the system uses pre-generated secret keys for both access and refresh tokens. These keys are generated securely once using the `crypto` library in JavaScript, which allows the creation of cryptographically strong random byte strings. The server then signs each token by combining a payload—typically containing the user ID and metadata such as expiration time—with the appropriate secret key. The resulting signed tokens are returned to the client.
On the client side, the mobile application uses Expo SecureStore to store both tokens. SecureStore allows sensitive data such as authentication tokens to be stored securely and in encrypted form as key-value pairs directly on the device, ensuring that them are protected even in the event of local storage access attempts.
To further enhance the security of the system, a token blacklisting mechanism was implemented. This mechanism allows the server to explicitly invalidate tokens when necessary, thereby preventing their reuse in the event of compromise or after logout. This ensures that potentially malicious actors cannot exploit previously issued tokens to access protected resources.
For this purpose, Redis was used due to its high-speed in-memory operations and native support for setting a TTL on entries. When a user logs out or a token must be invalidated, it is added to Redis with an expiration time matching its original validity, allowing for efficient and automatic management of invalid tokens without requiring persistent storage or affecting system performance.

4.3.2 Database Design

Before implementing the Node.js-based web server, it was necessary to define a strategy for the persistent storage of application data. A solution based on an ORM approach was adopted, specifically leveraging the Sequelize library for Node.js to interface with a PostgreSQL relational database. This choice enabled the definition of database tables and their relationships by leveraging object-oriented programming concepts such as classes and associations, thereby simplifying schema design and maintenance.
Moreover, the use of an ORM facilitated a more intuitive and secure interaction with the database, eliminating the need to manually write raw SQL queries. Instead, data access and manipulation could be performed using high-level JavaScript methods, improving code readability and reducing the risk of common database-related errors. This approach also supports easier migrations, validation, and synchronization of the database schema with the application logic.
Additionally, the use of Sequelize enabled the automatic inclusion of `createdAt` and `updatedAt` attributes in each table. These timestamps provide valuable metadata for tracking when records are created or modified, thereby facilitating debugging and helping identify potential anomalies or inconsistencies in the database over

time. The overall database structure is illustrated in Figure 4.11, which shows the main entities and the relationships between them. The database consists of four primary entities:

- **Users:** This table stores users' personal information, such as first name, last name, phone number, and profile image, along with their authentication credentials. For security purposes, passwords are not stored in plaintext; instead, only their hashed versions are saved.

- **Contacts:** This table stores each user's contact list. Each contact is defined by a first name, an optional last name, and a phone number. The table has two relationships with the *Users* table: one through the `ownerId` foreign key, representing the user who owns the contact, and one through the `userId` foreign key, indicating the user associated with the contact. Both are one-to-many relationships, although `userId` is optional, as a contact may not correspond to any registered user.

- **Calls:** This table logs the history of calls made between users. Each record represents a call made by a user (the owner), and for each call, two records are stored—one per participant. Each call entry includes a phone number, call type (incoming or outgoing), status (e.g., completed, missed, unanswered, rejected, or ongoing), date, and duration. The table includes two optional foreign keys: `userId`, linking to the user associated with the call, and `contactId`, linking to the associated contact. These are optional because a call may involve a user not stored in the contacts list or one who is no longer registered.

- **Tokens:** This table stores the FCM tokens used by the backend system to send push notifications. Each token is associated with a specific user, through the `ownerId` foreign key, and is defined by a combination of `deviceId` and `fcmToken`. This design supports multi-device usage, allowing users to receive notifications on all of their registered devices.

This database schema provides a robust and scalable foundation for supporting all the application's core functionalities. Its structure has been carefully designed to enable efficient data retrieval and management, facilitate synchronization across multiple devices, and ensure consistency in user interactions and communication features.

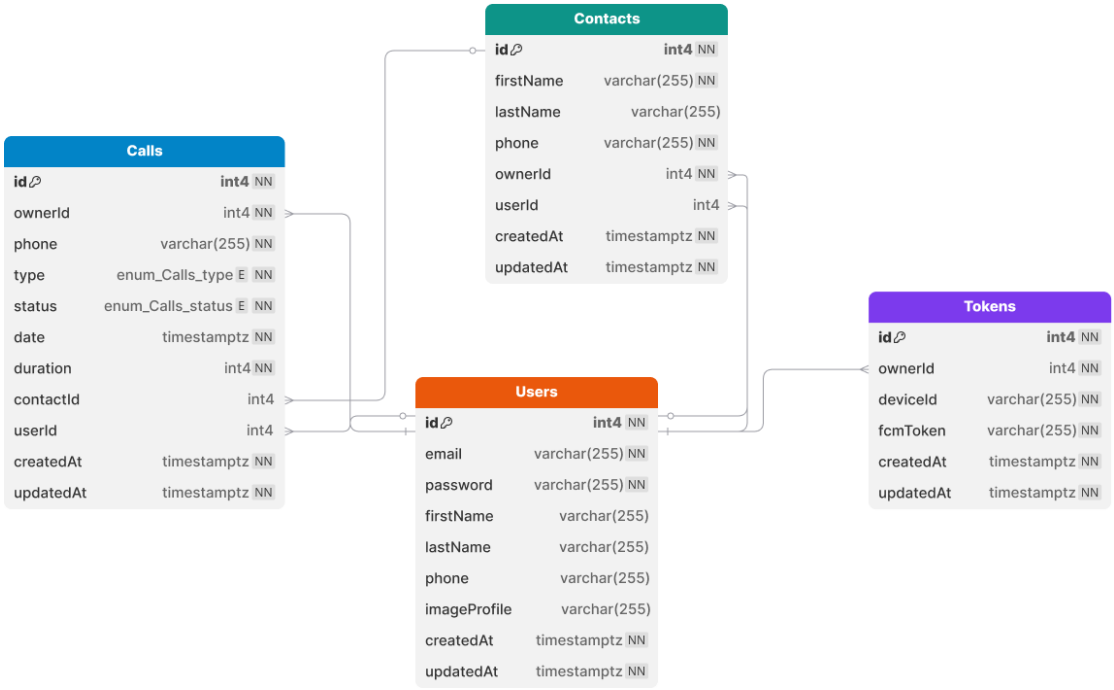


Figure 4.11: Database Structure.

4.3.3 API Design and Implementation

Although the core functionality of the application is real-time video communication, several auxiliary features have been implemented to provide a complete and seamless user experience. These features are supported through a set of RESTful APIs exposed by the Node.js-based web server.

Tables 4.1–4.2 summarize the main information regarding these APIs, including the HTTP method, endpoint, description, and whether authentication is required. The APIs are grouped by the corresponding resource, and for each resource, the necessary CRUD operations have been implemented based on the system's requirements. Furthermore, comprehensive API documentation has been generated using the Swagger/OpenAPI standard. This documentation is accessible through the`/api-docs` endpoint and provides a web interface that details each API, including the required parameters and response formats. It also enables manual testing of each individual API endpoint.

The implementation of each API follows a consistent and structured pattern. For endpoints requiring authentication, a custom middleware has been developed. This middleware verifies the presence of a valid JWT token in the `Authorization` header of the HTTP request. It checks whether the token is valid, not expired, and not blacklisted by querying the Redis store. If any of these checks fail, the server

returns a `401 Unauthorized` error response to the client. Otherwise, the request proceeds to the next step: input validation.
To validate all incoming data, the implementation relies on the `express-validator` library—a middleware for Express.js that provides a comprehensive set of tools to validate request bodies, query parameters, and URL parameters. It also supports the creation of custom validators to enforce application-specific constraints. This ensures that the data meets all expected criteria before any business logic is executed. Once validation is successfully completed, the necessary data is extracted from the HTTP request and used to perform the appropriate operations by interacting with the database. All database operations are executed within transactions. This approach ensures that, in the event of an error during execution, previous operations can be rolled back to maintain data consistency and avoid a corrupted or inconsistent state. By following this pattern, the API layer is made robust and resilient to failures, providing meaningful error messages to the client whenever issues arise and ensuring overall reliability and maintainability of the system.

Method	Endpoint	Description	Auth
POST	/auth/register	Register a new user	✗
POST	/auth/login	Log in user	✗
POST	/auth/refresh-token	Refresh user tokens	✗
POST	/auth/change-password	Change user password	✓
POST	/auth/reset-password/request	Request reset password	✗
POST	/auth/reset-password/verify-otp	Verify email OTP code	✗
POST	/auth/reset-password/confirm	Confirm reset password	✓
POST	/auth/logout	Log out user	✓
POST	/auth/verify	Verify authentication	✓

Table 4.1: Authentication APIs.

Method	Endpoint	Description	Auth
GET	`/users/{id}`	Retrieve user info by id	✓

Table 4.2: Users APIs.

Method	Endpoint	Description	Auth
GET	/calls	Retrieve all calls	✓
GET	/calls/{id}	Retrieve call by id	✓
DELETE	/calls	Delete calls by id	✓

Table 4.3: Calls APIs.

Method	Endpoint	Description	Auth
GET	/contacts	Retrieve all contacts	✓
GET	/contacts/{id}	Retrieve contact by id	✓
POST	/contacts	Create a new contact	✓
PUT	/contacts/{id}	Update a contact by id	✓
DELETE	/contacts/{id}	Delete a contact by id	✓
POST	/contacts/sync	Sync local contacts info	✓

Table 4.4: Contacts APIs.

Method	Endpoint	Description	Auth
GET	/profile	Retrieve user profile	✓
PUT	/profile	Update user profile	✓
DELETE	/profile	Delete user profile	✓
POST	/profile/image	Upload user profile image	✓
DELETE	/profile/image	Delete user profile image	✓

Table 4.5: Profile APIs.

Method	Endpoint	Description	Auth
GET	/downloads/app	Retrieve app apk file	✗

Table 4.6: Downloads APIs.

Method	Endpoint	Description	Auth
POST	/tokens	Register a new token	✓
POST	/tokens/sync	Synchronize an updated token	✓
DELETE	/tokens	Delete a registered token	✓

Table 4.7: Tokens APIs.

4.3.4 Push Notification Architecture

This section provides a brief overview of the push notification mechanism, focusing on its functionality and the implementation details of the service.
Push notifications are a form of asynchronous communication where messages are sent from the server to the client without the client having to explicitly request them. This approach differs from the traditional pull model, in which the client periodically queries the server to check for updates. The push model is significantly more efficient in terms of both network and energy consumption, especially in mobile environments, as it eliminates the need for continuous polling.
In the context of the developed application, push notifications are crucial to ensure that users are promptly informed of incoming or missed video calls—even when the app is running in the background or has been terminated. This guarantees a more responsive and seamless communication experience, which is particularly important given the real-time and accessibility-oriented nature of the service.
In particular, the implementation of the push notification service FCM, a free cloud messaging solution provided by Google. FCM enables reliable delivery of messages between servers and client applications across platforms. Figure 4.12 illustrates the architecture of the notification service, highlighting the main actors involved—namely the Node.js-based web server, the FCM infrastructure, and the mobile client—and the actions they must perform to enable the notification flow.

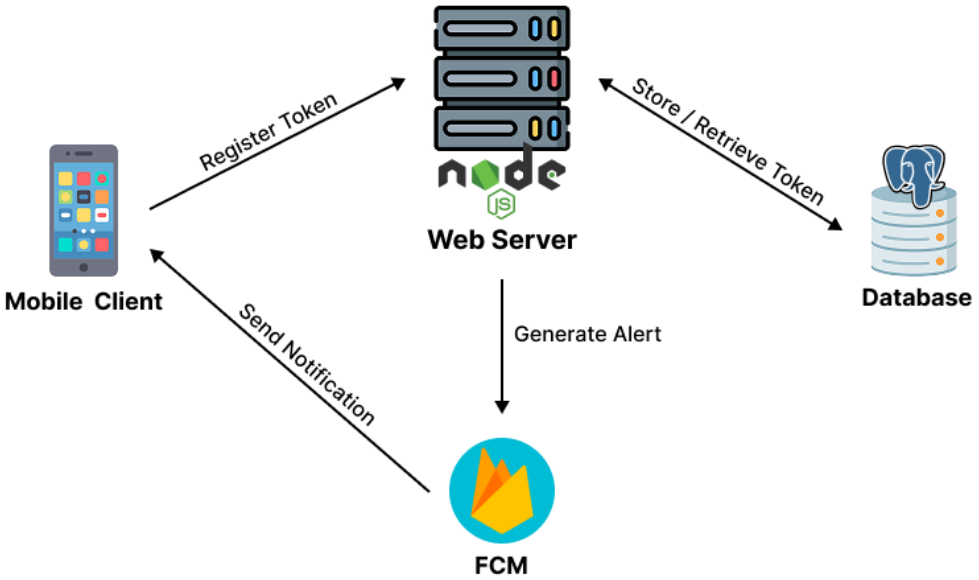


Figure 4.12: FCM Service Architecture.

As shown, the operation of the architecture is based on different steps that can be summarized as follows:

1. *Device Registration:* Before a user can receive push notifications, they must explicitly enable them within the application. This action triggers the registration of the corresponding device with the backend system to allow the delivery of push notifications via FCM.
   Specifically, once the required permissions have been granted, the mobile application registers the device with the FCM infrastructure and retrieves a unique FCM token. This token, together with a device-specific identifier, is then sent to the Node.js-based web server, which registers the device by storing the token–identifier pair in the PostgreSQL database.
   This approach allows the backend to associate multiple devices with the same user, thereby enabling full multi-device support for push notifications. As a result, users can receive incoming or missed call alerts on all their registered devices, regardless of which one was last active.
   In addition, a listener is implemented within the mobile application to monitor FCM token changes and ensure synchronization with the backend. Users may also choose to disable push notifications for a specific device at any time; in this case, the corresponding token is removed from the database.

2. *Alert Generation:* When the web server needs to send a notification to a user, it first retrieves all the tokens associated with the user's devices from the

database. Then, it generates an alert containing the relevant information and sends it to the FCM infrastructure, which is responsible for forwarding the message to the target devices using the push model.

3. *Push Notification Delivery:* Once the alert is received, the FCM backend delivers the notification to all specified devices using the tokens provided by the web server. FCM supports two types of messages: notification messages, which are automatically displayed in the notification panel by the system, and data-only messages, typically used for background data synchronization or customized handling. In this implementation, only data-only messages are used, as the `Notifee` library has been integrated into the mobile application. This library allows for fine-grained control over the appearance and behavior of notifications, as well as the definition of listeners to handle notification events when the app is in the foreground, background, or terminated state.

4.3.5 Video Call Management

Among all the features supported by the backend system, video call management constitutes its primary functionality, handled by the Node.js-based web server. The following paragraphs offers a detailed overview of this feature, highlighting the design choices and implementation strategies adopted during development.

Signaling Process

When a user initiates a video call to another user, a signaling phase is first required to correctly establish the connection between the two endpoints. This signaling process is entirely managed by the web server, which leverages both WebSocket and FCM to enable real-time communication with the users' devices. While FCM is used exclusively to deliver push notifications—for example, to alert the callee of an incoming call—WebSocket is employed for the remaining message exchange, thanks to its capability to establish a bidirectional communication channel over a single, long-lived TCP connection. This channel is used to exchange transport parameters and connection metadata necessary for configuring the media transport channels. Specifically, once the user logs in, a WebSocket connection is established with the server, creating a persistent TCP connection that enables message exchange. At this point, the server stores session-specific metadata, such as the socket identifiers associated with each of the user's devices (to support multi-device functionality), and whether accessibility features are enabled. In compliance with user privacy, this session data is not stored persistently in the database; instead, it is held in memory only for the duration of the session and discarded when the user disconnects.
Figure 4.13 illustrates the sequence of messages exchanged between the mobile client and the web server during this signaling phase.

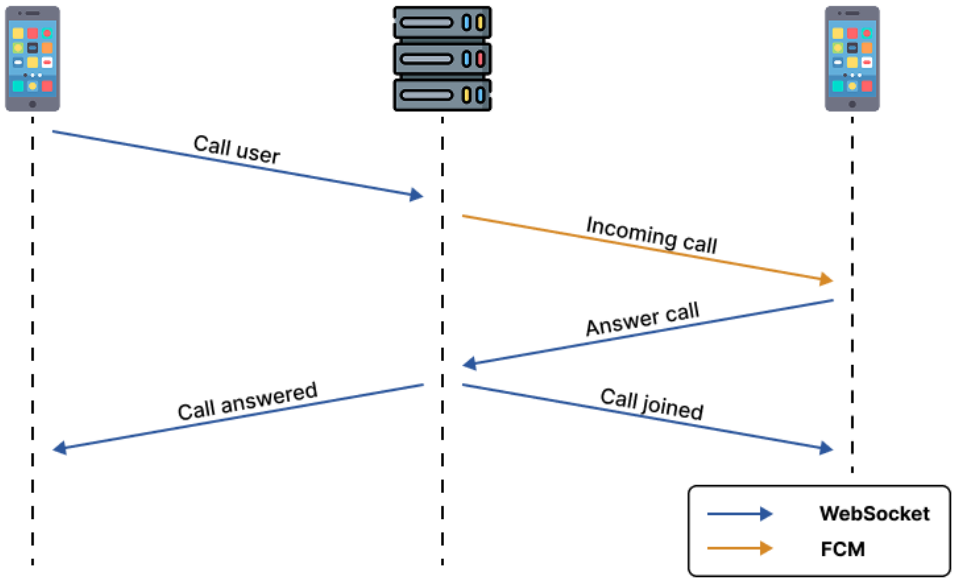


Figure 4.13: Video Call Signaling Process.

As shown, the process begins when a user requests to initiate a video call by providing the identifier of the target user. After performing several checks—such as verifying that the caller is not already engaged in another call—the server alerts the target user by sending a push notification to all of their registered devices. Upon receiving the notification, the target user can choose to ignore or decline the call, in which case the server informs the caller of the outcome. If the user accepts the call, the server creates a new call entry in the database with the status set to *ongoing*, updates the status of both users from *available* to *busy*, and notifies both participants of the successful connection. At this point, the server also shares the necessary transport parameters that will be used to establish the media transports for streaming audio and video data.

Media Stream Management

In typical real-time communication applications, media stream management is often not explicitly required, as peer-to-peer architectures handle the direct exchange of media between participants. However, in this project, such an approach was not suitable due to the need to intercept and process media streams through integrated sign language translation algorithms. Consequently, it was necessary to implement a dedicated media stream management system capable of routing and forwarding streams between the participants in a controlled manner.
While this type of functionality is usually delegated to a dedicated media server, in

order to reduce the complexity of the solution, a Node.js-integrated approach was adopted using the Mediasoup library. Mediasoup acts as a SFU, a type of media server that receives media streams from each participant and selectively forwards them to other participants without decoding and re-encoding the streams. This architecture allows for efficient media distribution, scalability, and, crucially in this case, the interception and processing of media for accessibility features.
In Mediasoup, several key components are involved in this process:

- **Worker:** A separate process that handles all low-level operations related to media transmission. For this application, a single Worker instance has been employed to manage all media operations.
- **Router:** Each Worker can host one or more Routers, which are responsible for routing media streams between participants. In this case, a single Router instance has been associated with the Worker to handle all active calls.
- **Transports:** These represent the underlying network connections used for media exchange. For each user participating in a call, two WebRTC transports are created: one sending transport (for producing media streams) and one receiving transport (for consuming media streams).
- **Producers and Consumers:** A Producer is created on a sending transport and is responsible for sending an audio or video stream, while a Consumer is created on a receiving transport to receive and process a specific media stream produced by another participant.

After the signaling phase, during which the transports are created for both participants, an additional exchange of signaling messages takes place between the clients and the server via WebSocket to finalize the setup. This includes connecting each transport to its remote counterpart and the creation of Producers and Consumers. Specifically, each client produces its own audio and video streams and consumes the audio and video streams of the other participant, ensuring a bidirectional flow of media.
Furthermore, camera and microphone control during the call has been implemented using the `pause` and `resume` methods provided by Mediasoup on Producers. When a user disables their camera or microphone, the corresponding Producer is paused, halting the stream transmission. When re-enabled, the Producer is resumed, allowing for seamless control over media flow without requiring full reconnection or renegotiation of the media session.

4.3.6 Integration with Translation Platform

To support the real-time sign language translation features, it was necessary to integrate a real-time communication platform into the backend system, enabling

seamless bidirectional interaction between deaf and hearing individuals. This platform incorporates an AI-powered translation engine capable of translating text into sign language representations and vice versa.

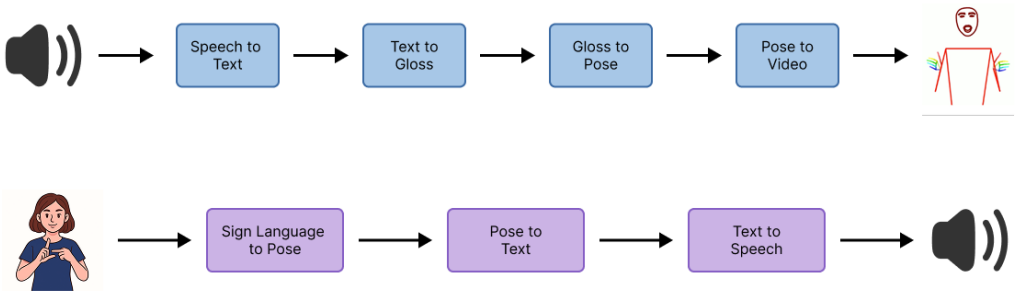


Figure 4.14: Translation Platform Architecture.

The architecture of the communication platform is illustrated in Figure 4.14. Each direction of the translation process consists of a multi-step pipeline, which is described in detail in the following paragraphs.

Voice to Sign Language

This direction of the translation process enables spoken language to be translated into sign language animations. It consists of the following steps:

- **Speech to Text** → Converts spoken language into written text using automatic speech recognition.
- **Text to Gloss** → Translates the recognized text into a sequence of glosses, which provides a condensed description of meaning, structured to closely match the syntax of sign language.
- **Gloss to Pose** → Converts the glosses sequences into pose series, which represent the body and hands movement required to perform a each sign. This transformation relies on a lookup table that associates a predefined sequence of poses to each gloss.
- **Pose to Video** → Renders pose information into a visual animation using a virtual avatar.

Sign Language to Voice

This direction of the translation process allows deaf users to communicate using sign language, which is then translated into synthesized speech. It consists of the following steps:

- **Sign Language to Pose** → Captures user's skeletal data representing the position of key body joints using pose estimation techniques based on computer vision.
- **Pose to Text** → Interprets the sequence of poses and maps them to the corresponding textual representation using a trained 3D-CNN model, which analyzes pose sequences three-dimensionally over time and associates them with natural language expressions.
- **Text to Speech** → Converts the generated text into natural-sounding speech using a text-to-speech system.

Although the initial idea was to implement a separate Python-based server exposing these functionalities, a microservices-based architecture was eventually adopted. Specifically, the approach involved the development of Python-based microservices responsible for real-time translation, which communicate with the Node.js-based web server via TCP sockets. This architectural choice was motivated by the need to ensure that the translation process would not negatively affect call performance, minimizing the processing delay and thus enabling an optimal user experience. Figure 4.15 shows a diagram illustrating the complete architecture.
To access and process user media streams, it was necessary to rely on *Plain Transports*—a special type of transport provided by Mediasoup that allows media streams to be intercepted or injected for further forwarding. Since Mediasoup does not expose APIs for directly accessing raw audio or video tracks, the use of `FFmpeg` was also required. FFmpeg is an open-source multimedia framework that offers a suite of tools for the processing and conversion of audio and video streams.
In the proposed solution, the Node.js-based web server spawns two FFmpeg processes for each communication direction between the two users. These processes act as intermediaries between the server and the corresponding Python-based microservices. The first FFmpeg instance intercepts the RTP stream from a *Plain Transport*, decodes it into a raw media stream, and forwards it to the corresponding microservice listening on a dedicated TCP port. After the translation output is generated, the microservice sends it to the second FFmpeg instance via TCP. This second process re-encodes the media and injects it into another *Plain Transport* to be delivered to the receiving user.
In this way, the hearing user's audio stream is processed by the microservice and translated into a sign language representation to be displayed to the deaf user.

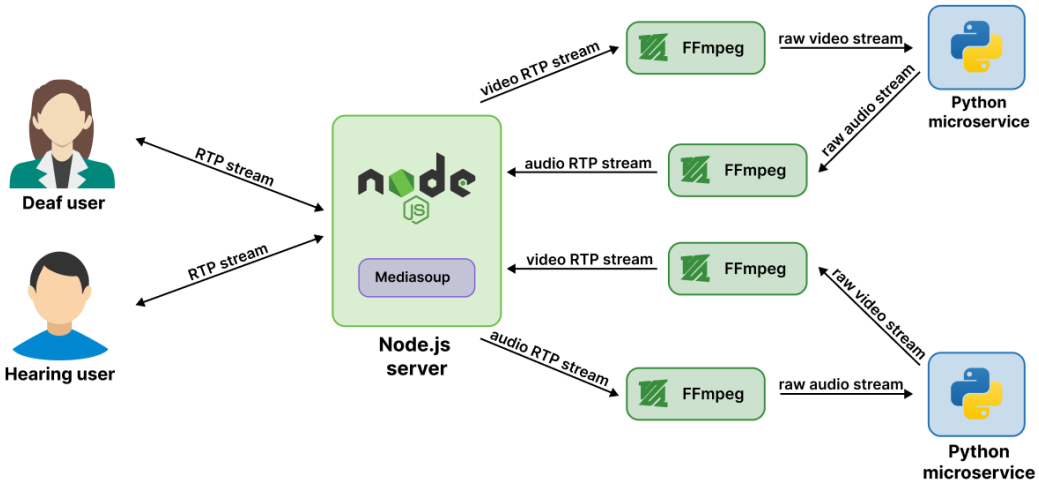


Figure 4.15: Video Call Architecture.

Conversely, the deaf user's video stream is analyzed by a separate microservice to recognize sign language gestures and convert them into voice, which is then sent to the hearing user. Moreover, for the voice-to-text and text-to-voice conversion operations—used as pre-processing or post-processing steps in the execution of sign language translation algorithms—TTS and STT models were employed. In particular, lightweight local models were selected to strike a balance between accuracy and processing speed, thereby reducing the overall latency of the translation pipeline. These models were carefully downloaded and configured during the backend system setup phase to avoid relying on external services, further minimizing delays caused by remote communication.
These architectural choices, combined with the modular microservice-based design, contributed to building a responsive, scalable, and efficient backend capable of supporting real-time translation without compromising performance or user experience, thereby enabling seamless communication between hearing and deaf users.

Chapter 5

Application Testing

This chapter provides a comprehensive description of the testing procedures conducted on both the mobile application and the backend system after the completion of their implementation. It begins with a brief overview of the testing setup, outlining the tools and environments used. Subsequently, it presents a detailed performance analysis based on key metrics, evaluating the system's behavior to assess its robustness, responsiveness, and overall reliability.

5.1 Testing Setup

To carry out the testing activities, the backend system was run locally on a Windows 11 machine equipped with an AMD Ryzen 5 3600 processor, 16 GB of RAM, and an NVIDIA RTX 3060 GPU. To simplify, speed up, and make deployment more flexible—particularly with regard to managing dependencies and supporting future deployments—all backend components were containerized using Docker and organized into a microservices architecture orchestrated via Docker Compose. As for the mobile application, it was tested on three different devices:

- Samsung Galaxy A04s (Android 14 – One UI 6.1)
- Samsung Galaxy A20e (Android 11 – One UI 3.1)
- Redmi Note 10 Pro (Android 13 – MIUI 14.0.9)

The use of these devices enabled performance analysis across different hardware tiers, including lower-end configurations, ensuring that the application remains functional and responsive even under suboptimal hardware conditions. Moreover, testing across different Android versions allowed verifying compatibility and consistent behavior of the application on multiple operating system releases. Additionally,

testing on multiple devices made it possible to verify the correct behavior of the multi-device support functionality.

5.2 Evaluation Metrics

The main goal of the testing phase, beyond verifying the correct functioning of the application and identifying potential bugs, was to evaluate its performance—particularly the efficiency of the real-time sign language translation system integrated into video calls. The tests aimed to assess whether the processing operations required for translation introduced any noticeable delays that could compromise the fluidity and effectiveness of communication between users.
In particular, the objective was to determine whether processing times remained within acceptable limits to ensure a smooth and natural conversational experience. For this reason, the metrics used in the evaluation focused on measuring the delays introduced during the various stages of media stream processing in both directions. These included the time required for data decoding, voice-to-text and text-to-voice conversion using local STT and TTS models, as well as sign language to text and text to sign language translation through the integrated algorithms.
This approach not only enabled the calculation of the overall additional processing delay introduced by the system—on top of the transmission and forwarding latency between clients, which is primarily influenced by external network conditions—but also helped to identify the most computationally demanding stages of the pipeline, highlighting potential targets for future optimization.

5.3 Results

The results of the delay measurements introduced by each stage of media stream processing in both directions are presented in Table 5.1. In the Sign Language to Voice direction, the total delay includes:

- *Frame decoding* , performed on the media stream captured via a Mediasoup plain transport using FFmpeg,
- *Pre-processing*, which involves pose estimation to extract the keypoints used as input for the sign-to-text translation model,
- *Model inference*,
- *TTS conversion*,
- and finally, *audio encoding*, which prepares the synthesized audio for injection back into the Mediasoup plain transport through FFmpeg.

Processing Stage	Video-to-Voice	Voice-to-Video
Frame Decoding	0.002s	–
Pre-processing	0.683s	0.002s
Model Inference	0.069s	0.269s
TTS / STT Conversion	0.071s	0.149s
Audio/Video Encoding	0.078s	0.003s
Total	**0.907s**	**4.529s**

Table 5.1: Average processing times (s) for each stage in both translation directions.

Although the overall latency introduced remains acceptable for a reasonably smooth user experience, several stages still present optimization opportunities—in particular the *Pre-processing* module, which could be replaced by a more efficient, lightweight model. Improvements in TTS conversion, audio encoding and model inference times could also significantly reduce end-to-end delay. While, in the Voice to Sign Language direction, the delay accounts for:

- *Audio pre-processing* (e.g., mono channel conversion),
- *STT conversion*
- *Model inference*, generating sign language representations from textual input,
- and *frame encoding*, which prepares the visual output for delivery into Mediasoup plain transport via FFmpeg. It is important to note that the encoding time reported in the table refers to a single frame, but this process is repeated for each frame, justifying the higher cumulative delay in this direction.

Despite the additional computational demand, the overall latency remains within acceptable limits to ensure a smooth user experience. Nonetheless, further optimizations—particularly in frame encoding efficiency—could enhance the responsiveness of sign language playback, especially when transmitting to deaf users.
In conclusion, the system demonstrates solid performance in real-time translation tasks, with latency levels that support a fluid interaction. However, future improvements may target both performance optimizations, especially in delay-critical stages, and model accuracy, particularly for the sign-to-text translation component.

Chapter 6

Conclusion and Future Works

This chapter serves as the concluding section, providing a comprehensive overview of the work carried out throughout this thesis. It summarizes the key design decisions and the main contributions made to the research. In addition, particular attention is devoted to outlining potential improvements, optimizations, and future developments that could enhance or extend the proposed solution.

6.1 Thesis Summary

This thesis aimed to bridge the communication gap between deaf and hearing individuals by developing a mobile application that integrates a bidirectional sign language translation platform. To achieve this goal, the development and implementation of two separate modules were required: a backend system responsible for managing all functionalities, and a mobile application serving as the client interface. The work began with a brief introduction outlining the core problem and the starting point of the project, namely the bidirectional translation platform. Its architecture and the functioning of each component were described to provide the necessary background. A more detailed problem analysis followed, reviewing the most relevant contributions from the scientific community over the years, highlighting their strengths and limitations. After a brief technical overview of the tools and frameworks used, the focus shifted to the architectural and implementation choices that shaped the final solution.
Starting from a high-level overview of the system architecture, the thesis first examined the mobile application, discussing its functional requirements, use cases, and navigation design. It then explored the backend system in detail, describing its components and the services they provide through mutual interaction. Special

emphasis was placed on the authentication mechanism, the database design, and the RESTful API structure. The discussion concluded with an in-depth look at video call management and the integration of the translation platform.
Finally, the testing phase was presented, detailing the performance evaluation process and analyzing the collected metrics. This analysis also identified the most computationally demanding stages, suggesting potential areas for future optimization. Overall, the resulting product is a fully functional prototype that provides a solid foundation for future enhancements and developments.

6.2 Future Works

Although the developed system constitutes a fully functional prototype and a solid foundation, several limitations remain that suggest opportunities for further enhancement. Future developments may aim to improve both the system's technical capabilities and the overall user experience. The following directions are proposed as potential avenues for future work:

1. *Support for Group Video Calls:* Extending the system to support multi-user video calls would enable interaction among multiple participants without sacrificing the benefits of real-time translation. This enhancement would significantly improve the application's usability in social, educational, or professional settings.

2. *Integration of a Chat Functionality:* The addition of a chat system would provide a complementary communication channel, enriching the overall interaction experience. This could be further augmented by incorporating accessibility-oriented features—such as voice message transcription via the translation platform—thereby addressing communication barriers in text-based conversations.

3. *Implementation of Live Transcription Mode:* A live transcription feature would allow users to benefit from real-time translation without needing to initiate a remote video call. This would enable live, in-person interactions using only the phone's microphone and camera. Such a feature would be especially useful in educational (e.g., lectures for deaf students) and medical contexts (e.g., patient-doctor communication).

4. *iOS Support:* Extending support to iOS devices would ensure platform independence and provide broader accessibility to all users, regardless of the operating system used.

5. *Backend Deployment and Mobile App Publication:* Deploying the backend system and officially publishing the mobile application are essential steps for

making the service publicly accessible. These steps should be preceded by a thorough beta testing phase to identify and resolve any potential bugs or usability issues.

6. *Integration with Existing Communication Platforms:* Developing dedicated plug-ins to integrate the translation features into widely adopted platforms, such as Zoom, Microsoft Teams, or Google Meet, would expand the reach and impact of the system. This is particularly valuable in educational and professional environments where such platforms are already deeply integrated.

Bibliography

[1] World Health Organization. *World Report on Hearing*. 2021. URL: https://www.who.int/publications/i/item/9789240020481 (visited on 07/03/2025) (cit. on p. 1).

[2] Wendy Sandler and Diane Lillo-Martin. *Sign Language and Linguistic Universals*. Cambridge University Press, 2006 (cit. on p. 1).

[3] Mandlenkosi Shezi and Abejide Ade-Ibijola. «Deaf Chat: A Speech-to-Text Communication Aid for Hearing Deficiency». In: *Advances in Science, Technology and Engineering Systems Journal* 5 (Jan. 2020), pp. 826–833. DOI: 10.25046/aj0505100 (cit. on p. 4).

[4] N. Azodi and T. Pryor. *"SignAloud gloves" Multilingual Magazine, University of Washington*. 2016. URL: https://multilingual.com/signaloud-gloves/ (visited on 06/17/2025) (cit. on p. 4).

[5] KinTrans Inc. *"KinTrans: Real-Time Sign Language to Voice Translation"*. URL: https://dallasinnovates.com/kintrans-sign-language-tech-translates-movements-text-voice/ (visited on 06/17/2025) (cit. on p. 4).

[6] Necati Cihan Camgoz, Simon Hadfield, Oscar Koller, Hermann Ney, and Richard Bowden. «Neural Sign Language Translation». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018 (cit. on pp. 5, 7).

[7] Mark Borg and Kenneth P. Camilleri. «Sign Language Detection "in the Wild" with Recurrent Neural Networks». In: *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019, pp. 1637–1641. DOI: 10.1109/ICASSP.2019.8683257 (cit. on p. 5).

[8] Camillo Lugaresi et al. *MediaPipe: A Framework for Building Perception Pipelines*. 2019. arXiv: 1906.08172 [cs.DC]. URL: https://arxiv.org/abs/1906.08172 (cit. on p. 5).

[9] Zhe Cao, Gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. *OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields*. 2019. arXiv: 1812.08008 [cs.CV]. URL: https://arxiv.org/abs/1812.08008 (cit. on p. 5).

[10] Amit Moryossef, Ioannis Tsochantaridis, Roee Aharoni, Sarah Ebling, and Srini Narayanan. *Real-Time Sign Language Detection using Human Pose Estimation*. 2020. arXiv: 2008.04637 [cs.CV]. URL: https://arxiv.org/abs/2008.04637 (cit. on p. 5).

[11] Katrin Renz, Nicolaj C. Stache, Samuel Albanie, and Gül Varol. *Sign language segmentation with temporal convolutional networks*. 2021. arXiv: 2011.12986 [cs.CV]. URL: https://arxiv.org/abs/2011.12986 (cit. on p. 6).

[12] M. Madhiarasan and Partha Pratim Roy. *A Comprehensive Review of Sign Language Recognition: Different Types, Modalities, and Datasets*. 2022. arXiv: 2204.03328 [cs.CV]. URL: https://arxiv.org/abs/2204.03328 (cit. on p. 6).

[13] Zeyu Liang, Huailing Li, and Jianping Chai. «Sign Language Translation: A Survey of Approaches and Techniques». In: *Electronics* 12.12 (2023). ISSN: 2079-9292. DOI: 10.3390/electronics12122678. URL: https://www.mdpi.com/2079-9292/12/12/2678 (cit. on p. 7).

[14] Necati Cihan Camgoz, Oscar Koller, Simon Hadfield, and Richard Bowden. *Sign Language Transformers: Joint End-to-end Sign Language Recognition and Translation*. 2020. arXiv: 2003.13830 [cs.CV]. URL: https://arxiv.org/abs/2003.13830 (cit. on p. 7).

[15] Kayo Yin and Jesse Read. «Better Sign Language Translation with STMC-Transformer». In: *Proceedings of the 28th International Conference on Computational Linguistics*. Ed. by Donia Scott, Nuria Bel, and Chengqing Zong. Barcelona, Spain (Online): International Committee on Computational Linguistics, Dec. 2020, pp. 5975–5989. DOI: 10.18653/v1/2020.coling-main.525. URL: https://aclanthology.org/2020.coling-main.525/ (cit. on p. 7).

[16] Ralph Elliott, John Glauert, Richard Kennaway, Ian Marshall, and Éva Sáfár. «Linguistic modelling and language-processing technologies for Avatar-based sign language presentation». In: *Universal Access in the Information Society* 6 (Feb. 2008), pp. 375–391. DOI: 10.1007/s10209-007-0102-z (cit. on p. 7).

[17] Thomas Hanke. «HamNoSys – Representing Sign Language Data in Language Resources and Language Processing Contexts». In: *4th International Conference on Language Resources and Evaluation (LREC 2004). Proceedings of the LREC2004 Workshop on the Representation and Processing of Sign Languages: From SignWriting to Image Processing. Information techniques*

and their implications for teaching, documentation and communication. Ed. by Oliver Streiter and Chiara Vettori. Lisbon, Portugal: European Language Resources Association (ELRA), May 2004, pp. 1–6. URL: https://www.sign-lang.uni-hamburg.de/lrec/pub/04001.pdf (cit. on p. 7).

[18] Ben Saunders, Necati Cihan Camgoz, and Richard Bowden. *Progressive Transformers for End-to-End Sign Language Production*. 2020. arXiv: 2004.14874 [cs.CV]. URL: https://arxiv.org/abs/2004.14874 (cit. on p. 8).

[19] Transcense Inc. *"Professional & AI-Based Captions for Deaf & HoH | Ava"*. URL: https://www.ava.me/ (visited on 06/17/2025) (cit. on p. 8).

[20] Pedius srl. *"Pedius - Phone calls for the deaf"*. URL: https://www.pedius.org/ (visited on 06/17/2025) (cit. on p. 8).