MANSOURA UNIVERSITY
FACULTY OF ENGINEERING
ARTIFICIAL INTELLIGENCE ENGINEERING PROGRAM

SAHLA

Project members	ID number
Adel Mahmoud Adel Shousha	806237372
Mohamed Ibrahim Taha Mohamed	806127660
Youssef Ahmed Farouk Elkotby	808962813
Abdallah Ahmed Elhoseny Rashad	806147735
Karim Atef Gamal Mashaly	806147227
Mazen Ayman Abdel-Aziz Awad	805502582
Ramy Waleed Abdullah Al-Baqiry	805774428
Ahmed Mohammed Ahmed Foda	805495108
Mostafa Saad Abdo Shaheen	805498138
Omar Amir Elmasry Zidane	805495104

Supervisor

Dr. Islam Ismael Abdullah

Eng. Amr Wahba

Academic Year: 2024 – 2025

Abstract

According to the World Health Organization (WHO), around 466 million people with hearing loss issues, and 34 million of them are children. It is claimed that by 2050 over 900 million people will have suffering hearing loss [1]. Hard hearing people can hear up to a specific limited degree and unobvious by a hearing aid. In contrast, deaf people cannot listen entirely due to head trauma, noise exposure, disease, or genetic condition

In SAHLA, we employ a structured scientific methodology to address communication barriers for deaf and mute individuals. Our approach begins with enhancing receptive language skills by teaching the system to recognize Arabic sign language gestures accurately.

This is achieved through a rich dataset of gestures that the model is trained on. Once the system learns to recognize gestures, it transitions to expressive communication, where recognized gestures are translated into text and speech. For example, a user's hand gesture is captured, processed, and then converted into audible speech or written text using advanced AI models.

This process is applied to a variety of essential vocabulary categories, such as everyday phrases and context-specific terms, to ensure a practical and versatile communication tool. To further enhance real-world usability, SAHLA places gestures into simulated conversation scenarios. After the system demonstrates how a specific gesture corresponds to a word or phrase, users are encouraged to practice these scenarios in real-world situations, ensuring the application's functionality extends beyond theoretical use to practical communication in daily life.

ACKNOWLEDGEMENTS

Dear Dr. Islam Ismael Abdullah and Eng. Amr Wahba, as we approach the end of our journey as students, we are grateful for the knowledge, support, and guidance that you have provided us throughout our project. Your constant encouragement and unwavering commitment to excellence have helped us achieve our goals. We would like to express our deepest appreciation for your invaluable contributions to our project. Your expertise and willingness to share your knowledge with us have been instrumental in shaping our project's success. Special thanks go to Dr. Abeer Tawakol for her exceptional guidance and mentorship. We appreciate her dedication to ensuring that we produce a high-quality project and her assistance in overcoming any challenges. Your support and commitment to our project have helped us develop not only academically but also professionally. We are grateful for the time and effort you invested in us, and we will always remember your valuable teachings. Again, thank you for everything. We will always be proud to have had the opportunity to work under such distinguished professionals.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Nomenclatures

SAHLA	Speech and Hearing Language Assistant
ArSL	Arabic Sign Language
AASL	Arabic Alphabet Sign Language
AI	Artificial Intelligence
ML	Machine Learning
API	Application Programming Interface
UI	User Interface
UX	User Experience
STT	Speech-to-Text
TTS	Text-to-Speech
CNN	Convolutional Neural Network

CHAPTER 1: Project Description

1.1 Problem Statement

1.1.1 Introduction to Hearing Loss

Hearing loss, which includes partial or total inability to hear, is a condition that significantly affects communication and social interaction. It can occur due to various factors such as genetic conditions, illness, aging, or exposure to excessive noise.

1.1.2 Impact of Hearing Loss

The impact of hearing loss varies widely among individuals. People with partial hearing loss may struggle to perceive certain sounds even with the assistance of hearing aids, while those who are completely deaf face profound challenges in everyday communication. Nonverbal communication, such as sign language, becomes their primary means of expression. However, this creates a barrier in interactions with those unfamiliar with sign language, limiting their ability to engage fully in personal, professional, and social activities.

1.1.3 Prevalence and Challenges

- According to the World Health Organization (WHO), around 466 million people with hearing loss issues, and 34 million of them are children. It is claimed that by 2050 over 900 million people will have suffering hearing loss [1].
- The prevalence of hearing impairment was 16.0% of the population of Egypt. This means more than 13 million people across all age groups .The prevalence was high in children up to 4 years old (22.4%).The most common type was conductive hearing loss of minimal to mild severity[2].

1.1.4 Sign Language as a Solution

Sign language is a vital communication tool for the deaf themselves and with ordinary people, and every country has its own language. One of these languages is ArSL, used in the Arabic regions; it was formally introduced in 2001 by the Arab Federation of the Deaf (AFOD) [3]. Sign Language depends on hand movements and gestures to accomplish what you want.

1.2 Literature Review

In [**4**], the authors presented the stages they used to achieve recognition: skin detection, background exclusion, face and hands extraction, feature extraction, and classification using Hidden Markov Model (HMM). The dataset consists of 29 alphabet Arabic letters and numbers from 0to nine with different brightness. They used 253 training images and 104 testing images with 640×480 pixels. The recognition system is tested when dividing the handshape's rectangle surrounding it into 4, 9, 16, and 25 zones. At 16 zones, the recognition rate with 19 states reaches 100%, while at 4 and 9 zones cannot match 100%.

In [**5**], the author explained the nature of the dataset. It is images for the positive samples with the hand sign in different scales, different illumination in the complex background for each hand posture, and the negative samples images from the Internet which do not contain hand posture. Figure 1 shows the stages of translation from Natural language into Sign language.

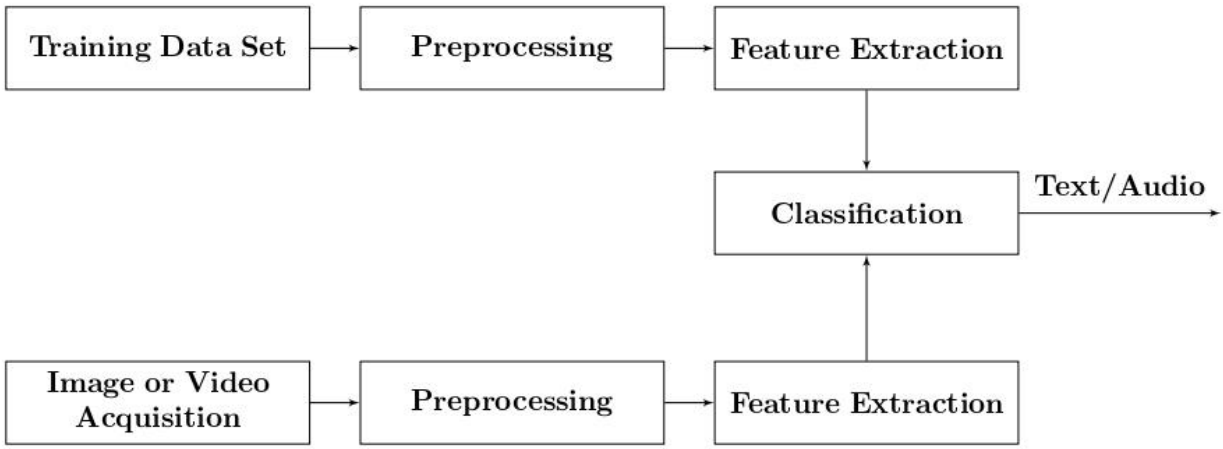


Figure 1: Sign language recognition system [5].

In [**6**], the authors designed a system to translate the ArSL alphabet gestures into text. The used dataset is captured from different smartphones by 30 volunteers. Each volunteer worked on a subset that has 30 images, so the dataset consists of 900 images. The authors used five descriptors to recognize. When using the Histograms of Oriented Gradients (HOG) descriptor,

the proposed ArSL system accuracy is 63.56%. The accuracy of Edge Histogram Descriptor (EHD) is 42%. The accuracy of Discrete Wavelet Texture Descriptor (DWT) is 8%. The accuracy of the Local Binary Pattern (LBP) descriptor is 9.78%. The worst accuracy result is obtained using the 5 Gray-Level Co-occurrence Matrix (GLCM) descriptor; the proposed ArSL system accuracy is 2.89%.

The paper [**7**] starts sorting the sign language into two components; manual and non-manual signs. The manual signs include hand position, orientation, shape, and trajectory. The non-manual signs represent body motion and facial expressions. Convolutional Neural Network (CNN) is a deep learning class employed in image classification; it makes the network quick to learn and find the complex pattern simplicity. CNN still uses the Backpropagation and its derivatives training methods to learn from data. The author used a dataset of images containing 2030 images of numbers (from 0 to 10) and 5839 images of 28 letters of Arab sign language, i.e., 7869 RGB colour images with 256×256 pixels. These images are taken from different signers and different luminosity intensities.

In [**8**], the authors exhibit sign language differently. Most researchers try to obtain the text rather than the semantic. To recognize the word with its semantic, they combined CNN with a semantic layer, and it maps the word to the meaning. A mobile camera picks the dataset in a different surrounding. The model achieved good recognition accuracy of 88.87%.

The authors in [**9**] proposed the first ArSL recognition system that converts ArSL to Arabic sentences. The machine translation system is Rule-based, and it has three stages; the input Arabic Sign language word is processed for Morphological analysis then Syntactic analysis. Finally, transfer to Arabic sentences. However, the system used a corpus that has sentences that are used in health centers. It has 600 sentences that consist of 3327 sign words with 593 unique sign words. The proposed dataset is divided into training, validation, and testing datasets, with a percentage of 70%, 15%, and 15%, respectively. The results of the system are calculated Manually and automatically. However, the manual evaluation shows that 80% of the sentences are accurately translated, and 2 ArSL experts do the evaluation. Also, it is evaluated automatically by BLEU and TER metrics and gets 0.39 and 0.45 repetitively.

1.3 Requirements

To build an application like SAHLA, there are some requirements to be met in order to come up with the best and expected results. They are mainly divided to technical, functional and non-functional requirements, In the following section we will present them.

1.3.1 Technical Requirements

1. **UI/UX**: Creating a user-friendly and accessible experience is a core focus of SAHLA. Considering the needs of the deaf and mute community, we designed a UI that is simple, intuitive, and easy to navigate, ensuring seamless interaction for all users.

2. **Mobile Application**: Developed using state-of-the-art mobile development technologies, SAHLA's mobile app delivers a responsive and efficient interface tailored for accessibility and performance.

3. **MediaPipe AI Model**: At the heart of SAHLA's gesture recognition is a robust AI model powered by MediaPipe. It ensures accurate, real-time identification of hand gestures, facilitating a smooth and reliable translation process.

4. **3D Blender Modeling**:Used for creating 3D models and animations to enhance the accuracy of gesture recognition and improve the application's visual appeal.

5. Google Text-to-Speech (TTS) and Speech-to-Text (STT): These technologies work together to enable seamless communication. TTS converts recognized gestures into speech output for auditory communication, while STT translates spoken words into text, facilitating integration for sign language conversion and interactive functionalities.

6. **Gradio**: Facilitates interactive interfaces for testing and refining AI models, ensuring efficient development and validation processes.

1.3.2 Functional Requirements

1. The app should provide real-time sign language translation, allowing users to communicate by translating their sign language gestures into spoken words, similar to how Google Translator works.

2. The app should feature a training module to help users improve their sign language skills by presenting visual prompts (e.g., images, 3D models) of various signs, and then having users replicate the gestures for feedback.
3. The app should process the user's sign language gestures using MediaPipe, which will detect and recognize the gestures for accurate translation into speech.
4. The app should allow users to translate their sign language gestures into spoken words in real-time using Google Speech-to-Text and Text-to-Speech APIs.
5. The app should interact with users by converting their sign language gestures into spoken words, providing real-time spoken responses using Google Text-to-Speech.
6. The app should use Gradio to create a simple interface for users to interact with, including gesture training and communication features.
7. The app should use React Native for mobile app development, ensuring compatibility across both iOS and Android platforms.

1.3.3 Non-functional Requirements

1. Performance:

a. Resize and compress images because big images slow mobile apps down.

b. The app should respond quickly to user interaction to keep their attention.

2. Compatibility:

a. The app should be compatible with different types of devices and operating systems.

3. Usability:

a. The app should be easy to use for deaf and mute individuals, including accessibility features like clear, high-contrast visuals and simple, uncluttered design to ensure ease of use.

b. The app should be designed with a simple and intuitive interface, making it easy for users to navigate and seamlessly interact with the sign language translation features.

4. Availability:

a. Use cloud services to store data.

1.4 Objectives And Outcomes

Objectives and outcomes are important for a project because they provide clarity and direction to the project team. Objectives define what the project is trying to achieve, while outcomes specify the specific results that will be delivered by the project.

1.4.1 Project Objectives

Having clear objectives helps to ensure that everyone involved in the project has a common understanding of what needs to be accomplished. In the planning phase we had clear objectives that helped us to continue which are:

- Be the first Arabic speaking application concerned with Arabic sign Language.
- Improve receptibility, expressibility and communication skills of deaf and mute individuals.
- Assist users in learning and enhancing their sign language skills through interactive features.
- Help caregivers, educators, and professionals track the user's progress in learning and using sign language effectively.

1.4.2 Expected Outcomes

Outcomes provide a tangible measure of the project's success. They define the specific results that need to be achieved and establish clear expectations for the project team. In the following section we will list what we expected from SAHLA at the end:

• To be a suitable solution used for the deaf and mute individuals .

• We hope that the deaf and mute community find our application to be a reliable tool that they can seamlessly integrate into their daily lives with confidence.

• Ultimately, we aspire for our solution to become widely adopted and the preferred choice recommended by the community, with members themselves becoming our main promoters.

CHAPTER 2: Building Blocks – Tools and Technologies

2.1 INTRODUCTION

The purpose of this chapter is to provide an in-depth understanding of the building blocks, tools, and technologies utilized in our graduation project, SAHLA, which aims to develop an application for bridging the communication gap for individuals who are deaf or mute by translating sign language into text or speech and vice versa. While the previous chapter provided an overview of the project's description and objectives, this section will focus on the high-level architecture and the main components that constitute our solution.

At the heart of SAHLA lies a meticulously designed high-level architecture that ensures a seamless user experience and facilitates effective communication for individuals who are deaf or mute. The architecture diagram provided below illustrates the interconnected components and their specific functionalities within the application.

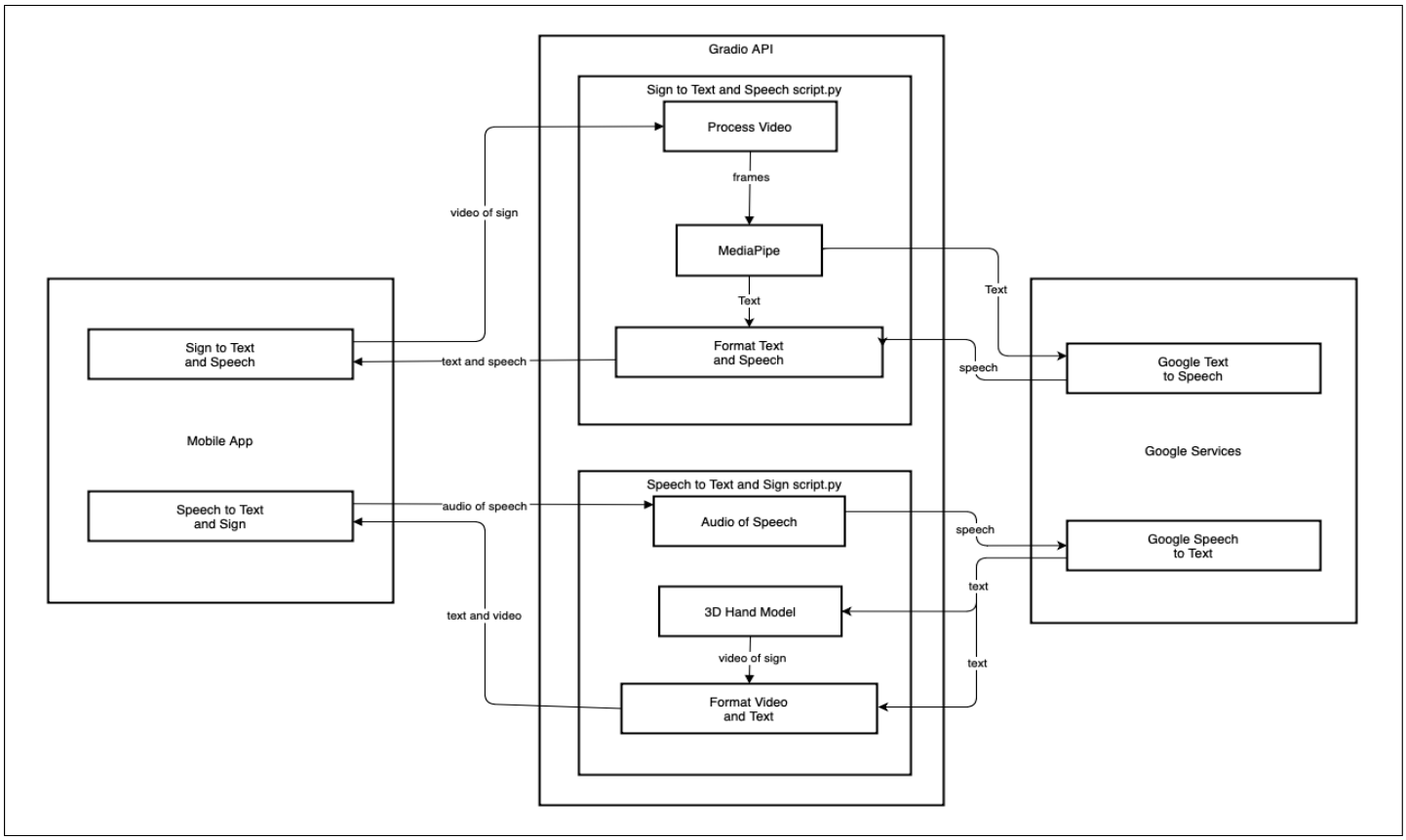


Figure 2: SAHLA SYSTEM ARCHTECHITURE

The SAHLA application is designed to bridge the communication gap for deaf and mute individuals by offering bidirectional translation between sign language and spoken language. The user interface presents two main functionalities upon opening: "Sign to Text and Speech" and "Speech to Text and Sign."

2.1.1 Sign to Text and Speech:

1. **Video Input**: The user records a video of their sign language gestures.
2. **Gesture Recognition**: MediaPipe processes the video to recognize and interpret the hand gestures.
3. **Text and Speech Conversion**: The recognized signs are converted into text, which is then transformed into speech using Google's Text-to-Speech service.
4. **Output**: The text and speech output are displayed or played back to the user through the mobile app.

2.1.2 Speech to Text and Sign:

1. **Audio Input**: The user speaks, and the speech is recorded.
2. **Speech Recognition**: Google's Speech-to-Text service converts the speech into text.
3. **Sign Language Conversion**: The text is mapped to corresponding sign language gestures, which are animated using a 3D hand model.
4. **Output**: The sign language video is displayed to the user via the mobile app.

2.1.3 Key Components and Features:

- **Gesture Recognition API**: Used for recognizing and interpreting sign language gestures.
- **Text-to-Speech and Speech-to-Text APIs**: Facilitate the conversion between text and speech.
- **3D Hand Model**: Generates realistic sign language animations based on text input.
- **Mobile App**: Serves as the user interface, handling video and audio capture, processing input through backend services, and presenting output in a user-friendly format.

2.1.4 User Experience and Future Improvements:

- The app is designed to be intuitive, with real-time processing to minimize delays.

- It relies on cloud services, which may require an internet connection.
- Future enhancements could include supporting phone calls directly within the app, expanding language support, and improving gesture recognition accuracy.
- Security and privacy are addressed through encryption and secure authentication methods to protect user data.

By presenting this high-level architecture, we aim to provide a comprehensive overview of our solution's structure. The subsequent sections of this chapter will delve into each building block, detailing the decisions made, the tools and technologies employed, and the respective pros and cons. Let us now embark on an exploration of the various components that make up our application across our sub-teams and for the team.

2.2 Gesture Recognition

2.2.1 AI-Based Gesture Recognition: Why It's Essential for SAHLA

In the SAHLA project, the ability to accurately recognize and interpret Arabic Sign Language (ArSL) gestures is paramount. Traditional methods relying on hardcoded logic would be inefficient due to the inherent complexity and variability of sign language gestures. Additionally, real-world environments often introduce noise and variability that can complicate gesture recognition. By employing AI, specifically machine learning models, we can achieve a more adaptable and accurate system capable of handling the diverse gestures used in ArSL. Furthermore, AI models can improve their accuracy over time with more data, without requiring manual updates, ensuring the system remains effective and up-to-date.

2.2.2 Custom Model Development vs. Pre-Trained Models

A critical decision in our project was whether to develop a custom gesture recognition model from scratch or utilize an existing pre-trained model. Given the significant time and

effort required to train a model from scratch, we decided to leverage a pre-trained solution that could be fine-tuned to our specific needs.

2.2.3 Evaluating Model Options: Challenges and Trade- offs

During the model selection phase, we considered several options, including open-source libraries and models from companies like OpenAI. For instance, models like OpenPose were evaluated for their hand-tracking capabilities, but they were found to be less accurate in recognizing the specific gestures used in ArSL. Other models required significant computational resources, making them impractical for deployment on mobile devices or in real-time applications. These trade-offs were crucial in guiding our final decision towards a more suitable solution that balanced accuracy, performance, and resource efficiency.

2.2.4. Selecting MediaPipe: Leveraging Google's Scalable Solution

MediaPipe emerged as the optimal choice for our gesture recognition needs. Developed by Google, MediaPipe offers a scalable and efficient framework for processing hand movements and recognizing gestures. Its cross-platform capabilities allow seamless integration with our application, ensuring a consistent user experience across different devices. Additionally, MediaPipe's active community and extensive documentation provide valuable support for customization and optimization, aligning perfectly with SAHLA's requirements.

2.2.5 How MediaPipe Processes and Recognizes ArSL Gestures

MediaPipe efficiently translates ArSL gestures into text through a streamlined process:

- **Hand Detection and Tracking:** MediaPipe identifies and tracks hands in real-time using advanced computer vision techniques.
- **Landmark Detection:** It pinpoints 21 key landmarks on each hand to capture detailed hand poses.
- **Feature Extraction:** Relevant features are extracted from these landmarks to identify specific gestures.
- **Gesture Interpretation:** A machine learning model, fine-tuned with ArSL data, interprets these features and converts them into corresponding text.

This approach ensures accurate and efficient gesture recognition, enhancing the SAHLA application's performance.

2.3 Sign to Text and Speech - Speech to Text and Sign.

In the SAHLA project, we have implemented a seamless and interactive user experience by integrating gesture recognition with speech synthesis. This integration allows users who are deaf or mute to communicate effectively in various settings, breaking down communication barriers.

2.3.1 Sign to Text and Speech Functionality:

The Sign to Text and Speech feature is designed to convert Arabic Sign Language (ArSL) gestures into spoken words, enabling users to communicate with hearing individuals effortlessly. The process begins with the user performing sign language gestures, which are captured by the application. These gestures are then recognized and converted into text using the MediaPipe framework, a robust tool for gesture recognition.

2.3.2 Role of Google's Text-to-Speech (TTS) API:

Once the gestures are translated into text, the next step is to convert this text into speech. For this purpose, we utilize Google's Text-to-Speech (TTS) API. This API is chosen for its ability to handle challenging conditions, such as noisy environments, and for its strong support for the Arabic language. The TTS API converts the text output from the gesture recognition process into natural-sounding speech, ensuring that the communication is clear and understandable.

2.3.3 Why Google's APIs are Suitable for SAHLA

Google's Speech-to-Text (STT) and Text-to-Speech (TTS) APIs are ideal for our project due to their robustness and language support. The STT API accurately transcribes spoken words into text, even in noisy conditions, while the TTS API provides high-quality speech synthesis in Arabic. These features are crucial for our project, as they ensure that the application can be used effectively in various real-world scenarios.

2.3.4 Enhancing User Experience

The integration of these APIs significantly enhances the user experience by providing a seamless communication loop. Users can express themselves through sign language, and the

application will convert these signs into spoken words, facilitating interaction with hearing individuals. This feature empowers users to communicate effortlessly in different settings, promoting inclusivity and accessibility.

In conclusion, the Sign to Text and Speech functionality in SAHLA is a testament to the power of integrating advanced technologies to address real-world challenges. By choosing Google's STT and TTS APIs, we have ensured that our application is not only effective but also adaptable to diverse communication environments, ultimately fostering a more inclusive society.

2.4 3D Modelling and Animation with Blender

The use of 3D modeling and animation in SAHLA plays a critical role in delivering accurate and visually engaging representations of Arabic sign language gestures. Our aim in this section was to help users understand and learn how to perform sign language accurately through our visually engaging animations.

2.4.1 Blender as the Core Tool

Blender, a powerful and open-source 3D modeling software, was chosen as the primary tool for creating and animating the hand models used in SAHLA. Its extensive capabilities in rigging, animation, and rendering made it the ideal choice for this project. Blender's open-source nature also allowed for customization and integration, ensuring that the animations met the unique requirements of Arabic sign language.

2.4.2 Creating Sign Language Animations with Blender

1. **Starting with the 3D Hand Model:** The process begins with selecting a highly detailed 3D hand model, Its anatomical precision ensures the animations are both functional and visually authentic, providing a lifelike and engaging learning experience.
2. **Rigging, Weight Painting, and Inverse Kinematics :**A skeletal framework is added to the hand model, enabling natural movement. Weight painting ensures smooth deformations of the hand as the bones move. To simplify animation, Inverse Kinematics

(IK) is applied, allowing precise control by automatically adjusting the entire hand's movement when specific parts, like fingertips, are manipulated.

3. **Animating the Gestures :**Each animation is crafted to represent an Arabic alphabet letter, designed to be smooth and easy to understand for users.
4. **Rendering the Animations :**Blender's rendering engine adds realistic lighting and shadows, enhancing the clarity and visual appeal of the gestures.
5. **Integration into SAHLA :**The animations are integrated into the SAHLA app, providing users with an engaging and educational tool to learn and perform sign language gestures.
6. **Making Sign Language Accessible :**The animations serve as a bridge for communication, helping users understand and replicate sign language gestures effectively.

2.5 Mobile Development

Mobile development is a critical component of our project, enabling us to deliver an interactive and accessible experience to users. This section outlines the decision-making process, trade-offs, and the tools and technologies employed in our mobile app development.

2.5.1 Decision Paradigms and Trade-offs

During the planning and development phases, we considered several factors to select the most suitable framework for our mobile app development. These factors included development efficiency, performance, cross-platform compatibility, and community support. After careful evaluation, we chose React Native as our framework of choice.

2.5.2 Tools and Technologies

Our mobile development process leveraged the following tools and technologies:

- React Native: The core framework used to develop our mobile app, offering cross-platform compatibility and a rich set of UI components. It allows for a single codebase deployment on both Android and iOS, enhancing development efficiency.
- Expo: Simplified our development process by providing a comprehensive set of tools and services, eliminating the need for native configurations. Expo's fast development cycles and hot-reload feature accelerated our iteration process.

- TypeScript: Enhanced our development experience with static typing, improving code quality and maintainability.
- Folder Structure: Organized our codebase logically, ensuring scalability and ease of maintenance. The structure was designed to separate components, screens, and utilities effectively.
- API Integration: Integrated with our backend services using Axios, facilitating smooth data communication and enhancing app functionality.
- **Node.js Server: Provided a robust backend solution, supporting our app's data handling and processing needs.**

2.5.3 Why React Native?

We selected React Native for its ability to deliver a high-performance, cross-platform application with a unified codebase. The advantages of React Native include:

- Cross-platform development: Enables a single codebase for both Android and iOS, reducing development time and effort.
- Fast development and hot-reload: Accelerates development cycles with quick iteration and experimentation.
- Rich UI capabilities: Provides a extensive set of pre-built components and customization options, allowing for visually appealing and interactive interfaces.
- Strong community support: Benefits from a large and active community, offering extensive resources, packages, and libraries that enhance development productivity.

In conclusion, our mobile development approach, centered around React Native, Expo, and TypeScript, provides a efficient and scalable solution for delivering a high-quality user experience across platforms.

2.6 UI/UX Experience

UI/UX Design plays a crucial role in creating an intuitive and engaging user experience for our application. This section explores the tools and technologies employed in the design process, highlighting their strengths and weaknesses.

2.6.1 Adobe Illustrator

Adobe Illustrator is a powerful vector graphics editor, ideal for creating logos, icons, and scalable designs.

Pros of Adobe Illustrator:

- High precision and control over design elements.
- Extensive tools for drawing and typography.
- Creates scalable graphics without loss of quality.

Cons of Adobe Illustrator:

- Steep learning curve for beginners.
- Not suited for photo editing.
- Subscription-based pricing can be expensive.

2.6.2 Adobe Photoshop

Adobe Photoshop is the industry standard for raster graphics and photo editing.

Pros of Adobe Photoshop:

- Extensive features for image manipulation and digital art.
- Widely used in the industry with a vast range of filters and effects.
- Supports complex photo editing tasks.

Cons of Adobe Photoshop:

- Requires powerful hardware for optimal performance.
- Complex interface for new users.
- Subscription model may be costly.

2.6.3 Adobe XD

Adobe XD is a design and prototyping tool known for its user-friendly interface.

Pros of Adobe XD:

- Robust prototyping capabilities for interactive user experiences.

- Seamless integration with Adobe Creative Cloud.
- Extensive library of UI kits and plugins.

Cons of Adobe XD:

- Limited collaboration features in the free version.
- Performance issues with complex designs.
- Smaller community compared to other tools.

2.6.4 Figma

Figma is a cloud-based design tool with real-time collaboration features.

Pros of Figma:

- Real-time collaboration for seamless team work.
- Cross-platform compatibility, accessible on any OS.
- Extensive plugin ecosystem for additional functionality.

Cons of Figma:

- Performance issues with large files or slow internet.
- Limited offline functionality.
- Steeper learning curve for some users.

2.6.5 Why We Chose Figma, Photoshop, and Illustrator

We selected Figma, Photoshop, and Illustrator for their unique strengths that align with our project needs:

- **Figma**: Offers excellent collaboration features, enabling real-time teamwork and version control.
- **Photoshop**: Provides unmatched photo editing capabilities, essential for our design work.
- **Illustrator**: Offers precision in vector graphics, crucial for creating scalable and detailed designs.

These tools together ensure a comprehensive design workflow, covering vector graphics, photo editing, and collaborative prototyping, meeting the diverse requirements of our project.

2.7 DEPLOYMENT

Deployment is a critical phase in bringing our application to users. It involves setting up the necessary infrastructure and deploying the application components to a production environment.

2.7.1 Mobile App Deployment

In the deployment of our application, we utilized a combination of tools to ensure a seamless and efficient launch. Here's a brief overview of the tools and their roles:

Koyeb: Our primary hosting platform, Koyeb, provides scalable and managed hosting solutions. It ensures that our server is deployed consistently across environments and handles load balancing and auto-scaling efficiently.

Hugging Face: Hugging Face was instrumental in hosting our Gradio app, which serves as the interface for our machine learning model. It offers a robust platform for deploying and managing ML models.

Gradio: Gradio simplifies the creation of web interfaces for machine learning models. It allows us to easily interact with our model through a user-friendly interface, which is hosted on Hugging Face.

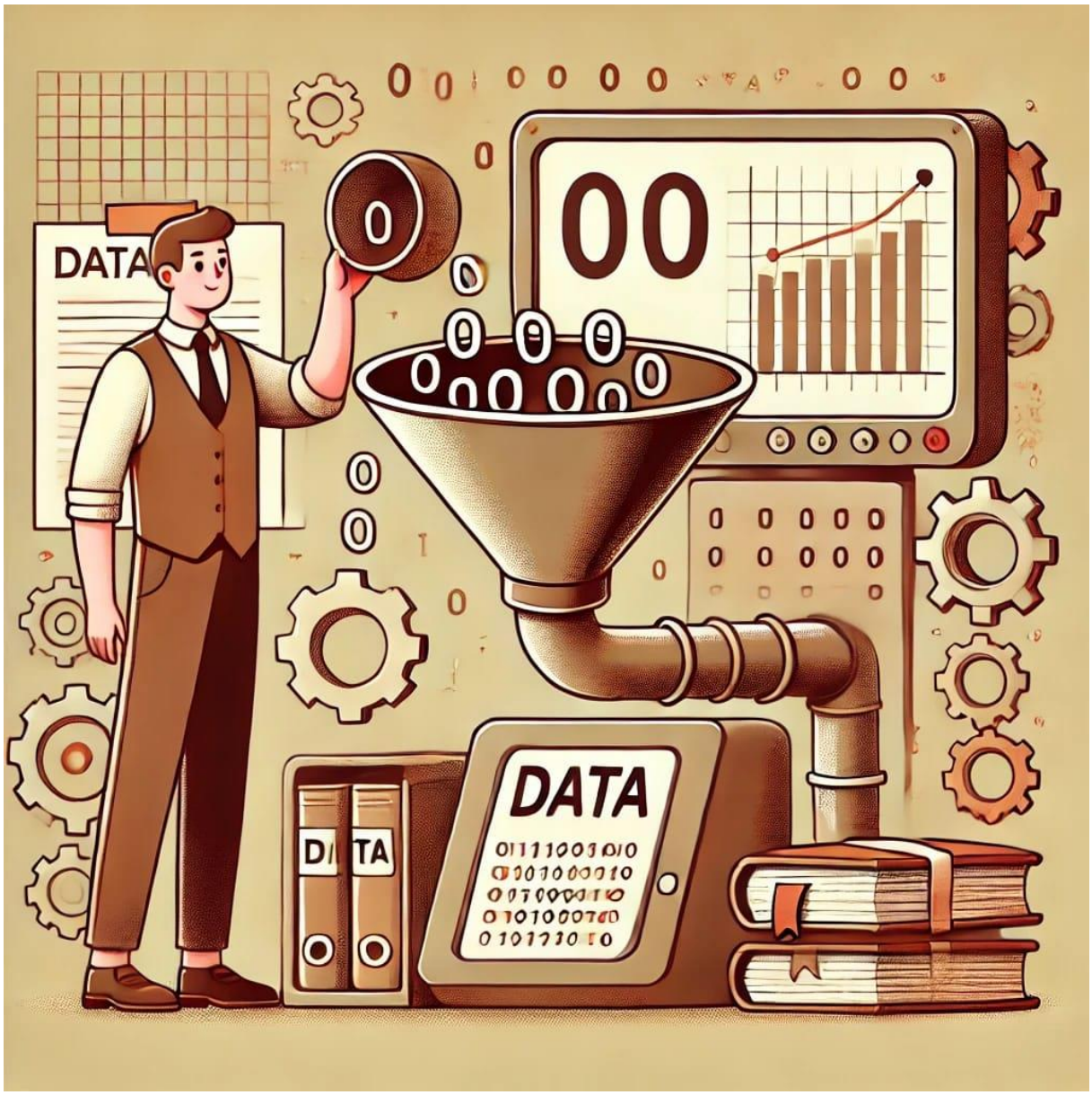
GitHub Actions: We employed GitHub Actions to automate our deployment pipeline. This ensures that our application updates smoothly and automatically whenever we push changes to our repository.

By integrating these tools, we achieved a comprehensive deployment strategy that leverages the strengths of each platform. Koyeb handles the hosting, Hugging Face and Gradio manage the ML interface, and GitHub Actions streamline our deployment process.

2.7.2 Cross App Deployment

Our mobile application is designed to be accessible on both Android and iOS platforms, ensuring a wide reach to users. Leveraging technologies like React Native and Expo, we achieve cross-platform compatibility, allowing for a consistent user experience across different devices. This approach not only broadens our audience but also simplifies maintenance and updates, as changes can be implemented uniformly across platforms.

CHAPTER 3: Data Preparation



3.1 The Role of Data in Machine Learning Models

The success of any machine learning model is fundamentally linked to the quality, diversity, and relevance of its dataset. Data acts as the essential fuel for machine learning algorithms, empowering them to learn patterns, adapt to variations, and make accurate predictions. . A dataset must not only be vast but also relevant, representative, and accurate to enable the system to generalize well across different scenarios. Poor data quality can lead to biased predictions, inefficiencies in training, and a lack of trust in the final system—a risk that is especially critical for real-world applications such as our sign language recognition.

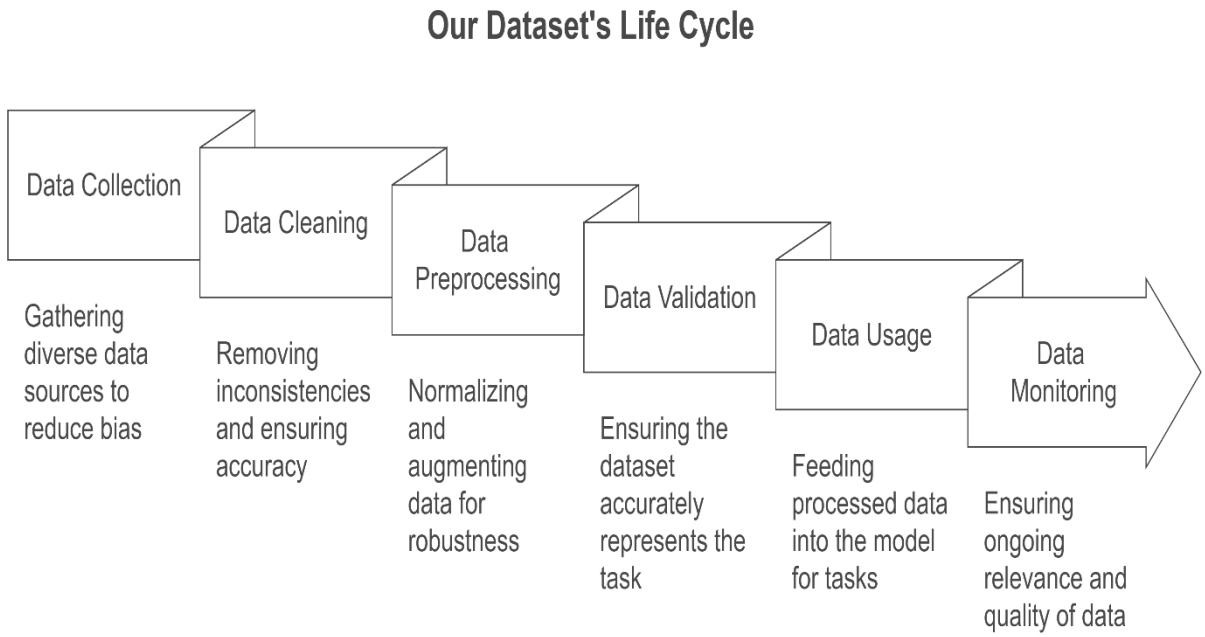


Figure 3 : The Dataset Life Cycle in SAHLA Project

3.1.1 Stages of the Data Lifecycle

The data lifecycle in machine learning involves multiple interconnected stages: data collection, cleaning, preprocessing, validation, usage, and monitoring. Each stage plays a critical role in ensuring the dataset effectively supports the model's training and long-term reliability.

During **Data collection**, it is essential to ensure diversity in sources and contributors. A diverse dataset reduces biases and improves generalization, ensuring the model performs well across various scenarios. For example, a dataset focused solely on one demographic group might fail when applied to a broader population.

Data cleaning follows, focusing on identifying and rectifying inconsistencies, removing duplicates, addressing noise, and ensuring overall accuracy. This step ensures that the dataset is free of errors that could hinder the training process.

Preprocessing, such as normalization and augmentation, is another vital step in the data lifecycle. These techniques help expose the model to a wide range of conditions it may encounter during real-world use. For example, normalization ensures that data remains consistent across different inputs, while augmentation adds variations such as flips, rotations, or lighting changes to make the model more robust and discriminating.

Validation involves rigorous checks to confirm the dataset accurately represents the task at hand.

Usage which is feeding the processed data into the trained model to perform tasks relevant to the application. For example, in our application, the data is utilized to recognize hand gestures and translate them into text or speech, enabling seamless communication for deaf and mute individuals.

Finally, **monitoring** ensures the dataset remains relevant and high quality over time, especially as model is deployed and new data becomes available.

3.1.3 Challenges in Dataset Generalization

Machine learning models thrive on well-prepared datasets that address specific goals and challenges. In the context of sign language recognition, data preparation becomes even more vital. The need to recognize subtle differences in hand shapes and movements demands a dataset rich in diversity and precision. Factors like skin tone, hand size, and background clutter must be considered during collection and preparation to ensure a model capable of robust real-world performance.

In our project, the dataset serves as the backbone for developing a robust system capable of translating Arabic sign language into text and speech. This data will be used to train and test a machine learning model, ensuring it recognizes and interprets gestures reliably. The addition of a "None" folder in the dataset further strengthens the model's ability to identify idle states and avoid false positives. Each gesture's inclusion and preparation are critical to building a model that performs effectively not only in controlled environments but also in unpredictable, real-world scenarios.

3.2 Challenges and Considerations in Data Quality

3.2.1 Data Diversity and Representation

High-quality data is not just about quantity—it must also be representative, clean, and consistent. The dataset must encompass all possible variations while minimizing noise, such as mislabeled samples or poor-quality images. This becomes especially critical in applications like sign language recognition, where small inconsistencies in gesture interpretation could lead to significant miscommunication and errors in translation.

Sign language recognition requires data that reflects a wide range of factors, including different hand shapes, orientations, movements, and variations in signer appearances. It must also account for environmental conditions, such as changes in lighting, background clutter, and the presence of other objects or people. A failure to incorporate these factors can introduce biases into the system. For instance, if a dataset primarily features gestures from a single individual or demographic group, the model may struggle to generalize to users from different backgrounds.

3.2.2 Data Consistency and Cleanliness

Ensuring data consistency is equally critical. Variations in data labeling, image resolutions, or capture devices can create inconsistencies that reduce model performance. Data cleanliness, therefore, becomes paramount, involving processes such as rigorous labeling, filtering for low-quality or irrelevant data, and ensuring uniformity in format and structure. These steps are vital to prevent errors from propagating during training and to produce a model that meets the high accuracy requirements of sign language recognition.

3.2.3 Class Imbalance and Noise Mitigation

Another key challenge is class imbalance, where certain signs may appear more frequently than others in the dataset. If left unaddressed, this can lead to a model that disproportionately favors more common signs, reducing its accuracy for less frequent gestures. To combat this we employed techniques such as data augmentation and synthetic data generation to address this imbalance, ensuring the model has sufficient exposure to all classes equally.

Noise within datasets is another challenge that requires careful mitigation. This noise can include irrelevant images, poorly cropped gestures, images with partial hand visibility, or variations in gesture execution. By implementing a robust preprocessing pipeline, these challenges can be mitigated. For example, we incorporate image filtering and augmentation techniques to standardize the data and prepare it for efficient learning.

3.3 Overview of the RGB Arabic Alphabet Sign Language Dataset

3.3.1 Dataset Composition and Collection

In our project, we employ the RGB Arabic Alphabet Sign Language (AASL) dataset, a meticulously curated resource comprising 7,857 labeled images representing the 31 alphabets of Arabic sign language. This dataset was developed through a collaborative effort involving over 200 participants, ensuring a diverse range of contributors and data collection methods. Images were captured using a variety of devices, including webcams, digital cameras, and mobile phones, introducing natural variations in resolution, angles, and lighting conditions. These variations help ensure the dataset's robustness for training machine learning models.

The AASL dataset's quality and reliability were carefully maintained through the involvement of Arabic sign language experts. These experts supervised the data collection process, validated the labeled gestures, and filtered out images that did not meet the required standards. This rigorous quality control process ensures that the dataset is both comprehensive and trustworthy.

3.3.2 Dataset Organization and Structure

The dataset is organized into 31 folders, each corresponding to an individual Arabic alphabet sign. To enhance its utility, an additional "None" folder was introduced, containing images of

neutral hand states where no specific gesture is being performed. This folder plays a crucial role in enabling the model to distinguish between active and idle states, reducing the likelihood of false detections during real-world application.

#	Letter name in English Script	Letter name in Arabic Script	# of Images
1	ALEF	أ (ألف)	287
2	BEH	ب (باء)	307
3	TEH	ت (تاء)	226
4	THEH	ث (ثاء)	305
5	JEEM	ج (جيم)	210
6	HAH	ح (حاء)	246
7	KHAH	خ (خاء)	250
8	DAL	د (دال)	235
9	THAL	ذ (ذال)	202
10	REH	ر (راء)	227
11	ZAIN	ز (زاي)	201
12	SEEN	س (سين)	266
13	SHEEN	ش (شين)	278
14	SAD	ص (صاد)	270
15	DAD	ض (ضاد)	266
16	TAH	ط (طاء)	227
17	ZAH	ض (ظاء)	232
18	AIN	ع (عين)	244
19	GHAIN	غ (غين)	231
20	FEH	ف (فاء)	255
21	QAF	ق (قاف)	219
22	KAF	ك (كاف)	264
23	LAM	ل (لام)	260
24	MEEM	م (ميم)	253
25	NOON	ن (نون)	237
26	HEH	ه (هاء)	253
27	WAW	و (واو)	249
28	YEH	ي (ياء)	272
29	TEH MARBUTA	ة (تاء مربوطة)	257
30	AL	ال	276
31	LAA	لا	268
32	None		218

Table 1 :ArsL Dataset distribution.

The inclusion of diverse contributors helps address biases that might arise in sign language systems. By collecting data from a range of individuals, we ensure the dataset's ability to generalize across different users. Variations in environmental conditions, such as lighting and backgrounds, further enhance the dataset's robustness, preparing it for the challenges of real-world use.

Figure 4: Sample from the dataset.

Figure 5 : Sample from the "None" folder

3.3.3 Sample Images and Variability

The figures above provide a visual representation of the dataset, illustrating the diverse range of gestures it encompasses, including both active signs and neutral states. Each image showcases an individual performing a specific sign with deliberate variations in lighting, angles, and backgrounds, enhancing the dataset's robustness for training and testing purposes. Additionally, a sample from the "None" folder highlights the neutral hand positions, which play a critical role in enabling the model to distinguish between active gestures and idle states. The dataset's comprehensive structure, including the carefully curated "None" samples, ensures a balanced representation of all scenarios. Combined with the validation process conducted by Arabic sign language experts, this dataset stands as an invaluable resource for developing accurate, reliable, and adaptable sign language recognition systems, capable of performing effectively in diverse real-world environments.

3.4 Preprocessing and Enhancing the Dataset

The preparation of the AASL dataset involved several critical steps to optimize it for training machine learning models. One of the most significant enhancements was the inclusion of the "None" folder, which contains images representing idle states. This addition is essential for enabling the model to distinguish between active gestures and neutral positions, thereby reducing the likelihood of false positives and improving overall performance.

3.4.1 Addressing Class Imbalance with Data Augmentation

To mitigate any class imbalance within the dataset, we applied a variety of data augmentation methods, such as flipping, rotating, scaling, and altering lighting conditions. These enhancements expand the dataset's variability by introducing simulated real-world conditions, allowing the model to learn from a wider array of scenarios. For example, gestures captured under dim lighting are adjusted to represent well-lit environments, aiding the model in adapting to diverse situations with improved reliability.

By preparing a dataset that incorporates diverse and high-quality images, the stage is set for leveraging advanced tools like Media pipe. This hand-tracking library will allow for the seamless recognition of gestures, ensuring that our application achieves both accuracy and efficiency. The steps taken in preprocessing will enable the smooth integration of gesture recognition with downstream systems, such as text-to-speech and mobile deployment.

3.4.2 Ensuring Data Quality for Effective Arabic Sign Language Recognition.

The preparation of high-quality data is the linchpin for developing a robust sign language recognition system. Our meticulous handling of the AASL dataset ensures that the nuances of Arabic sign language gestures are captured effectively, paving the way for accurate gesture recognition through tools like Media pipe. By addressing challenges such as class imbalance and environmental variation, we have created a dataset that is both comprehensive and reliable. This effort forms the cornerstone for subsequent steps in our project, enabling seamless integration of gesture recognition with text-to-speech and speech-to-text systems, as well as the creation of an intuitive user interface and a reliable mobile application

CHAPTER 4: Gesture Recognition

4.1 Introduction

In our journey of developing SAHLA, a sign language translation application, we sought to bridge the communication gap by converting Arabic Sign Language (ArSL) gestures into Arabic text. Achieving this required training a machine learning model capable of recognizing the diverse hand gestures integral to our system. To accomplish this, we turned to MediaPipe, an open-source framework that provides robust tools for building pipelines to perform computer vision inference on various sensory inputs. MediaPipe played a pivotal role in enabling efficient and accurate gesture recognition for our application.

4.2 Mediapipe Overview

4.2.1 MediaPipe Solutions guide

We leveraged MediaPipe Solutions, a powerful suite of libraries and tools designed for the seamless integration of artificial intelligence (AI) and machine learning (ML) techniques into applications. MediaPipe Solutions enabled us to accelerate the creation of SAHLA by providing pre-built models and tools that could be easily plugged into our application.

By using MediaPipe, we customized its solutions to align with the specific needs of SAHLA, such as recognizing Arabic Sign Language (ArSL) gestures and ensuring cross-platform compatibility. As part of the MediaPipe open-source project, we had the flexibility to further tailor its codebase, adapting it to enhance the accuracy and efficiency of gesture recognition in our application. The MediaPipe Solutions suite offered several components that were instrumental in building the functionality of SAHLA, making it an ideal choice for our project.

The MediaPipe Solutions suite includes the following:

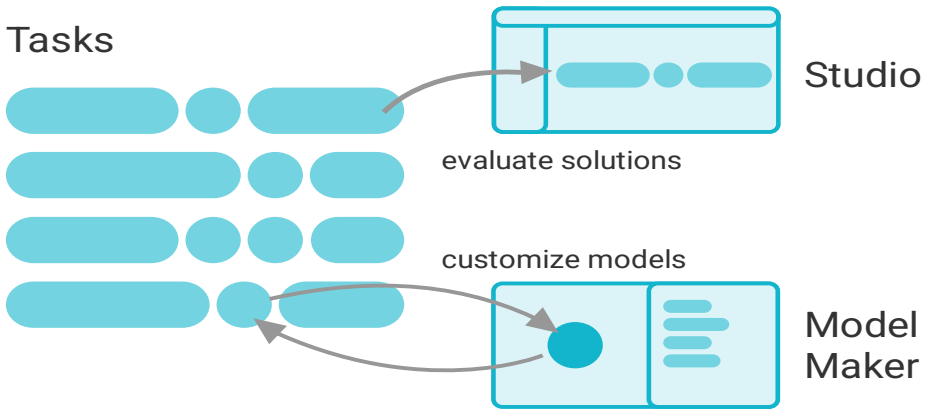


Figure 6 : MediaPipe Overview Diagram

1 These libraries and resources provide the core functionality for each MediaPipe

2 Solution:

- **MediaPipe Tasks**: Cross-platform APIs and libraries allowed us to seamlessly deploy gesture recognition capabilities within SAHLA.
- **MediaPipe Models**: Pre-trained, ready-to-run models for use with each solution.

3 These tools let you customize and evaluate solutions:

- **MediaPipe Model Maker**: Customize models for solutions with our data. ensuring accurate gesture recognition specific to our application
- **MediaPipe Studio**: Visualize, evaluate, and benchmark solutions in your browser. To make sure they meet our needs in SAHLA

4.2.2 MediaPipe Tasks

MediaPipe Tasks provides the core programming interface of the MediaPipe Solutions suite, including a set of libraries for deploying innovative ML solutions onto devices with a minimum of code. It supports multiple platforms, including Android, Web / JavaScript, Python, and support for iOS is coming soon. The availability of Python, in particular, was highly

beneficial to our project, as it allowed us to leverage its simplicity and versatility for efficient implementation of gesture recognition.

- **Easy to use, well-defined cross-platform APIs:**
  Run ML Inferences with just 5 lines of code. Use the powerful and easy-to-use solution APIs in MediaPipe Tasks as building blocks to build our own ML features.
- **Customizable solutions:**
  You can leverage all benefits MediaPipe Tasks provide, and easily customize it using models built with your own data via Model Maker. For example, you can create a model that recognizes the custom gestures you defined using the Model Maker GestureRecognizer API, and deploy the model onto desired platforms using the Tasks GestureRecognizer API. Which helped us tremendously in implementing our ArsL dataset.
- **High performance ML pipelines:**
  Typical on-device ML solutions combine multiple ML and non-ML blocks, slowing performance. MediaPipe Tasks provides optimized ML pipelines with end-to-end acceleration on CPU, GPU, and TPU to meet the needs of real time on-device use cases.

4.2.3 MediaPipe Model Maker

MediaPipe Model Maker is a tool for customizing existing machine learning (ML) models to work with your data and applications. We used this tool as a faster alternative to building and training a new ML model. Model Maker uses an ML training technique called transfer learning which retrains existing models with new data. This technique re-uses a significant portion of the existing model logic, which means training takes less time than training a new model and can be done with less data.

Model Maker works on various types of models including object detection, gesture recognition, or classifiers for images, text, or audio data. The tool retrains models by removing the last few layers of the model that classify data into specific categories and rebuilds those layers using new data you provide. Model Maker also supports some option to fine tune model layers to improve accuracy and performance.

Retraining a model using Model Maker generally makes the model smaller, particularly if you retrain the new model to recognize fewer things. This means you can use Model Maker to create more focused models that work better for your application. The tool can also help you

apply ML techniques like quantization, so your model uses less resources and runs more efficiently.

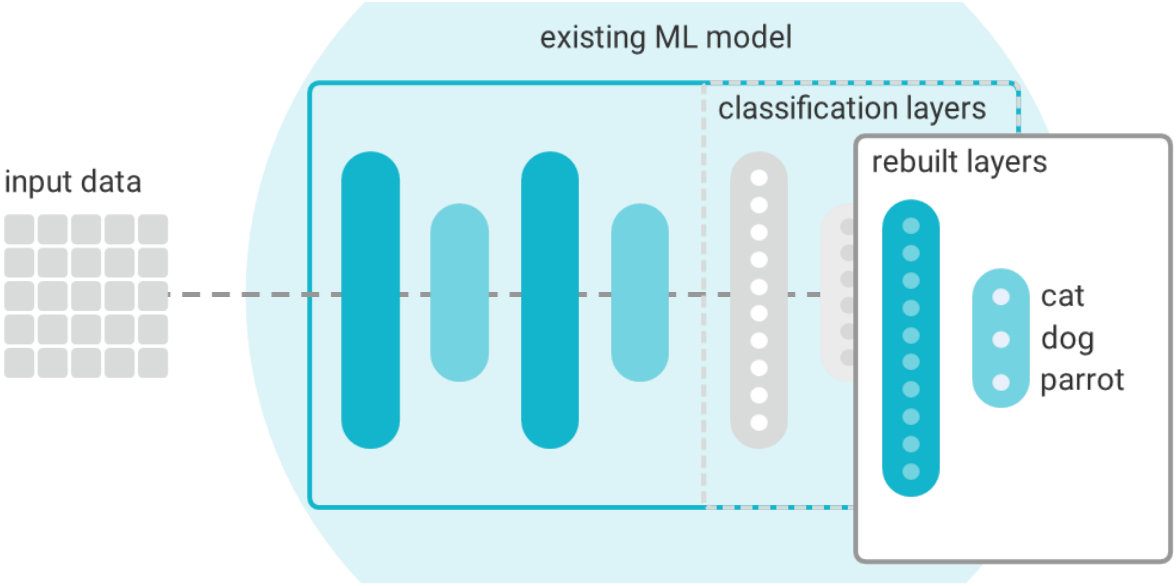


Figure 7: Model Maker Rebuilds Final Layers with New Data

Training data requirements:

Model Maker allows you to retrain a model with significantly less data than training a new model. When retraining a model with new data, you should aim to have approximately 100 data samples for each trained class. For our project, we ensured the robustness of our model by including more than 200 images for each letter of the Arabic alphabet, as well as over 200 images in the "none" folder to account for irrelevant gestures or noise. Depending on your application, you may be able to retrain a useful model with even less data per category, although a larger dataset generally improves the accuracy of your model. When creating your training dataset, remember that your training data gets split up during the retraining process, typically 80% for training, 10% for testing, and the remainder for validation. The ability to retrain models with minimal data was particularly advantageous for our project. It allowed us to fine-tune existing models efficiently while using Python, enabling faster experimentation and improved model accuracy with limited data resources.

Limitations of customization:

Since the retraining process removes the previous classification layers, the resulting model can only recognize items, or classes, provided in the new data. If the old model was trained to recognize 30 different item classes, and you use Model Maker to retrain with data for 10 different items, the resulting model is only able to recognize those 10 new items.

Retraining a model with Model Maker cannot change what the original ML model was built to solve, even if those jobs are similar. For example, you can't use the tool to make an image classification model perform object detection, even though those tasks share some similarity.

To combat the limitations of Model Maker, We tried to ensure complete class coverage in the retraining process by compiling an inclusive library of images for each Arabic alphabet letter and for non-gesture data, while also selecting a base model pre-trained for image classification to align with our project's objective of sign language recognition, maximizing the model's adaptability within its architectural constraints.

4.2.4 MediaPipe studio

MediaPipe Studio is a web-based application for evaluating and customizing on-device ML models and pipelines for your applications. The app lets you quickly test MediaPipe solutions in your browser with your own data, and your own customized ML models. Each solution demo also lets you experiment with model settings for the total number of results, minimum confidence threshold for reporting results, and more.

4.3 Gesture Recognition

After reviewing the features of MediaPipe and understanding the tasks it offers, we chose **Gesture Recognition** because it is the most suitable task for our idea.

4.3.1 Gesture recognition task guide

The MediaPipe Gesture Recognizer task lets you recognize hand gestures in real time and provides the recognized hand gesture results along with the landmarks of the hands detected. This functionality proved to be a good match for our project, as it enabled us to effectively implement real-time recognition of Arabic sign language gestures, ensuring accurate interpretation and seamless integration with our application.

This task operates on image data with a machine learning (ML) model and accepts either static data or a continuous stream. The task outputs hand landmarks in image coordinates, hand landmarks in world coordinates, handedness (left/right hand), and the hand gesture categories of multiple hands.

Task inputs	Task outputs
The Gesture Recognizer accepts an input of one of the following data types: • Still images • Decoded video frames • Live video feed	The Gesture Recognizer outputs the following results: • Categories of hand gestures • Handedness of detected hands • Landmarks of detected hands in image coordinates • Landmarks of detected hands in world coordinates

Table 2 : Gesture Recognition Inputs and outputs

4.3.2 Features

Input image processing: Processing includes image rotation, resizing, normalization and color space conversion.

Score threshold: Filter results based on prediction scores.

Label allowlist and denylist: Specify the gesture categories recognized by the model.

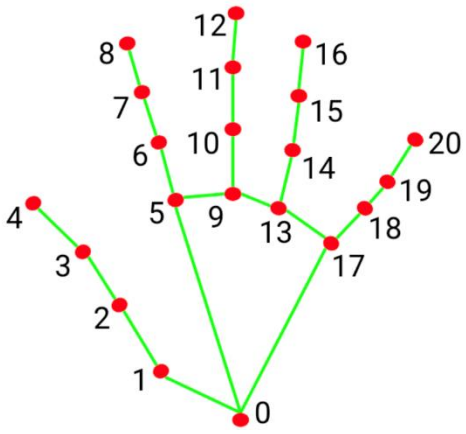
4.3.3 Models

3.1 The Gesture Recognizer uses a model bundle with two pre-packaged model bundles: a hand landmark model bundle and a gesture classification model bundle. The landmark model detects the presence of hands and hand geometry, and the gesture recognition model recognizes gestures based on hand geometry.

1. Hand landmark model bundle

The hand landmark model bundle detects the key point localization of 21 hand-knuckle coordinates within the detected hand regions. The model was trained on approximately 30K real-world images, as well as several rendered synthetic hand models imposed over various backgrounds.

See the definition of the 21 landmarks below:



0. WRIST
1. THUMB_CMC
2. THUMB_MCP
3. THUMB_IP
4. THUMB_TIP
5. INDEX_FINGER_MCP
6. INDEX_FINGER_PIP
7. INDEX_FINGER_DIP
8. INDEX_FINGER_TIP
9. MIDDLE_FINGER_MCP
10. MIDDLE_FINGER_PIP
11. MIDDLE_FINGER_DIP
12. MIDDLE_FINGER_TIP
13. RING_FINGER_MCP
14. RING_FINGER_PIP
15. RING_FINGER_DIP
16. RING_FINGER_TIP
17. PINKY_MCP
18. PINKY_PIP
19. PINKY_DIP
20. PINKY_TIP

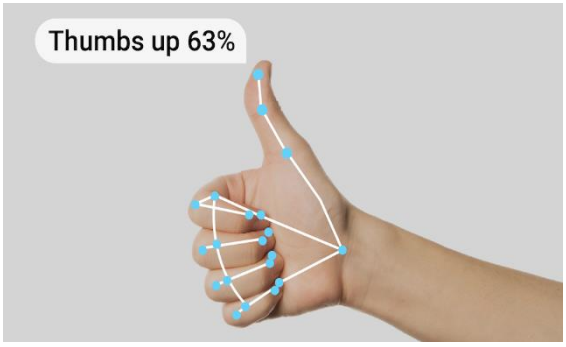
Figure 8 : Hand landmark model bundle

The hand land marker model bundle contains palm detection model and hand landmarks detection model. Palm detection model localizes the region of hands from the whole input image, and the hand landmarks detection model finds the landmarks on the cropped hand image defined by the palm detection model.

Since palm detection model is much more time consuming, in Video mode or Live stream mode, Gesture Recognizer uses bounding box defined by the detected hand landmarks in the current frame to localize the region of hands in the next frame. This reduces the times Gesture Recognizer triggering palm detection model. Only when the hand landmarks model could no longer identify enough required number of hands presence, or the hand tracking fails, palm detection model is invoked to relocalize the hands.

2. Gesture classification model bundle

The Gesture classification model bundle can recognize these common hand gestures:



1. Unrecognized gesture, label: Unknown
2. Closed fist, label: Closed_Fist
3. Open palm, label: Open_Palm
4. Pointing up, label: Pointing_Up
5. Thumbs down, label: Thumb_Down
6. Thumbs up, label: Thumb_Up
7. Victory, label: Victory
8. Love, label: I Love You

If the model detects hands but does not recognize a gesture, the gesture recognizer returns a result of "None". If the model does not detect hands, the gesture recognizer returns empty.

The gesture classification model bundle contains two step neural network pipeline with a gesture embedding model followed by a gesture classification model.

The gesture embedding model encodes the image features into a feature vector, and the classification model is a lightweight gesture classifier taking the feature vector as input. The provided gesture classification model bundle contains the canned gestures classifier, which detects the seven common hand gestures introduced above.

This structure was a suitable starting point for our project, as it allowed us to leverage the canned gestures while extending the model bundle to recognize additional gestures specific to Arabic sign language. By training our own custom gesture classifier, we adapted the recognizer to meet the unique needs of our application.

Gesture Recognizer, with both canned gesture classifier and custom gesture classifier, prefers the custom gesture if both classifiers recognize the same gesture in their categories. If only one gesture classifier recognizes the gesture, Gesture Recognizer outputs the recognized gesture directly.

4.4 Hand Gesture Recognition Model Customization Guide

The objective was to customize a gesture recognition model by training it to recognize 32 unique gestures:

- all Arabic letter gestures
- some other symbols.

Finally, we will test it to ensure its quality and performance.

4.4.1 Training Our Model

1. Preparing Dataset:

Applying transformations: like resizing, normalization, augmentation, or other adjustments required for consistent input to the model.

Landmark Extraction: Identifies key points (e.g., fingers, palm) in the image for gesture recognition.

Data Split: The data was divided into:

- **Training Set (80%):** Used to teach the model patterns of each gesture.
- **Validation Set (10%):** Used to tune the model during training.
- **Test Set (10%):** Used to assess the model's performance on unseen data.

1. **Model Design:**
   We used gesture recognition model, so it only retrained the final layers to adjust the specific gesture categories in our dataset.

2. **Training Configuration:**

   Epochs: The model was trained for 20 epochs to ensure sufficient learning without overfitting.

   Batch Size: A small batch size of 8 was used for faster updates and efficient memory usage.

   Loss Function: Categorical Cross-Entropy to optimize multi-class classification performance.

   Optimizer: Adam optimizers to dynamically adjust learning rates during training.

Evaluation Metric: Categorical accuracy was tracked to monitor the model's performance.

3. **Evaluation:**

- **Train accuracy**: 91.32%
- **Validation accuracy:** 92.13%
- **Train Accuracy vs Validation Accuracy**

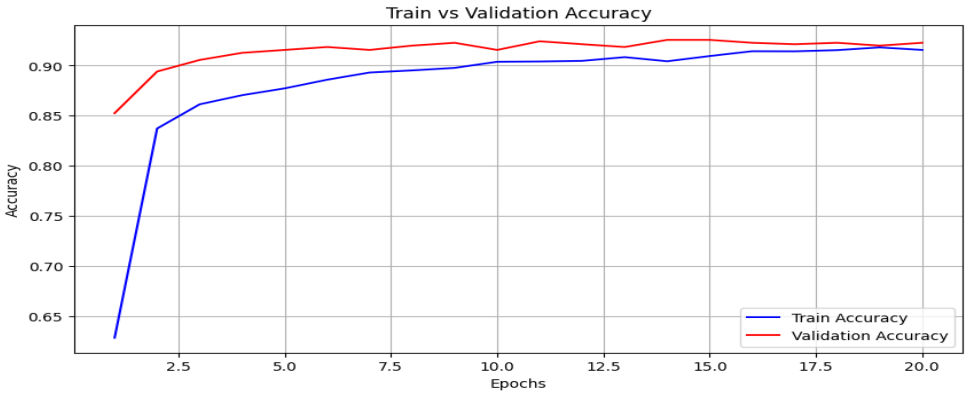


Figure 9: Training vs Validation Accuracy Across Epochs

- **Train Loss vs Validation Loss**

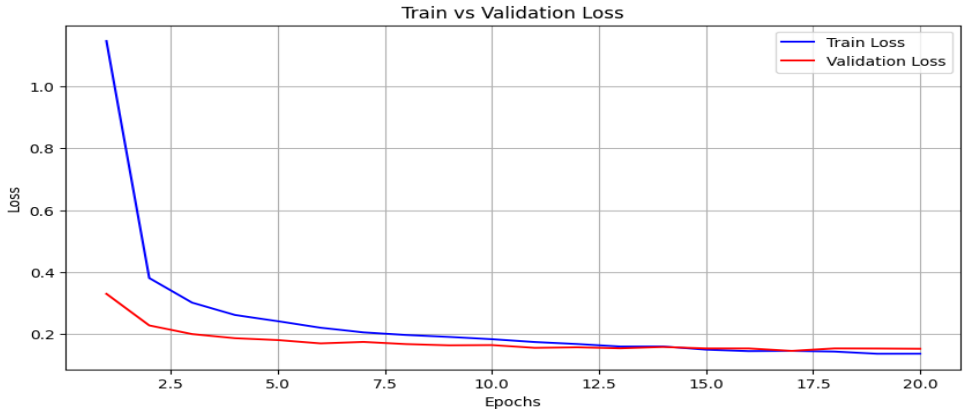


Figure 10: Training vs Validation Accuracy Across Epochs

4. **Model Deployment:**
   The trained model was exported to **TensorFlow Lite** for deployment on mobile and edge devices. This lightweight format ensures fast and efficient gesture recognition in real-world applications, such as mobile apps for Arabic sign language interpretation

4.4.2 Testing Our Model

After completing the model's training, we must assess its quality and evaluate how effectively it handles various data.

1. **Testing Our Model Using the Unseen Part of Our Dataset (Test Set)**
   The trained model was tested on the test dataset to measure its effectiveness:
   - **Test Accuracy:** Achieved approximately **90.71%**, indicating high reliability in recognizing Arabic sign language gestures.
   - **Test Loss**: Low, showing that the model successfully captured the patterns without significant errors.
2. **Offline Testing with Custom Dataset**
   Prepare the Dataset:
   - We have created a new dataset which contains images which express sign for each Arabic letter and some other symbols to check the performance of our model.

   Testing Method:

   - The model will predict the class to which each image belongs. To verify the results, we will display each image, overlay hand landmarks, and label them with the predicted class and the model's confidence in the prediction.
   - We will evaluate the accuracy of the model by comparing each label with the true result.

Output:

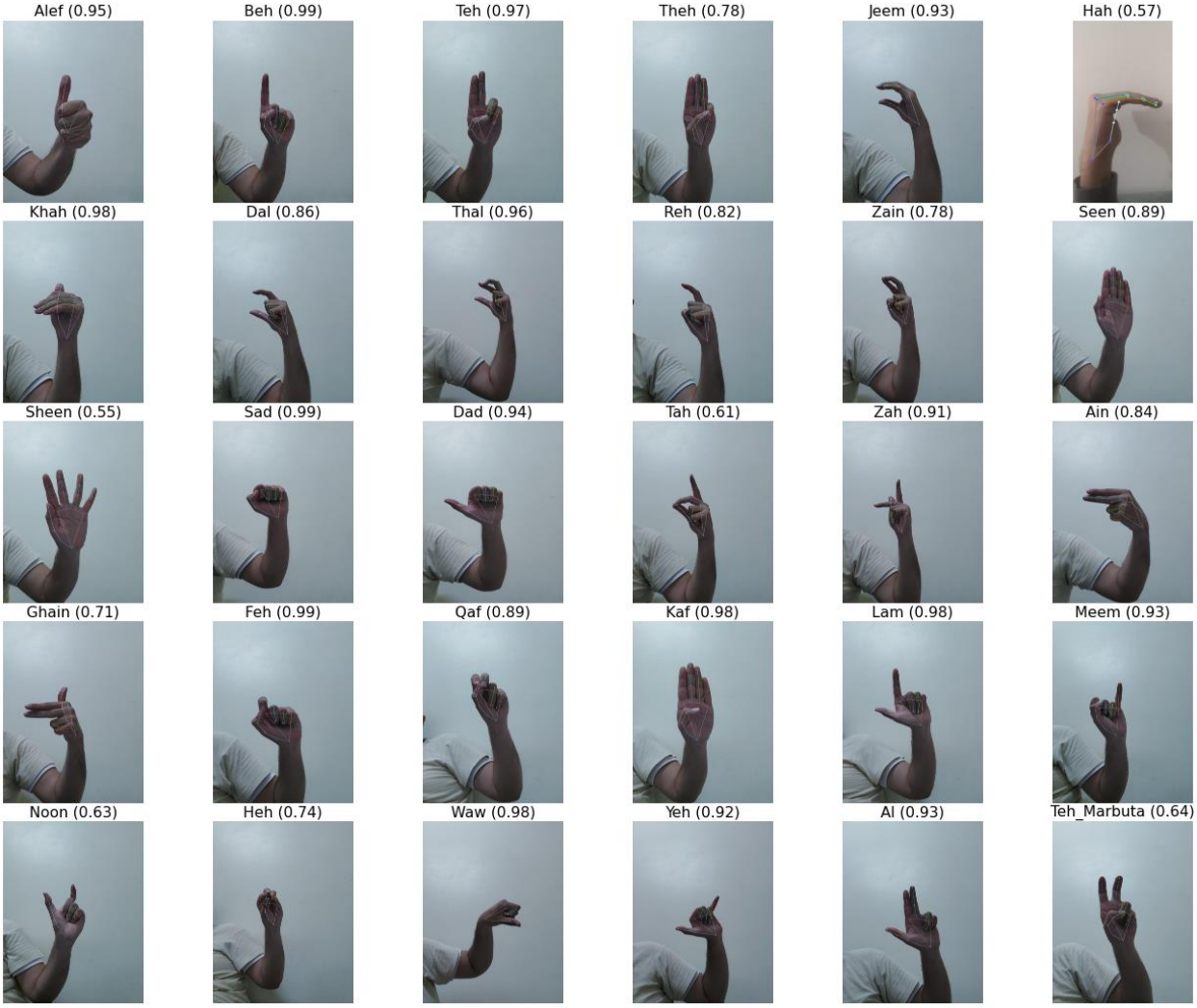


Figure 11 : Arabic Sign Language Hand Gesture Recognition with Confidence Scores

Evaluating Model Performance:

- The model accurately predicted all the gestures included in the dataset with high confidence, indicating that it is performing with the required efficiency.

3. Testing with MediaPipe Studio

- We will upload our model to MediaPipe Studio and test it by feeding input data, such as images into the pipeline. MediaPipe Studio will process the data using the model, generate the output, and visualize it.

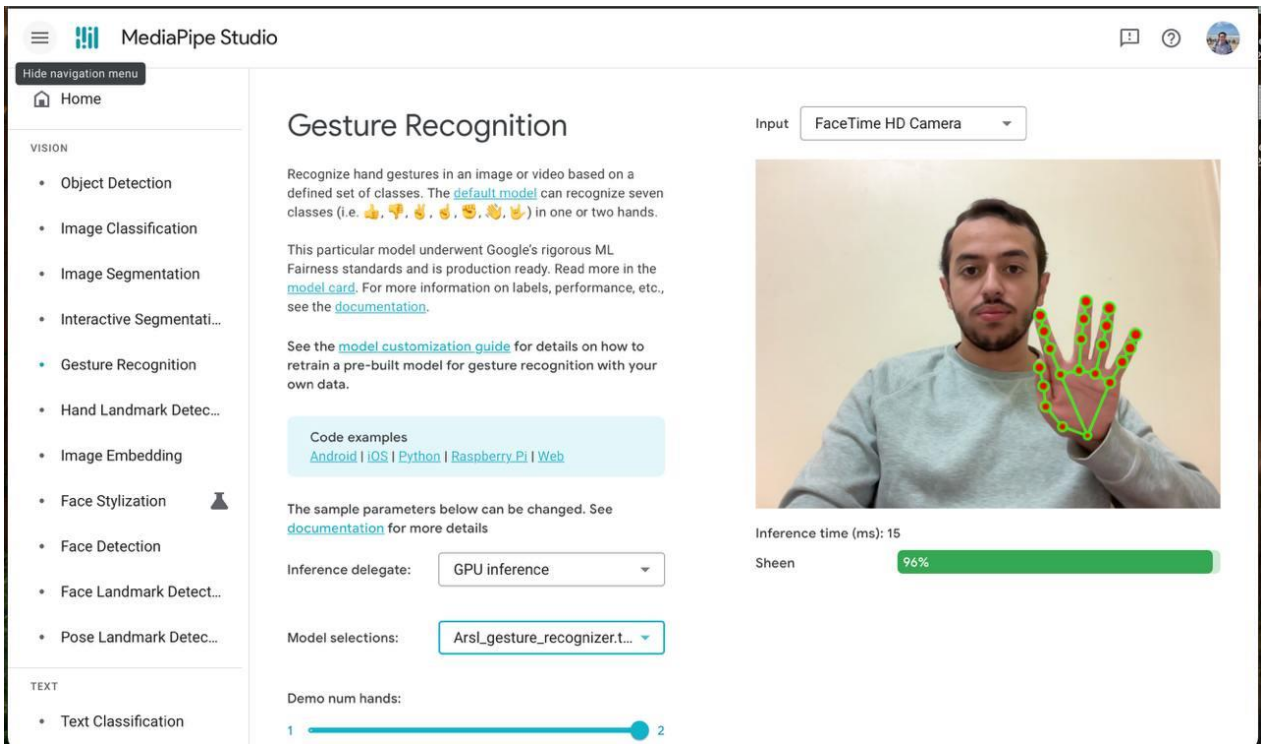


Figure 12: MediaPipe Gesture Recognition with Real-Time Tracking

Evaluate Performance:

- After evaluating the model's predictions and confidence scores, we confirmed that the performance of our model was excellent.

4.5 Integration between MediaPipe and applications.

As demonstrated in the model's test, the system processes an image and accurately identifies the corresponding letter that the image represents. But this raises an intriguing challenge: how can we seamlessly integrate these capabilities to transform a continuous stream of sign language gestures in a video into coherent Arabic text?

4.5.1 Main Implementation Steps

1. **Setting up inputs:**
    - Defined paths for the input video
2. **Gesture Recognition Setup:**
    - Increasing the number of consecutive frames required to confirm a gesture to **10** frames.
    - Used our customized gesture recognition model to analyze video frames and identify gestures.
    - Maps recognized gesture labels in English ("**Alef**") to their corresponding Arabic letters ("أ").
    - Returns "Letter not found" if a gesture is unrecognized.
    - If the model predicts ("**Laa**") we will map it to a space between words (" ").
3. **Video Processing Workflow:**
    - **Load Video**:
        - Opened the video using **OpenCV**.
    - **Frame Processing:**
        - Processed each frame to detect hand gestures.
        - Overlaid detected Arabic letters and constructed sentences.
        - Drew hand landmarks for visualization.
    - **Save Processed Frames:**
        - Combined frames to create an output video
4. **Text Rendering:**
    - Used an Arabic-compatible font (**Amiri-Regular.ttf**) to write Arabic sentences on frames.
    - Ensured proper rendering for right-to-left text direction.
5. **Output Creation:**
    - Generated the processed video with gestures and Arabic text.
    - Saved the translated Arabic sentence in a text file.

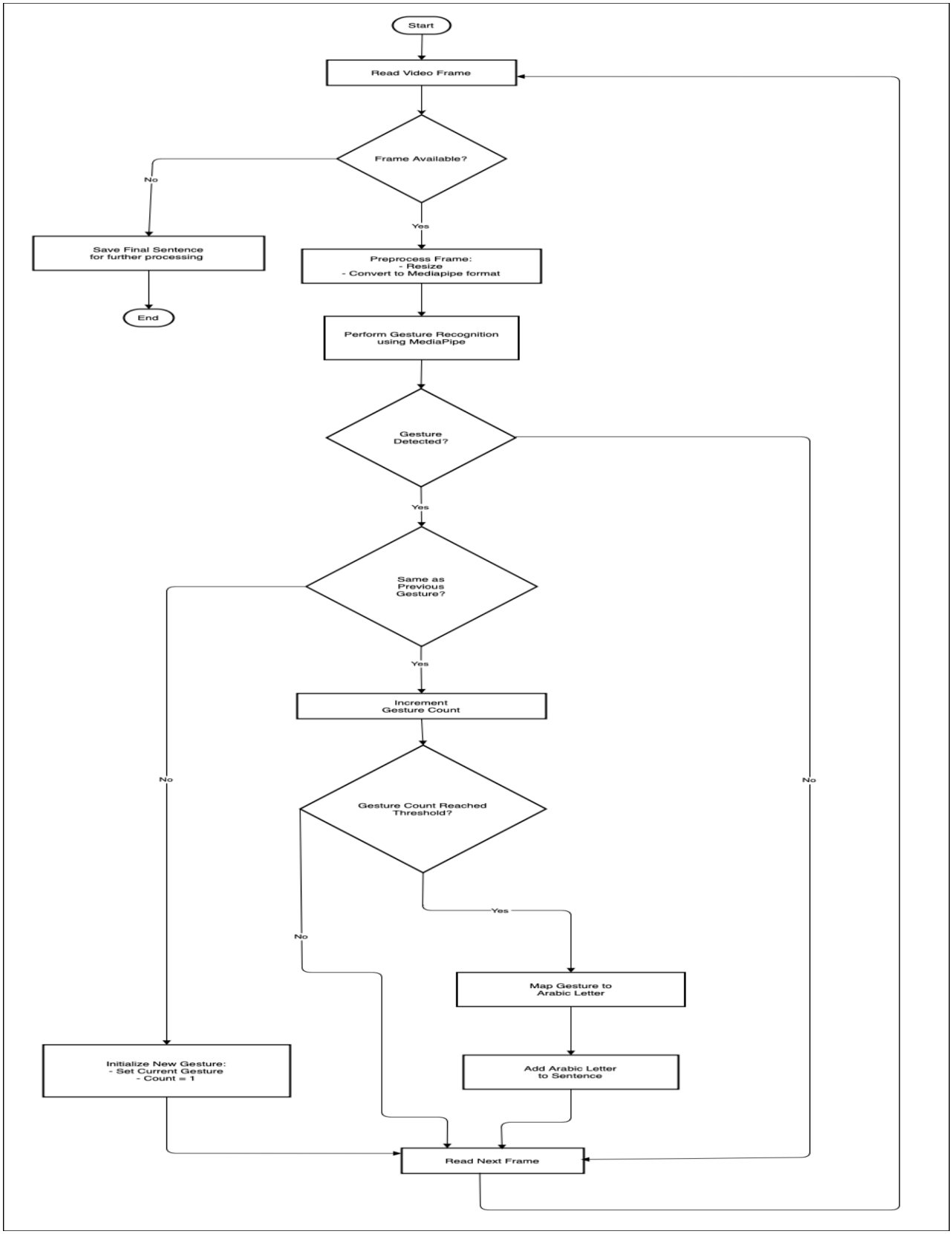


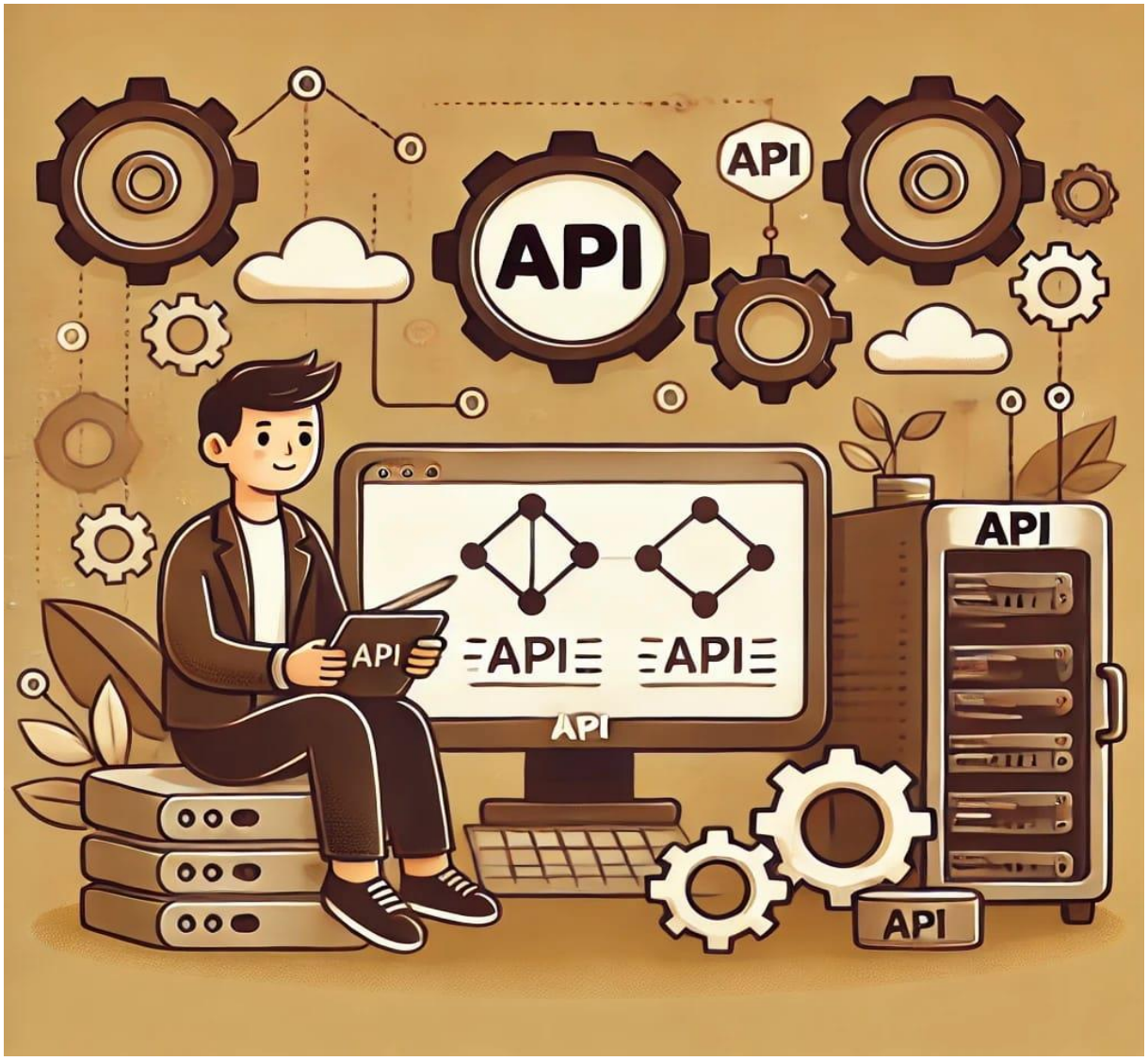
Figure 13 : MediaPipe Activity Diagram

4.6 Conclusion

This module of our application successfully translated sign language from a video into a meaningful Arabic sentence, showcasing the powerful integration of machine learning and natural language processing in bridging communication gaps for the deaf and mute community. The processed video and accompanying text file provide a comprehensive demonstration of the translation results, clearly illustrating how the system interprets gestures and converts them into understandable Arabic text.

Future enhancements could focus on incorporating real-time recognition capabilities, which would significantly increase the practicality and usability of the application in dynamic, real-world scenarios. Additionally, in the next chapter, we will explore in greater detail how we seamlessly integrated our model with speech-to-text and text-to-speech technologies. This integration allows us to transform the Arabic sentences generated by MediaPipe into spoken words, paving the way for an intuitive and interactive communication experience that combines visual gestures and auditory feedback.

CHAPTER 5: Integrating Google Speech APIs



5.1 Introduction

5.1.1 The Role of Speech Technologies in Modern Applications

In recent years, speech technologies have become integral to a wide range of applications, enabling richer and more interactive user experiences. Two of the most essential speech technologies are Text-to-Speech (TTS) and Speech-to-Text (STT), which allow for the conversion of written text into spoken audio and the transcription of spoken language into text, respectively. These technologies not only make systems more accessible but also improve their interactivity by allowing users to engage through voice commands and feedback.

For our application, we leverage Google Cloud's Text-to-Speech API and Speech-to-Text API, two powerful cloud-based services that provide advanced speech synthesis and recognition capabilities. These APIs are central to our project's functionality, enabling seamless interactions where text is converted to natural-sounding speech and voice input is transcribed into text that can be processed further by the system.

5.1.2Why Google Cloud?

We chose Google Cloud for this project due to its advanced, scalable, and highly reliable speech technologies. The Text-to-Speech and Speech-to-Text APIs provided by Google Cloud offer several advantages:

1. **Scalability:** Google Cloud services are designed to handle large volumes of requests, which is ideal for applications that may need to scale with growing demand. Whether our system serves a small user base or experiences rapid growth, Google Cloud can efficiently manage any load our application requires.

2. **High-Quality Speech Recognition and Synthesis:** the Speech-to-Text API ensures precise transcription of spoken language into written text, while the Text-to-Speech API generates clear, natural-sounding speech. Both APIs, powered by Google's advanced machine learning models, enable our application to seamlessly process Arabic text and deliver accurate and expressive voice output. This integration ensures that SAHLA can accommodate diverse accents and speech patterns, enhancing

accessibility and user experience for the deaf and mute community.

3. **Customization Options:** Google Cloud allows for a high degree of customization. For Text-to-Speech, users can select different voices, adjust speech parameters such as pitch and rate, and even choose accents and genders to personalize the speech output. The Speech-to-Text API supports a wide array of languages and dialects, which makes it flexible and a perfect option for our application.

4. **Ease of Integration:** With comprehensive documentation and SDKs for multiple programming languages, Google Cloud makes it easy to integrate the Text-to-Speech and Speech-to-Text APIs into our system. This simplicity and ease of use allow us to focus more on the functionality of our application rather than the technical details of implementing speech technology.

5.1.3 How We Are Using Google Cloud's Speech Technologies

In our application, Google Cloud's Text-to-Speech API is seamlessly integrated to convert the text input, received from MediaPipe , into lifelike speech. For example, when the system needs to communicate a translated message or provide feedback, the text output from MediaPipe is sent to the Text-to-Speech API. This API processes the text and generates spoken audio, which is then played back to the user. This functionality greatly enhances the interactivity and accessibility of our application, delivering a more engaging and effective user experience.

Similarly, we use Google Cloud's Speech-to-Text API to transcribe audio input into text. This feature is crucial when the system needs to recognize and process spoken commands or transcribe meetings, interviews, or any other form of speech. The Speech-to-Text API processes the audio data and returns the corresponding text, allowing us to interpret user input more effectively.

For example, the system may listen for specific voice commands from the user and then process the speech to take appropriate actions based on the recognized text. Whether it is adding an event to a calendar or communicating through sign language, the ability to convert speech into text is vital for making the system more intuitive and responsive. bridging the gap between spoken language and sign language.

5.1.4 The Role of Google Cloud in Enhancing User Experience

The integration of Google Cloud's Speech-to-Text and Text-to-Speech APIs is pivotal to the user experience in our project. By using Text-to-Speech, we are able to create a more engaging and dynamic interaction, where users receive spoken feedback in a natural voice. On the other hand, the Speech-to-Text API allows the system to understand and process spoken commands, making the interaction seamless and intuitive.

By utilizing these cloud-based technologies, we eliminate the need to build complex infrastructure and machine learning models in the application. Instead, we can rely on Google Cloud's powerful APIs to handle the heavy lifting, allowing us to focus on the unique aspects of our project and its functionality. Moreover, the scalability and reliability of Google Cloud ensure that our system can grow with the demands of the users, maintaining performance and availability at all times.

In the following sections, we will take a deep dive into the technical aspects of how Google Cloud's Text-to-Speech and Speech-to-Text APIs were seamlessly integrated into our SAHLA application. This exploration will cover the step-by-step process of incorporating these advanced APIs, highlighting the specific customization options we utilized to optimize them for our unique requirements. Additionally, we will shed light on the practical benefits these cloud-based speech technologies brought to our project, such as enhancing accessibility for the deaf and mute community and fostering real-time interaction between users. Finally, we will reflect on the broader impact of leveraging such cutting-edge technologies in modern applications, emphasizing their role in bridging communication gaps and driving innovation.

5.2 **Google Cloud Speech-to-Text and Text-to-Speech APIs**

Google Cloud offers powerful APIs for Speech-to-Text and Text-to-Speech that facilitate seamless integration of speech recognition and synthesis capabilities into applications. These APIs are a cornerstone of modern voice-enabled applications, enabling real-time transcription and natural-sounding speech generation. In our application, these Google Cloud services play a pivotal role in creating an interactive, voice-responsive user experience.

5.2.1 Overview of Speech-to-Text API

The Google Cloud Speech-to-Text API converts audio into text by leveraging advanced machine learning models. This API supports multiple languages and can transcribe audio with high accuracy, even in noisy environments. The primary features of the Speech-to-Text API that benefit our project include:

- **Real-Time Transcription:** The API can transcribe live speech, enabling users to interact with the system in real-time, which is essential for voice command functionalities.
- **Speaker Diarization:** It can differentiate between multiple speakers, which adds flexibility when the project requires transcription of conversations or multi-speaker interactions.
- **Punctuation and Formatting:** The API intelligently inserts punctuation marks into transcribed text, improving readability and making it easier for further processing.

In our innovative project, the integration of Google Cloud's Speech-to-Text API plays a pivotal role in processing user commands and input efficiently and in real-time. Specifically, when a user provides voice commands, the audio input is seamlessly transmitted to the Speech-to-Text service, which then processes the audio and returns a transcribed text version of the spoken words. This text serves as the foundation for triggering specific system actions, allowing the application to respond dynamically to the user's needs. The accuracy of speech recognition, combined with the system's ability to handle a wide range of accents and dialects, is critical in delivering a smooth, intuitive, and accessible user experience. By leveraging this powerful technology, SAHLA ensures that users can interact effortlessly, making it a reliable tool for bridging communication gaps and facilitating real-time functionality.

5.2.2 Overview of Text-to-Speech API

The Text-to-Speech API from Google Cloud converts written text into natural-sounding speech. With a wide variety of voices, languages, and accents available, it offers a high level of customization for applications like ours that require voice feedback. The key features of the Text-to-Speech API that are leveraged in our project include:

- **Voice Customization:** The API allows the selection of different voices (male or female), language accents, and even the tone or pitch of the speech. For example, in

our project, the system generates voice feedback in the user's preferred language or accent, improving user engagement.

- **WaveNet Voices:** One of the most remarkable aspects of the Text-to-Speech API is the WaveNet technology, which uses deep learning to produce highly natural-sounding voices. The voices generated using this technology are more expressive, with human-like nuances such as appropriate pauses, inflections, and tone changes.
- **Audio Configuration:** The API provides settings for controlling the speed, pitch, and volume of the speech, giving further flexibility in adapting the voice output to fit the context of the application.

The Text-to-Speech API is used to generate audible responses from the system. For example, when a user interacts with the application, the system processes the input and provides verbal feedback. This feedback is generated using the Text-to-Speech service, which converts predefined strings into natural-sounding speech that is then played back to the user.

By combining both Speech-to-Text and Text-to-Speech APIs from Google Cloud, our project achieves seamless, interactive voice communication. The Speech-to-Text API enables accurate recognition of user input, while the Text-to-Speech API ensures that the system responds in a natural and personalized manner. This integration allows the project to provide an intuitive voice interface, greatly enhancing the overall user experience.

5.3 Text-to-Speech Implementation

In our project, the Google Cloud Text-to-Speech API is the core component that provides users with an interactive, voice-based experience. This section outlines how the Text-to-Speech service is integrated into our system, from setting up the API to generating speech outputs based on user input or system responses.

5.3.1 Overview of Text-to-Speech in the Project

The integration of Google Cloud's Text-to-Speech API into our project, SAHLA, serves a vital purpose:

converting textual information generated by the system into natural, audible speech. This functionality enables the application to provide voice feedback that enhances communication

between the system and its users. For instance, whenever a user performs an action or when a specific system event occurs, the Text-to-Speech API generates clear, human-like speech to inform the user of the outcome or deliver relevant information. By incorporating this feature, SAHLA creates a more immersive and accessible experience, , the use of the Text-to-Speech API ensures the application remains responsive, engaging, and user-friendly, further reinforcing its role as a reliable and innovative communication tool.

5.3.2 Architecture Diagram Explanation

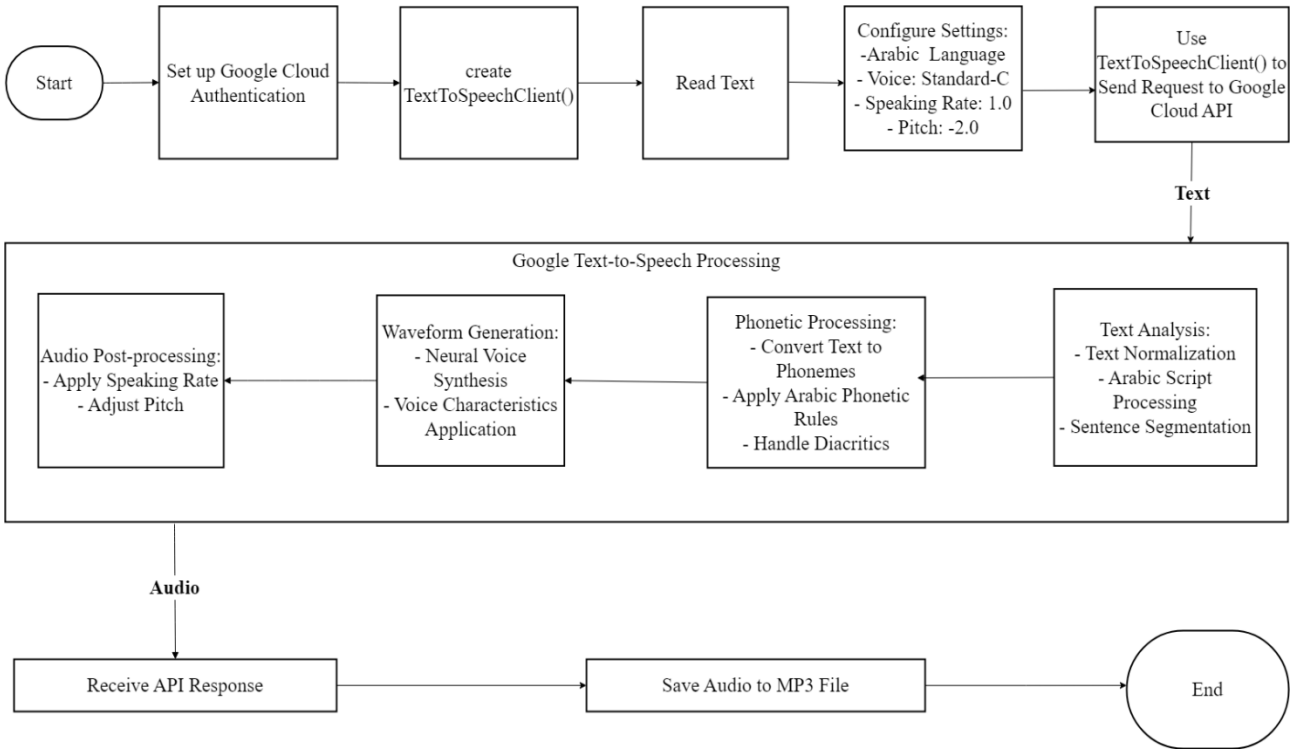


Figure 14 : TTs activity Diagram Explanation

1. **Set Up Google Cloud Authentication**:
   The process begins by establishing secure access to Google Cloud services. Authentication ensures that all subsequent interactions with the API are authorized and protected.
2. **Initialize the Text-to-Speech Client**:
   The application creates an instance of the `TextToSpeechClient()` class provided by Google Cloud. This client serves as the interface for sending text-to-speech requests.

3. **Input Text for Conversion**:
   The input text coming form mediapipe gesture recoginition model which serves as the basis for generating speech.
4. **Configure Text-to-Speech Settings**:
   Key voice and audio parameters are specified to customize the speech output. These include:
   - **Language**: Arabic is selected for the output speech.
   - **Voice Model**: The "Standard-C" voice is chosen for this use case.
   - **Speaking Rate**: Set to 1.0 for a natural pace.
   - **Pitch**: Adjusted to -2.0 for a slightly deeper tone.
5. **Send Request to Google Cloud API**:
   The configured `TextToSpeechClient()` sends the text input, along with the specified settings, to the Google Text-to-Speech API for processing.
6. **Google Text-to-Speech Processing**:
   The API processes the request through several stages:
   - **Text Analysis**: The text is normalized, ensuring consistency in output, and segmented into appropriate sentences. Arabic script is handled with rules specific to the language.
   - **Phonetic Processing**: The text is converted into phonemes while applying Arabic-specific phonetic rules, including diacritic handling for accurate pronunciation.
   - **Waveform Generation**: Neural voice synthesis creates a high-quality audio waveform, incorporating the requested voice characteristics.
7. **Audio Post-Processing**:
   Additional adjustments are made to fine-tune the audio output, such as applying the desired speaking rate and pitch.
8. **Receive API Response**:
   The API returns the generated audio content in a format ready use. This is a base64-encoded string containing the audio file.
9. **Save Audio to MP3 File**:
   The application decodes the received audio content and saves it as an MP3 file, making it accessible to be sent to the application interface.

5.3.3 Key Steps in Text-to-Speech Integration

The integration of Google Cloud's Text-to-Speech API in our application follows several key steps to ensure smooth operation:

1. **API Setup and Authentication:**
   To use Google Cloud Text-to-Speech, an authentication process is required. In our project, we utilized the

```
gcloud auth print-access-token
```

command to retrieve an access token for the service, ensuring secure communication between our application and the API. Proper configuration of API access is essential to ensure that requests are authenticated and authorized.

2. **Defining Input Text for Speech Generation:**

   The Text-to-Speech API requires input in the form of text that it will convert to speech. In our project, the text for conversion is typically generated dynamically based on user actions or system status. For example, when the system successfully adds an event to the user's calendar, the system generates the text string “I’ve added the event to your calendar” and passes it to the Text-to-Speech service.

3. **Choosing Voice Parameters:**

   One of the critical aspects of Text-to-Speech is selecting the voice that will speak the text. Our project makes use of the wide variety of voice options provided by the API, including different languages, accents, and genders. For example, we choose voices such as "en-GB-Standard-A" for a British accent, or "en-US-Wavenet-D" for a more natural-sounding, WaveNet-generated voice. Additionally, we specify the gender of the voice (female or male) based on user preferences or project requirements.

4. **Audio Configuration:**

   The audio output configuration plays a significant role in controlling how the speech sounds. In our project, we set parameters like audio encoding, which determines the file format of the generated speech (e.g., MP3), and other characteristics such as speech rate and pitch. These settings allow for further customization of the voice output, ensuring that it fits the tone and pace needed for the specific user interaction.

5. **Generating Speech and Audio Output:**

   Once the text and voice parameters are defined, the system sends a request to the Text-to-Speech API to synthesize the speech. The API returns the audio content in the form of a base64-encoded string, which is then decoded into a playable audio file. This audio file is then played back to the user as feedback, providing a seamless and natural interaction.

6. **WaveNet Voices for Enhanced Naturalness:**

   In our project, we make use of WaveNet voices offered by Google Cloud for an enhanced, more human-like quality to the speech output. WaveNet technology uses deep learning models trained on large datasets of human speech, allowing for more nuanced pronunciation, emphasis, and natural-sounding pauses. This was particularly

useful in enhancing the user experience by making the feedback sound more lifelike and less robotic.

Challenges and Considerations

While implementing Text-to-Speech was relatively straightforward, some challenges arose:

- **Voice Selection:**
  Selecting the appropriate voice for different contexts required careful consideration. The tone, pitch, and accent of the voice needed to align with the application's purpose and the user's expectations. We had to ensure the voices sounded natural and appropriate for the interaction.

- **Audio File Size and Playback:**
  Since the speech output is generated as an audio file, considerations about the file size and quality were essential. Optimizing the size while maintaining high-quality audio was crucial to ensure smooth playback, especially in real-time interactions.

- **Real-Time Performance:**
  Generating and delivering speech in real time is crucial for maintaining an interactive user experience. We had to ensure that the system responded quickly to user input without noticeable delays in speech generation.

5.4 Speech-to-Text Implementation

The Google Cloud Speech-to-Text API serves as one of the cornerstones in the functionality of our SAHLA project, providing the critical capability of converting spoken language into written text. This feature is instrumental in enabling voice-based input from users, which significantly enhances the system's accessibility and intuitiveness. By processing audio data captured from the user, the Speech-to-Text API transcribes the spoken words into accurate text, allowing the system to interpret and act upon the input dynamically. This process not only supports seamless user interaction but also ensures that the application can respond to commands or queries effectively, making it an invaluable tool for bridging communication gaps.

In SAHLA, the integration of this API allows users to provide voice input, which is especially helpful in scenarios where typing or other forms of input may not be practical. For instance, a user could issue a voice command to trigger a specific action, and the API ensures that the spoken words are quickly and accurately translated into text that the system can process further. This section delves into the technical aspects of how the Speech-to-Text API was

integrated into our project, exploring its customization for handling diverse accents and speech patterns, and illustrating its versatility across various use cases. By leveraging this technology, SAHLA ensures a user-friendly experience that fosters interaction and opens up new possibilities for communication and engagement.

5.4.1 Overview of Speech-to-Text in the Project

The primary objective of implementing the Speech-to-Text API in the SAHLA app is to convert spoken language into text, enabling users to communicate naturally using their voice. The transcribed text is then used to generate sign language gestures, ensuring seamless interaction and accessibility for Deaf and Hard of Hearing individuals. This functionality bridges the communication gap and makes the application inclusive for a wider range of users.

5.4.2 Architecture Diagram Explanation

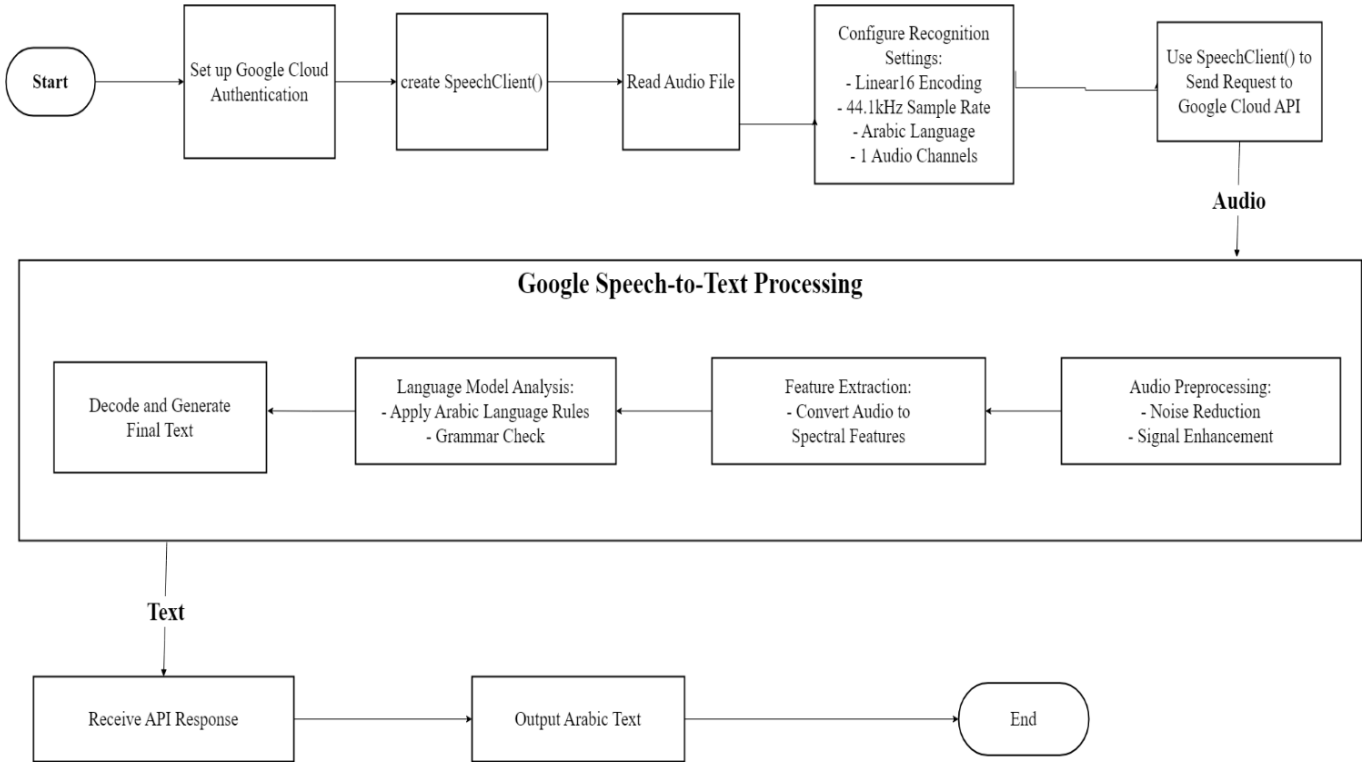


Figure 15 : STT Activity Diagram

1. **Set Up Google Cloud Authentication**:
   The process begins by securely authenticating with Google Cloud services, ensuring authorized access to the Speech-to-Text API.
2. **Initialize the Speech Client**:
   An instance of the `SpeechClient()` is created to act as the interface between the SAHLA app and Google's Speech-to-Text service.
3. **Read Audio File**:
   The user's audio input, recorded via the app, is read and prepared for processing. This input serves as the basis for generating the corresponding text.
4. **Configure Recognition Settings**:
   The recognition settings are configured to optimize the transcription process for Arabic speech. Key parameters include:
   - **Audio Encoding**: Linear16 for high-quality audio input.
   - • **Language**: Arabic is specified to transcribe speech in the chosen language.
   - • **Audio Channels**: Single-channel audio is used for compatbility with mobile phone input.
5. **Send Request to Google Cloud API**:
   Using the `SpeechClient()`, the app sends the audio file along with the configured settings to the Google Cloud Speech-to-Text API for transcription.
6. **Google Speech-to-Text Processing**:
   The API processes the audio input through several stages:

- **Audio Preprocessing**: Noise reduction and signal enhancement techniques are applied to ensure clear input for transcription.
- **Feature Extraction**: The audio is analyzed and converted into spectral features, facilitating accurate speech recognition.
- **Language Model Analysis**: Arabic language rules and grammar checks are applied to ensure accurate transcription.

7. **Decode and Generate Final Text**:
   The API decodes the processed data and generates the final text transcription of the user's speech.
8. • **Receive API Response**:
   The API returns the transcribed text, which represents the spoken input accurately.
9. • **Output Arabic Text**:
   The transcribed Arabic text is displayed or processed further within the SAHLA app. This text serves as the input for generating sign language gestures, completing the communication cycle.

5.4.3 Key Steps in Speech-to-Text Integration

The integration of the Speech-to-Text API into our system follows several important steps to ensure proper functionality:

1. **API Setup and Authentication:** As with the Text-to-Speech API, using Google Cloud's Speech-to-Text API requires setting up an account and authenticating requests. In our project, we made use of the

   ```
   gcloud auth print-access-token
   ```

   command to obtain an access token, allowing our system to securely interact with the API. This step ensures that only authorized applications can use the Speech-to-Text service.

2. **Capturing Audio Input:** The first step in utilizing Speech-to-Text is capturing the user's spoken input. In our project, we set up a microphone interface to record the user's speech. The audio data is captured in a format compatible with Google Cloud's API, typically as an audio file in formats such as WAV or FLAC.

3. **Sending Audio to the Speech-to-Text API:** After capturing the user's voice input, the next step is to send the audio data to the Speech-to-Text API. The system makes an API request to convert the audio file into text. In our project, this step involves encoding the audio file and sending it along with configuration details (such as language settings) to the API endpoint.

4. **Language and Model Configuration:** Google Cloud's Speech-to-Text API supports a variety of languages and dialects. In our project, we configure the API to transcribe speech in the user's preferred language, such as English (en-US) or other supported languages, depending on the context. Additionally, we select a transcription model optimized for specific use cases, such as video transcription, phone call transcription, or default speech recognition.

5. **Processing the Transcription Response:** Once the audio is processed, the API returns the transcribed text in response. The transcribed text can include not only the words spoken by the user but also additional features like speaker diarization (identifying different speakers) or punctuation, if applicable. In our project, this text is then used to determine the next action the system should take, whether it's scheduling an event, answering a question, or performing a search.

6. **Real-Time Speech Recognition:** For interactive and real-time voice-based applications, we enabled the real-time speech recognition feature of the Speech-to-Text API. This allows the system to continuously listen and transcribe speech while the user is talking, making the interaction feel more natural and responsive. As the user speaks, the transcriptions are updated, and the system can take immediate actions based on the input.

Challenges and Considerations

While integrating the Speech-to-Text API, we encountered several challenges and had to make considerations to ensure optimal performance for our application:

- **Accuracy of Transcription:**
  Speech recognition accuracy can be affected by background noise, accents, and the clarity of speech. To address this, we ensured that the audio was captured in quiet environments and configured the API for the appropriate language model to improve accuracy.

- **Handling Different Accents and Dialects:**
  Since the Speech-to-Text API supports multiple languages and dialects, we had to account for the variety of accents that users might have. Choosing the right language model and dialect for transcription was crucial to ensuring the accuracy of transcriptions across different regions.

- **Latency in Real-Time Transcription:**
  In real-time applications, reducing latency is critical to provide a seamless user experience. We had to optimize the audio capture and transmission process to ensure that there were minimal delays between the user's speech and the system's response.

- **Managing Multiple Audio Formats:**
  The Speech-to-Text API supports various audio formats, but ensuring compatibility with our system's input and output requirements was necessary. We made sure that the audio captured from users was in the right format (WAV, FLAC, etc.) and properly encoded before sending it to the API.

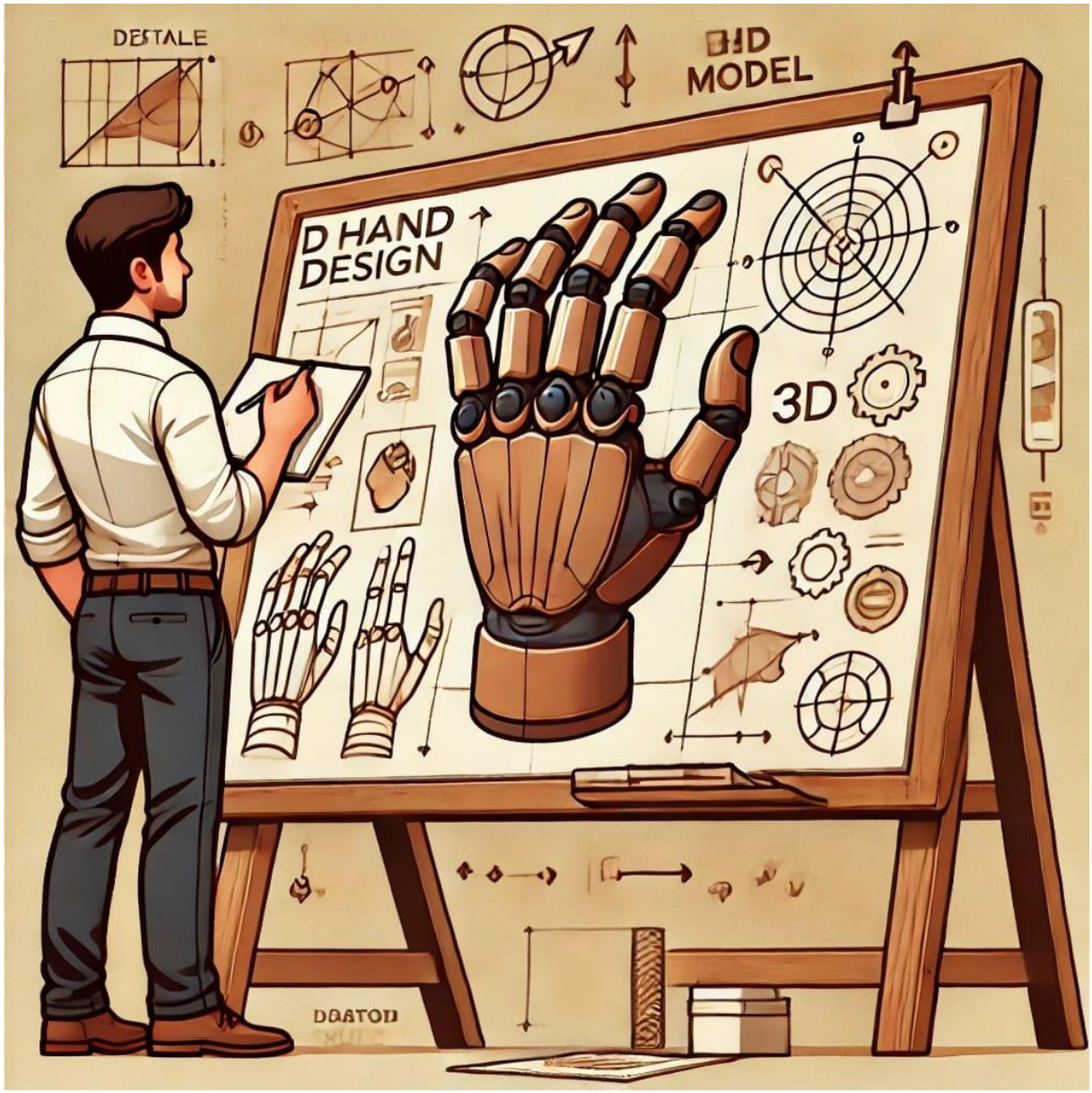
5.5 Conclusion

In this chapter, we explored the integration of Google Cloud's Text-to-Speech and Speech-to-Text APIs into our application. These powerful tools enabled us to convert text into natural-sounding speech and transcribe spoken language into text, which are core components of modern voice-based applications.

We demonstrated how the APIs are utilized within our project, detailing the setup process, configuration of various parameters, and the methods used for speech synthesis and recognition. By leveraging Google Cloud, we were able to enhance the user experience with high-quality audio outputs and seamless speech input processing.

The integration of these APIs not only streamlined our development process but also provided robust, scalable solutions that can be easily adapted to a wide range of use cases. Despite some challenges faced in optimizing speech recognition accuracy and audio quality, the overall results highlight the effectiveness and flexibility of Google Cloud's offerings in building advanced voice-enabled applications.

CHAPTER 6 : 3D



6.1 Introduction to 3D Modeling

As we focus on enhancing expressive language in our application, we recognized the need to integrate motion in graphics to effectively bridge the gap between hearing individuals and the deaf and mute community. 3D Blender emerged as the perfect solution, offering robust tools for creating detailed and dynamic animations. Its capabilities allowed us to precisely represent gestures, making our platform more accessible and visually engaging. This chapter delves into the process, from model preparation to rendering, showcasing the critical role 3D Blender played in achieving our objectives.

6.2 Model Preparation

6.2.1 Model Selection

As we began our search for 3D models to support our project, we explored various online resources and platforms offering high-quality assets. Through this process, we discovered Sketchfab, a leading online platform renowned for its vast library of 3D models. Sketchfab hosts contributions from a global community of artists and designers, offering models across diverse categories such as characters, objects, and environments. This extensive selection ensures access to high-quality resources suitable for a wide range of projects. The platform also features an intuitive interface that allows users to preview models in 3D before downloading, helping ensure that the selected assets align with project requirements.

Our team chose Sketchfab not only for its diverse library but also for its ability to provide realistic, detailed models essential for creating accurate and expressive animations. For our project, we chose a realistic hand model from Sketchfab due to its detailed anatomy and high resolution. This level of realism was essential for accurately representing Arabic sign language gestures, as it enabled us to capture the subtle nuances of hand movements and gestures, which are critical for representing Arabic sign language effectively. This decision reinforced the visual authenticity of our application, ensuring that it resonates with users and achieves its goal of bridging communication gaps.

By selecting a high-quality hand from Sketchfab, we were able to align the project with our visual identity while minimizing the time required for manual modeling. This decision

significantly streamlined the workflow, enabling the team to focus on rigging and animation, which are vital for bridging communication gaps in our application.

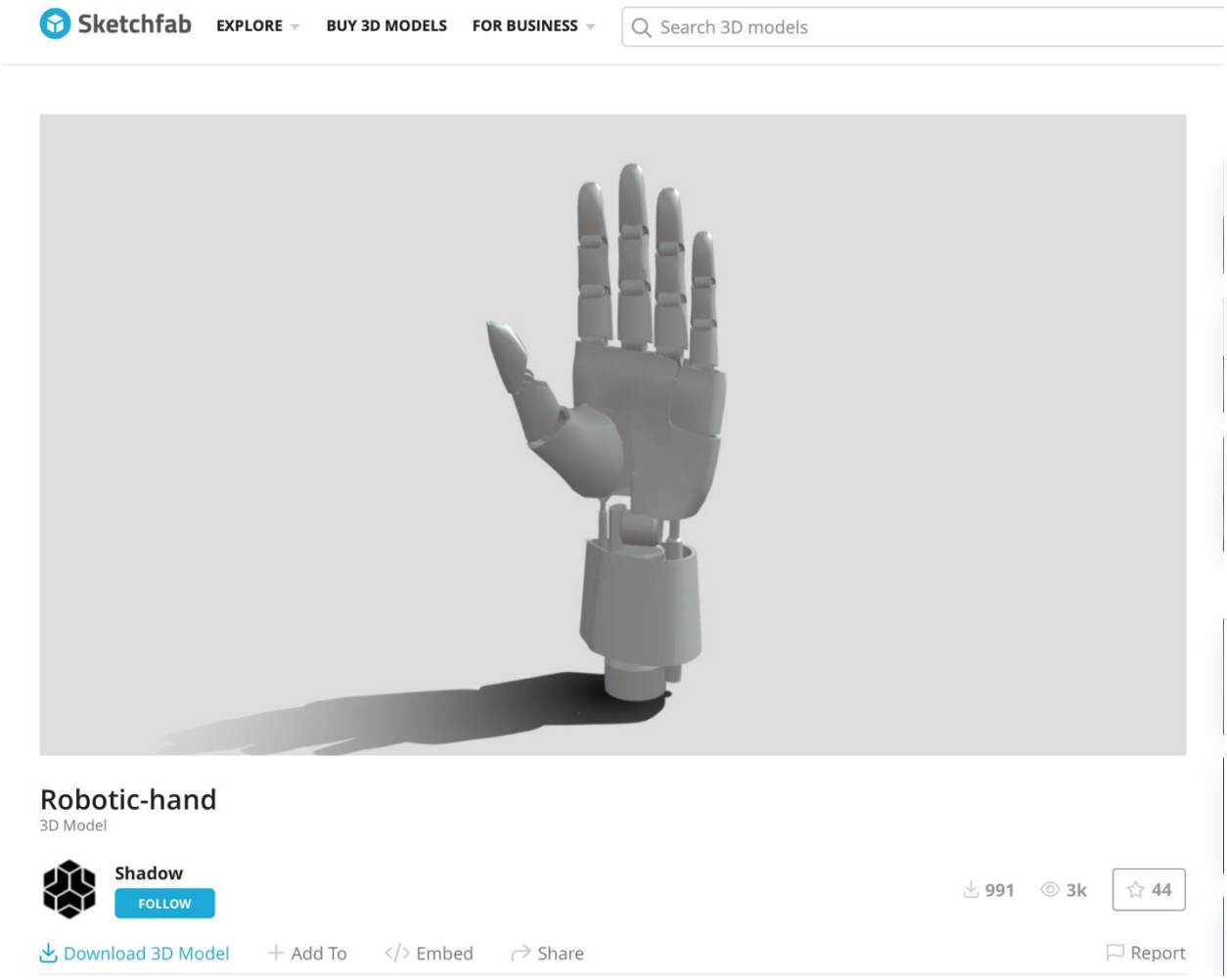


Figure 16: Robotic Hand 3D Model from Sketchfab

6.2.2 Model Optimization

The selected model underwent a detailed preparation process to ensure its compatibility with rigging and animation stages. Each part of the hand model—including fingers, the palm, and joints—was carefully separated into individual components. This segmentation was essential to provide greater control over the movements during the rigging phase. Additionally, the mesh of the model was meticulously cleaned, removing any unnecessary details or imperfections that might hinder the smoothness of animations.

This preparatory work was not only a technical necessity but also a foundational step for achieving seamless integration with subsequent rigging and animation workflows. By dividing the model into its constituent parts, we ensured that each element could be manipulated independently, allowing for more fluid and accurate gestures. This level of precision was critical to maintaining the visual and functional integrity of the animations, ensuring that they accurately conveyed the nuances of sign language gestures.

6.3 Rigging the Model

6.3.1 Creating the Armature

Rigging in SAHLA began with the creation of an armature, which serves as the skeletal framework for the hand model. The armature consists of interconnected bones that mimic the anatomical structure of the hand. Each bone was carefully positioned and aligned in Blender's Edit Mode to match the layout of the hand, ensuring that all key components such as fingers, joints, and the palm were properly represented. This precise positioning was essential for enabling realistic and accurate motion control during animations.

6.3.2 Defining Bone Hierarchies

The bones in the armature were organized into a hierarchical structure, where each bone's movement influenced its connected parts. For instance, moving the base of a finger automatically adjusted the connected segments, replicating natural hand movements. This hierarchical organization ensured a logical and efficient system for posing and animating the model.

6.3.3 Weight Painting

Weight painting played a critical role in assigning specific areas of the hand mesh to the corresponding bones. In Blender's Weight Paint mode, each bone's influence was visualized as a gradient of colors, from red (maximum influence) to blue (no influence). This allowed the team to fine-tune how the mesh deformed during bone movements. For example, the finger bones were carefully adjusted to ensure smooth bending without distorting nearby areas like the palm.

6.3.4 Parenting the Armature to the Mesh

To connect the armature to the hand model, the mesh was parented to the armature using Blender's "With Automatic Weights" option. This step automatically distributed the influence of each bone across the mesh, creating a functional base for further refinement. The automatic weights were reviewed and adjusted as needed to ensure precise deformations that adhered to the natural movements of the hand.

6.3.5 Enhancing Realism in Motion

Fine-tuning the rigging setup allowed the hand model to respond seamlessly to various poses and gestures. This was particularly important for representing Arabic sign language, where subtle finger movements convey specific meanings. Each finger and joint was rigged to articulate nuanced gestures, making the animations both realistic and expressive.

Rigging was an essential step in SAHLA, providing the foundation for dynamic and lifelike animations. By implementing a carefully structured armature and employing meticulous weight painting techniques, the project successfully bridged the communication gap between hearing individuals and the deaf and mute community, ensuring that every gesture was both visually accurate and meaningful.

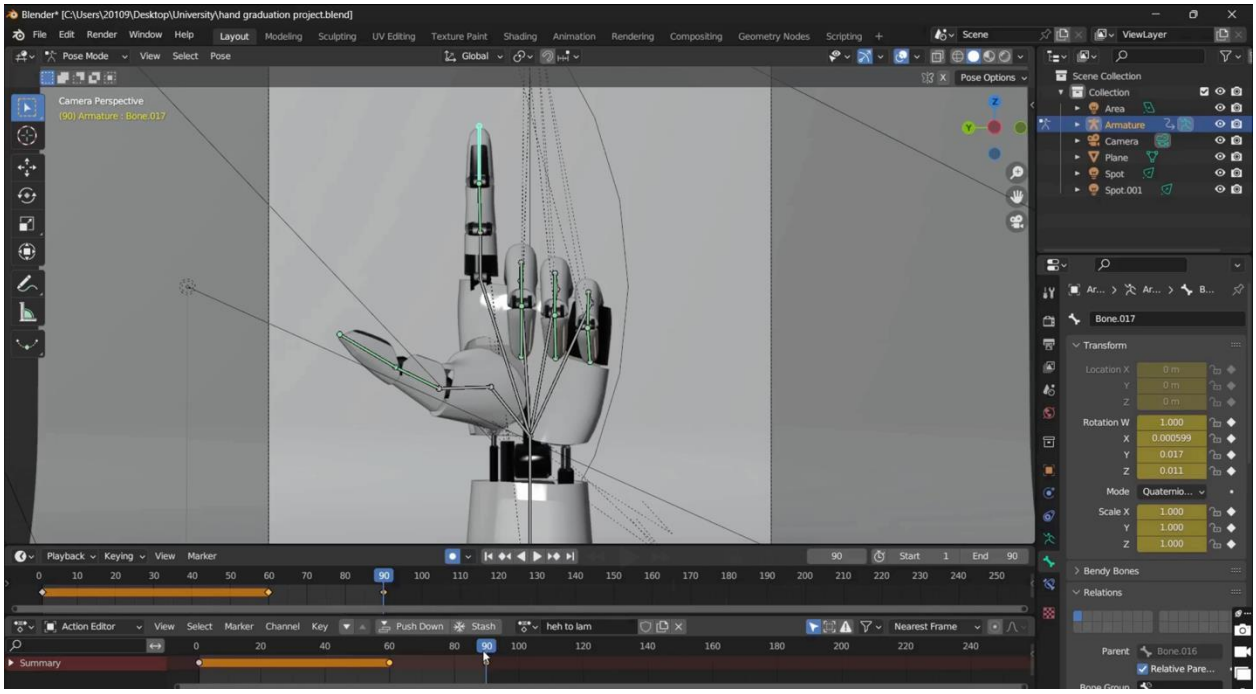


Figure 17: Robotic Hand Animation in Blender with Bone Rigging and Pose Mode

6.4 Setting Up Inverse Kinematics

6.4.1 Understanding IK in Blender

Inverse Kinematics (IK) simplifies the animation process, and makes it possible to make more advanced animations with lesser effort.

Inverse Kinematics allow you to position the last bone in a bone chain and the other bones are positioned automatically. This is like how moving someone's finger would cause the rest of the hand to follow it. By normal posing techniques, you would have to start from the root bone, and set bones sequentially until you reach the tip bone: When each parent bone is moved, its child bone would inherit its location and rotation. Thus making tiny precise changes in poses becomes harder farther down the chain, as you may have to adjust all the parent bones first.

This effort is effectively avoided by use of IK.

6.4.2 Creating the Armature for IK

The process began with the creation of an armature that consisted of a chain of bones representing the hand and fingers. In Edit Mode, bones were added and positioned to form a continuous chain from the lower hand to the fingers. This setup laid the foundation for implementing the IK system.

6.4.3 Setting Up the IK Constraint

To enable IK, we switched to Pose Mode and selected the final bone in the chain, such as the hand bone. An IK constraint was applied using Blender's constraints menu, allowing the system to calculate joint movements based on the position of the end bone. The chain length was specified to include the lower and upper bones, ensuring accurate motion across the entire limb.

6.4.4 Assigning the IK Target

An empty object was added to the scene to act as the IK target. This target served as the control point for positioning the limb. By moving the empty object, the entire chain of bones adjusted accordingly, enabling precise control over hand and finger movements. The empty object was parented to the armature for easier management during animation.

6.4.5 Testing and Refinement

Rigorous testing was conducted to ensure that the IK behaved naturally and consistently. Various poses were tested to identify and resolve any unnatural deformations or movements. Adjustments were made to the IK constraints and bone weights to refine the motion, particularly for complex gestures required in Arabic sign language.

6.4.6 Streamlining the Animation Process

The IK setup significantly reduced the time and effort required to create animations. By moving the IK target, animators could achieve fluid and lifelike gestures with minimal manual adjustments. This was crucial for SAHLA, where precise hand movements are essential for accurately representing sign language. The use of IK ensured that each gesture was not only visually appealing but also functionally effective in bridging the communication gap for the deaf and mute community.

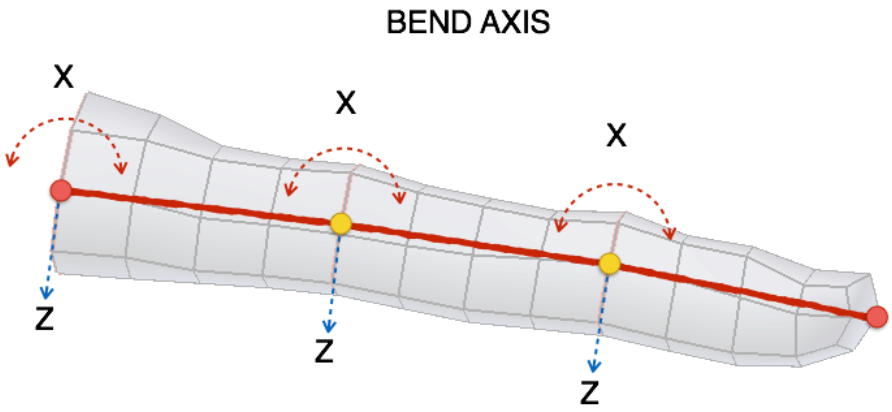


Figure 18: Bone Rigging and Deformation for Joint Animation in 3D Models

6.5 Scene Preparation

6.5.1 Creating the Background

The first step in scene preparation was designing a simple yet effective background that reflected the visual identity of SAHLA. This background was carefully chosen to maintain a professional aesthetic while ensuring that the focus remained on the hand animations. By aligning the background with our project's branding, we established a cohesive visual language that reinforced the accessibility and inclusivity goals of the application.

6.5.2 Setting Up Lighting

Lighting played a critical role in enhancing the clarity and realism of the animations. Soft fill lighting was employed to illuminate the hand model evenly, minimizing harsh shadows and ensuring that every gesture was clearly visible. The use of soft lighting not only improved the visual quality but also helped maintain the professional tone required for SAHLA's target audience.

6.5.3 Positioning the Camera

The camera was strategically positioned to capture the hand animations from an optimal angle. This involved adjusting the field of view and ensuring that the camera focused solely on the hand gestures, eliminating distractions. The positioning was refined through multiple iterations to guarantee that every animation was presented clearly and consistently.

6.5.4 Importance of Scene Preparation

Scene preparation was a foundational step in ensuring the overall success of the SAHLA animations. By thoughtfully designing the background, lighting, and camera setup, we created a visually engaging environment that emphasized the clarity and professionalism of the gestures. This meticulous preparation was vital for making the animations accessible and appealing to the target audience, ultimately supporting the project's mission of bridging communication gaps for the deaf and mute community.

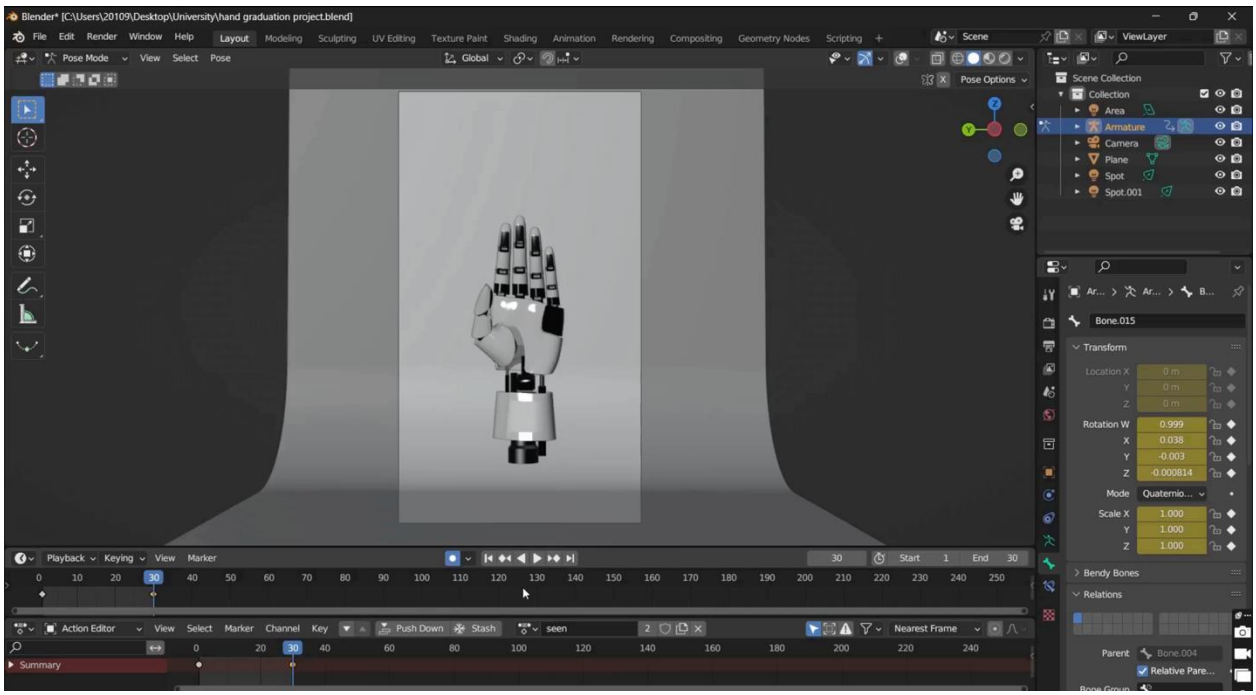


Figure 19: Blender Robotic Hand Scene Preparation

6.6 Animating

To animate the hand gestures in SAHLA, we followed a structured and detailed process to ensure consistency, accuracy, and clarity in every motion. This approach was integral to representing Arabic sign language effectively, bridging communication gaps for the deaf and mute community.

6.6.1 Individual and Transition Animations

Our animations were categorized into two types. Individual animations represented each letter of the Arabic alphabet as a standalone gesture. These animations were crafted meticulously to align with standardized sign language conventions, ensuring authenticity and usability. Transition animations, on the other hand, were designed to create seamless motions between consecutive letters. These transitions added fluidity and naturalness to the overall presentation, enhancing the user experience by maintaining visual continuity.

6.6.2 Key Poses and Frame Structure

Each animation was standardized to 45 frames to maintain consistency across all gestures. This structure was divided into three distinct sections:

- Frames 1-15: The hand remains in a default "idle" position, providing a neutral starting point for each animation.
- Frames 16-30: The hand transitions from the idle pose to the target letter gesture. This phase ensures a smooth and logical movement.
- Frames 31-45: The hand holds the final pose, clearly displaying the letter gesture for the user to interpret with ease.

This framework ensured that every animation was visually cohesive and easy to follow.

6.6.3 Auto Keyframing and Workflow Efficiency

To simplify the animation process, we enabled auto keyframing. This feature automatically recorded adjustments made to the bones, capturing every movement precisely. Auto keyframing not only streamlined the workflow but also allowed us to focus on refining the animations. Each motion was reviewed and adjusted to ensure fluidity and accuracy, particularly in transitions between letters.

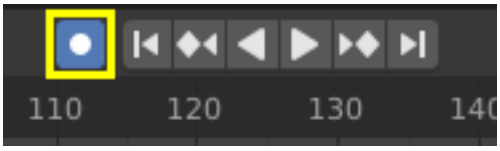


Figure 20: Blender Animation Playback Controls

6.7 Rendering Settings

Rendering serves as the final step in the animation process, transforming the 3D models and motions into a polished visual output ready for use. For SAHLA, this step was crucial not only for delivering a high-quality visual experience to the user but also for validating the success of the preceding processes such as modeling, rigging, and animation.

6.7.1 Using the Cycles Render Engine

We utilized the Cycles render engine in Blender, known for its ability to produce realistic lighting and shadows. This choice was instrumental in achieving an authentic visual

representation of the hand animations, ensuring that the details of the gestures were clear and impactful. The lighting and shadows enhanced the depth and realism of the scenes, aligning with the professional aesthetic of our project.

6.7.2 Resolution and Output Format

All animations were rendered at a resolution of 1920x1080 in portrait mode, a format ideal for mobile viewing and our intended audience. The AVI raw format was selected to ensure compatibility and to preserve the highest possible quality of the rendered animations. This choice allowed for seamless integration into the application without compromising on visual clarity.

6.7.3 Evaluating Project Success

Rendering also served as a critical checkpoint to evaluate the success of our earlier workflows. By reviewing the rendered output, we could confirm that the animations were accurate, fluid, and visually cohesive. Any discrepancies observed during this stage were addressed and refined before finalizing the output.

6.7.4 Preparing for Deployment

The rendering process marked the transition from development to deployment. With the animations fully rendered, the models and motions were ready to be integrated into the SAHLA application. This step ensured that the project met its objectives of delivering accessible, professional-quality animations that effectively bridge communication gaps for the deaf and mute community.

Rendering not only finalized the visual aspect of our project but also validated the effectiveness of our methods and preparations. It was a pivotal step that ensured SAHLA’s animations were both technically sound and user-friendly, aligning with the overarching mission of inclusivity and accessibility.

6.7.5 Sample 3D Letters

Figure 21: Seen(س)

Figure 22: Letter Heh (ه)

Figure 23: Letter Lam(ل)

Figure 24: Letter Teh marbota(ة)

6.8 Integrating With SAHLA

6.8.1 Introduction

The integration process utilizes a collection of pre-rendered transition videos for all possible letter combinations in Arabic. By concatenating these videos based on the input text, the app creates smooth and accurate sign language representations, making it a powerful tool for communication with the hearing-impaired community.

The key innovation lies in leveraging a pre-rendered dataset of transition videos between Arabic letters and special characters. With 29 letters (teh marbota included) and 2 additional characters(Al and space) resulting a total of 31 possible characters, each possible transition has been meticulously created, resulting in a comprehensive collection of 930 transition videos (31 × 30). Using this collection, SAHLA can dynamically generate sign language videos for any given word or sentence by concatenating the corresponding transitions, ensuring smooth and natural representation.

6.8.2 Workflow for Sign Video Creation

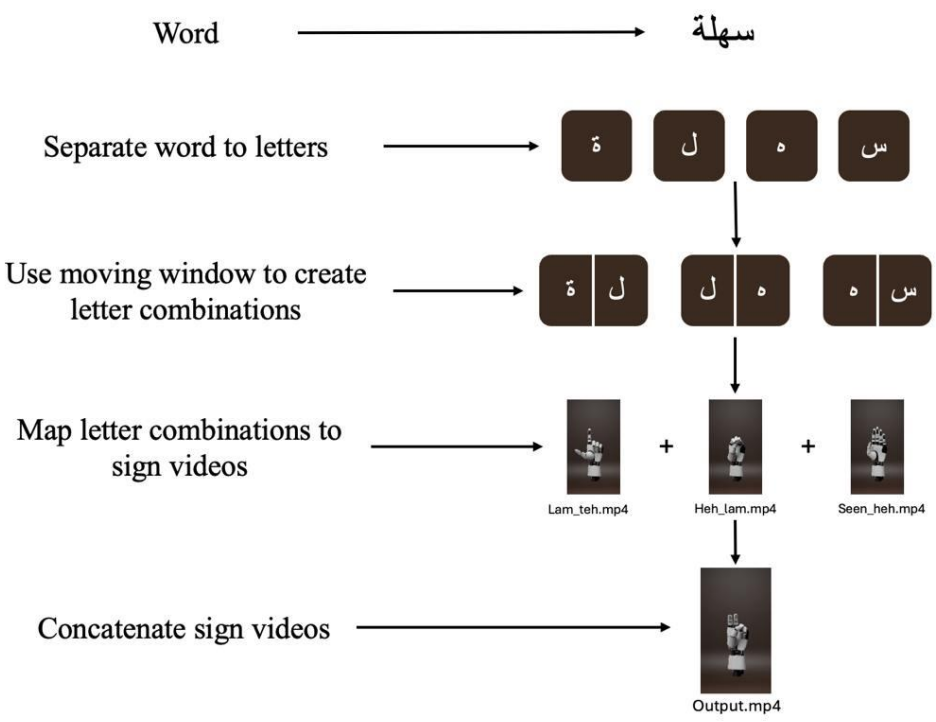


Figure 25: sign language video generation workflow

The figure illustrates the integration process for generating sign videos using the SAHLA system:

1. Separate Word into Letters:
   The input word, "سهلة", is broken down into its individual Arabic letters: س, ه, ل, and ة.
2. Create Letter Combinations Using a Moving Window:
   A sliding window approach is applied to form pairs of consecutive letters, representing transitions between them. For the word "سهلة," the transitions generated are:
   - **س → ه**
   - **ه → ل**
   - **ة → ل**
3. Map Letter Combinations to Sign Videos:
   Each letter pair is matched to its corresponding pre-rendered sign transition video from the library. For example:
   - س → ه maps to Seen_heh.mp4
   - ه → ل maps to Heh_lam.mp4
   - ل → ة maps to Lam_teh.mp4
4. Concatenate Sign videos:
   The mapped videos are concatenated in sequence to create a complete sign language representation of the word. In this case, the output is a single video file (Output.mp4) that visually represents the word "سهلة" in sign language.

6.9 Conclusion

The integration of 3D Blender modeling and animation into SAHLA provided a robust framework for bridging communication barriers and offering a voice to the voiceless. Every step in the process, from model preparation to rendering, was a meticulous effort aimed at creating a visually engaging and highly functional system. The rendered animations now serve as a transformative medium, enabling seamless communication for the deaf and mute community by visually translating Arabic sign language gestures. This achievement underscores the project's mission to empower individuals and foster inclusivity. Moving forward, the focus will be on refining these animations further and exploring advanced rendering techniques to amplify the project's impact and usability, ensuring it continues to provide an accessible and meaningful experience for all users.

CHAPTER 7: React Native

7.1 INTRODUCTION

In today's mobile-first world, developing applications that are user-friendly and responsive across platforms is essential. For the mobile development phase, we focused on creating a robust, user-friendly interface for both iOS and Android. The frontend in mobile applications refers to the part of the app that users see and interact with. It includes everything from the app's design and layout to its buttons, text, images, and other interactive elements.

7.2 USING FRAMEWORKS

Our mobile frontend development relied on frameworks rather than native platform-specific coding (e.g., Objective-C/Swift for iOS or Java/Kotlin for Android). This decision was made to simplify the development process, speed up development time, and maintain high code quality.

Frameworks provide pre-built libraries, templates, and components that can be seamlessly integrated into the project. This allows developers to focus on writing application-specific logic while the framework handles the lower-level implementation details.

Additionally, frameworks follow standardized conventions and guidelines, enabling better collaboration among developers. The modular structure they enforce makes the code more organized, maintainable, and easier to scale.

Most frameworks also come equipped with built-in solutions for common tasks such as navigation, state management, and animations, reducing the need for repetitive code. This results in faster development cycles, consistent code quality, and higher productivity. Features like live or hot reloading further enhance the development process by providing instant feedback on code changes.

7.3 WHAT IS REACT NATIVE?

React Native is a framework based on the React library that enables developers to build mobile applications using a component-based architecture. This approach divides the app into reusable components, making the codebase easier to understand, debug, and maintain. Each component encapsulates its own logic and design, allowing it to be reused across the application.

React Native uses JSX, a syntax extension that combines JavaScript with XML-like tags, to define components. Instead of using HTML elements (like or <button>), React Native provides platform-specific components (like <View>, <Text>, and <TouchableOpacity>). JSX also allows developers to write conditional logic directly within the component's layout, enabling dynamic rendering based on app state or user interaction.

React Native leverages a **shadow tree** and communicates directly with native platform elements through a **bridge**. This bridge translates JavaScript code into native code, enabling high-performance updates to the UI. When a component's state changes, React Native efficiently updates the shadow tree and synchronizes the changes with the native views.

This architecture allows React Native to deliver near-native performance while maintaining the flexibility and productivity of JavaScript development. Its ability to use a shared codebase for both iOS and Android makes it a standout framework for cross-platform mobile application development.

7.3.1 Why Use React Native?

When deciding on a technology for our mobile application, we evaluated various options and ultimately chose React Native. This decision was driven by React Native's unique strengths that align with our project goals. Here are the key reasons why React Native stood out as the ideal choice for mobile app development:

1. Rapid Development : React Native promotes rapid development by enabling developers to build mobile apps with reusable components, hot reloading, and a shared JavaScript codebase. This approach simplifies cross-platform development for both iOS and Android, significantly reducing development time and effort. The ability to see real-time changes during development enhances productivity and accelerates the iterative process.

2. Code Reuse: One of React Native's strongest features is its support for code reuse. The codebase is shared between iOS and Android, reducing duplication and ensuring consistency across platforms. Additionally, React Native offers pre-built components and libraries tailored for mobile UI, which developers can leverage to streamline development and speed up delivery.

3. Efficiency and Organization: React Native's component-based architecture promotes efficiency and organization. This modular approach allows developers to divide the app into reusable, manageable components, making it easier to debug, maintain, and scale. React Native's structure fosters cleaner, more maintainable code, which is especially beneficial in larger projects with multiple developers.

4. Developer Productivity: React Native significantly enhances developer productivity with features like **live reload** (providing instant feedback on code changes) and **hot reload** (allowing state retention during updates). These features make it easier to iterate and experiment without restarting the application. Additionally, the robust ecosystem of tools and libraries available in React Native empowers developers to focus on building application logic rather than dealing with repetitive or low-level tasks.

5. Community Support: React Native has a large and active developer community, which is a valuable resource for developers. The community provides extensive documentation, tutorials, third-party libraries, and tools, ensuring that developers can find solutions to challenges and adopt best practices. Continuous contributions from the community help keep React Native updated and relevant, making it a reliable choice for modern mobile development.

7.4 React Native Architecture

The React Native architecture is the foundation of how we built SAHLA, a cross-platform application designed to provide seamless accessibility solutions for users on both Android and iOS. Leveraging React Native's capabilities, we implemented a robust system to ensure efficient data flow, smooth UI rendering, and responsiveness in user interactions. This architecture operates through a series of steps, broken into three main phases:

1. Render Phase

React executes the JavaScript code to create a React Element Tree. This tree is then used to generate a React Shadow Tree in C++.The React Shadow Tree mirrors the structure of the React Element Tree but operates at a lower level for efficient layout calculations.

2. Commit Phase

The React Shadow Tree undergoes layout calculation using a library called Yoga, which determines the size and position of each element. After this, the Shadow Tree is promoted to the "next tree" ready for mounting.This phase ensures the tree is optimized and ready for rendering, with asynchronous operations improving performance.

3. Mount Phase

The React Shadow Tree is transformed into a Host View Tree that corresponds to the platform's native views. For example, on Android, it creates `android.view.ViewGroup` and `android.widget.TextView`, while on iOS, it creates `UIView`.At the end of this phase, the UI is rendered on the screen.

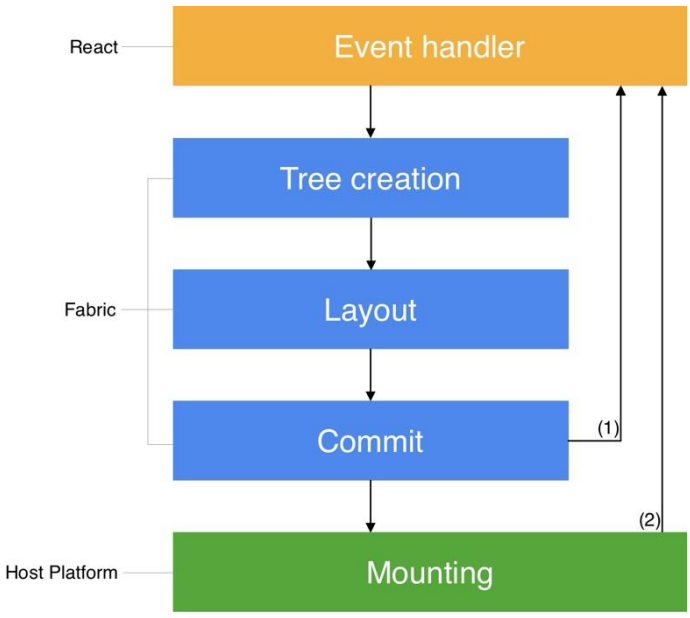


Figure 26: React Rendering Lifecycle with Fabric Architecture

7.5 Cross-Platform Implementation

The introduction of a core C++ renderer, called Fabric. Fabric provides a unified and efficient rendering logic across platforms, allowing us to focus on creating a consistent experience for all users.

Key features of the Fabric renderer utilized in SAHLA include:

- **Shared Core:** A single C++ implementation ensures uniform behavior and reduces platform-specific discrepancies in SAHLA's UI.
- **Reduced Overhead:** By integrating Yoga directly into the core renderer, this minimized the performance overhead for layout calculations, especially for interfaces like the recording and translation screens in SAHLA.
- **Immutability Enforcement:** Leveraging C++'s immutability features, this ensured thread-safe operations for SAHLA's UI updates, preventing concurrency issues during heavy usage.

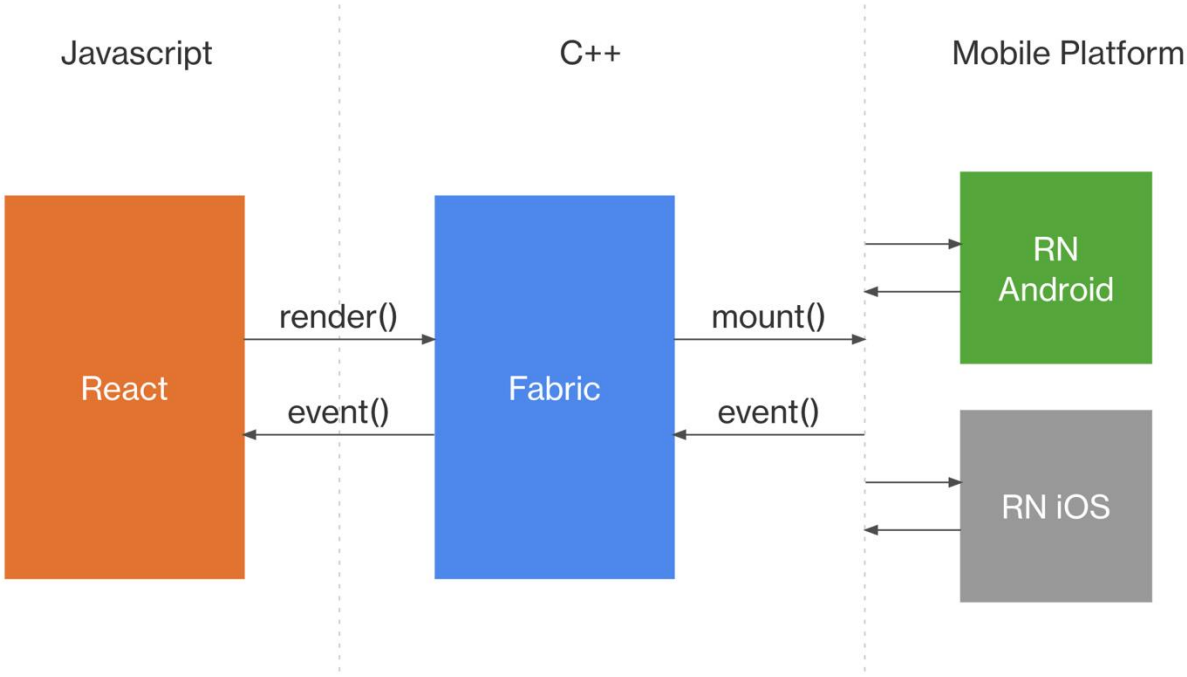


Figure 27: React Native Communication Flow using Fabric Architecture

The architecture's flow can be visualized in three main layers:

1. JavaScript Layer

- **React:** This layer contains the JavaScript code for SAHLA's logic, defining the structure and behavior of components such as the interfaces for choosing translation modes and displaying results. React creates the React Element Tree and interacts with the Fabric renderer through render() and event() methods.

2. C++ Core (Fabric)

- **Fabric Renderer:** The central component that:
  - Accepts React's render() calls to create and manage the React Shadow Tree.
  - Handles layout calculations using Yoga for elements like SAHLA's translation selection buttons and recording interfaces.
  - Communicates with both the JavaScript layer and the host platform layer to synchronize rendering and user interactions.

3. Mobile Platform Layer

- **RN Android and RN iOS:** Fabric interacts with platform-specific APIs to render SAHLA's UI and handle user interactions. It uses mount() to place components into the native view hierarchy and event() to process actions like taps and gestures.

Example Flow Using SAHLA:

1. **Render:** React calls render() to pass the React Element Tree to Fabric. For example, this could represent SAHLA's main interface where users select translation modes (e.g., speech-to-text or sign language).
2. **Mount:** Fabric uses mount() to create platform-specific views. For instance:
   - On Android, we created ViewGroup elements for the recording interface and the translated output display.
   - On iOS, we used UIView components for interactive elements like video recording buttons and translation result screens.
3. **Event Handling:** When a user taps the "Record" button in SAHLA, an event is generated and sent back to Fabric. Fabric processes this event and communicates it to React, which updates the app's state to start the recording or display the translated results.

By utilizing React Native's architecture, we built SAHLA to ensure efficient rendering, seamless updates, and a consistent experience for users across both platforms.

7.6 Expo: Simplifying Mobile Development

In the process of creating the **SAHLA app**, Expo played a vital role in streamlining our development workflow. By providing a comprehensive toolkit tailored for mobile app development, Expo allowed us to focus on building features that aligned with SAHLA's mission of delivering a seamless and user-friendly experience. Its tools and services simplified testing, configuration, and deployment, enabling us to iterate quickly while maintaining high-quality standards. With Expo, we could prioritize what truly matters: ensuring the SAHLA app is visually appealing, intuitive, and responsive to the needs of our users, all while reducing the complexity of mobile app development.

How Expo Simplifies Development

1. **Preconfigured Environment**:

   Expo takes care of all the complex native code implementations for both iOS and Android platforms, effectively handling the heavy lifting involved in mobile app development. This allowed our team to focus entirely on writing JavaScript, without needing to dive into the intricacies of platform-specific configurations or setups. By eliminating the need for such detailed platform-level work, Expo significantly accelerated our development process. It not only saved us a considerable amount of time but also ensured that our workflow remained streamlined and efficient, enabling us to deliver high-quality results with greater speed and consistency.

2. **Expo Go**:

   The **Expo Go** app enabled us to test the SAHLA app instantly on physical devices or simulators without recompilation. This real-time feedback made debugging faster and ensured the app performed as intended across different devices.

3. **Over-the-Air (OTA) Updates**:

   OTA updates provided us with the ability to deliver bug fixes and introduce new features directly to SAHLA users without the delays typically associated with app store approval processes. This functionality ensured that SAHLA stayed current and responsive to user feedback, maintaining a seamless and efficient experience for our audience. By bypassing traditional update hurdles, we could quickly adapt and continuously enhance the platform to meet evolving user demands.

4. **Expo Libraries**:

   Expo provided us with a rich set of libraries that simplified common development tasks for the SAHLA app:

- **expo-asset**: Streamlined image caching and static resource management, improving app performance.
- **expo-font**: Allowed us to incorporate custom fonts, like Poppins, enhancing the app's design and branding to reflect SAHLA's identity.
- Other tools for notifications, permissions, and app functionalities reduced development effort while maintaining functionality.

5. **EAS Build and Deploy**:

   With **Expo Application Services (EAS)**, we could build and distribute production-ready versions of the SAHLA app effortlessly for both iOS and Android. This streamlined process saved time and ensured a smooth transition from development to production.

7.7 TypeScript in the SAHLA App

During the development of the **SAHLA app**, we chose to use **TypeScript** as our primary programming language. This decision was driven by TypeScript's ability to enhance the development process, providing tools that contribute to the app's reliability, maintainability, and scalability. TypeScript proved to be an essential part of creating a seamless and user-friendly experience for SAHLA's users.

Key Benefits of Using TypeScript in SAHLA App:

- **Robust Type-Checking System**:

  Ensuring the accuracy of data throughout the SAHLA app was critical. TypeScript enforces strict rules on the types of data used in the code, allowing us to catch potential issues early in development. This reduced the likelihood of bugs and ensured smooth functionality across the app.

- **Improved Code Clarity and Comprehension**:

  By explicitly defining data types, TypeScript made the SAHLA app's codebase easier to read and understand. This not only streamlined collaboration among developers but also made the code easier to maintain and enhance over time. Clear type definitions also acted as helpful documentation for the app.

- **Error Prevention**:

  With the interconnected features of the SAHLA app, TypeScript served as a safeguard against unexpected behaviors. It helped detect and prevent errors during development, leading to a smoother development process and a more stable application.

- **Enhanced Maintainability**:

  TypeScript's consistent and predictable patterns ensured that the SAHLA app's codebase was well-organized and easy to maintain. This makes it simpler to introduce new features or updates in the future without introducing unintended issues.

7.8 Folder Structure of the SAHLA App:

To ensure clarity and scalability, the SAHLA app is organized into a well-defined folder structure. This structure facilitates efficient collaboration, maintainability, and seamless scalability as the project grows. Here's an overview of the key directories and their functionalities:

Figure 28: SAHLA Project Directory Structure

1. app: This directory contains the main application screens and layout components.It includes files like `audio-to-sign.tsx`, `splash-screen.tsx`, and other key pages that make up the core user interface and functionality of the SAHLA app.

2. components:

Houses reusable components to maintain consistency throughout the app.Includes modular components like `button.tsx` and `index-box.tsx`, which are used across different parts of the app to ensure a unified design and functionality.

3. api:

Manages API integration and client setup.Contains configurations for Axios and React Query in files like `api-provider.tsx` and `client.tsx`. It also includes service functions for interacting with external APIs such as Gradio.

4. stores:

Centralized state management using the **zustand** library to handle app-wide states, ensuring smooth and efficient screen transitions, with implementation in files like `resources.tsx` and `result.tsx`.

5. assets:

Organizes all static resources like images, fonts, and icons.Stores visual and multimedia assets in subdirectories (`fonts`, `images`), making it easier to manage and reference files like `Splash.gif`, `logo-2.gif`, and `index_image_1.png`.

7.9 Navigation

For seamless transitions between screens, the SAHLA app leverages **React Native Stack Navigation**. This navigation solution provides a smooth and intuitive user experience, ensuring that users can effortlessly navigate through the app's interface.

The navigation flow is managed by registering all application screens within a **stack navigator**. This configuration is centralized in the `Layout.tsx` file, which serves as the core navigation hub for the app. React Native Stack Navigation offers a robust and flexible solution for handling navigation, built upon the reliable foundation of the React Navigation library.

This setup ensures that transitions between screens are fluid and user-friendly, contributing to an overall polished and cohesive app experience.

7.10 State Management

To effectively manage the SAHLA app's state and data, we employed the **Zustand library**, a powerful and lightweight solution designed for scalability and flexibility. Zustand acts as a dependable system for organizing and managing state across the app, ensuring seamless communication between different parts of the application.

In practical terms, Zustand provides a centralized repository, or "store," where we define and organize essential data and actions using plain JavaScript objects and functions. This approach keeps the state management process simple, efficient, and easy to maintain, even as the application evolves.

Technically, Zustand's API is built around React hooks, enabling direct and efficient access to the app's state. Components connected to the store automatically re-render when the state they depend on changes, eliminating the need for complex data-passing mechanisms or excessive boilerplate.

Zustand also tackles common pitfalls in state management, such as context loss and React concurrency, making it a robust choice for managing the SAHLA app's data flow. Its balance

of simplicity and power ensures a smooth and consistent user experience throughout the application.

7.11 API Integration and Node.js Server

In the SAHLA app, seamless communication between the mobile application and external services is essential for delivering accurate results and providing a smooth user experience. To achieve this, we implemented a two-fold approach: integrating APIs within the mobile app and creating a **Node.js server** to act as a bridge between the mobile app and the **Gradio API**.

API Integration in the Mobile App

To handle API requests in the mobile app, we chose to combine the **@tanstack/react-query library** with **Axios**, leveraging the strengths of both tools:

- **React Query**:
  - Ensures improved performance through intelligent caching and background fetching.
  - Automatically updates the app's state when API data changes, reducing the need for manual state management.
  - Handles complex API scenarios like retries and stale data handling, ensuring the app remains responsive even during network issues.
- **Axios**:
  - Provides a robust solution for sending HTTP requests with easy-to-use syntax.
  - Offers built-in error handling and request cancellation, allowing for more efficient API interactions.
  - Simplifies the process of setting headers, query parameters, and payloads for API requests.

This combination of **React Query** and **Axios** empowers the SAHLA app to deliver a reliable and efficient experience, ensuring data is fetched and displayed in a user-friendly manner.

Node.js Server as a Connection Point

Since the **Gradio Client** cannot run natively on React Native, we created a **Node.js server** to act as the intermediary between the SAHLA app and the Gradio API. This server handles the following responsibilities:

1. **Functionality**:
   - The server sends data from the mobile app to the appropriate Gradio API endpoint.
   - It retrieves text, audio, or video results from Gradio and sends them back to the app for display on the result page.
2. **Middleware for Endpoint Management**:

- To ensure each API request is routed to the correct Gradio client, we implemented middleware. This middleware verifies that the appropriate Gradio client is initialized before processing the request, improving reliability and error handling.

3. **Gradio Client Initialization**:
   - The server initializes Gradio clients for each endpoint using their respective URLs.
   - It logs successful connections to ensure all clients are ready and handles errors gracefully if any initialization fails.
4. **Server Deployment**:
   - The server was deployed on **Koyeb.com**, a platform that simplifies the deployment of Node.js applications via GitHub repositories. This ensures high availability and ease of maintenance.
   - **Server Link**: https://mere-dog-selvster-ef99c8cd.koyeb.app

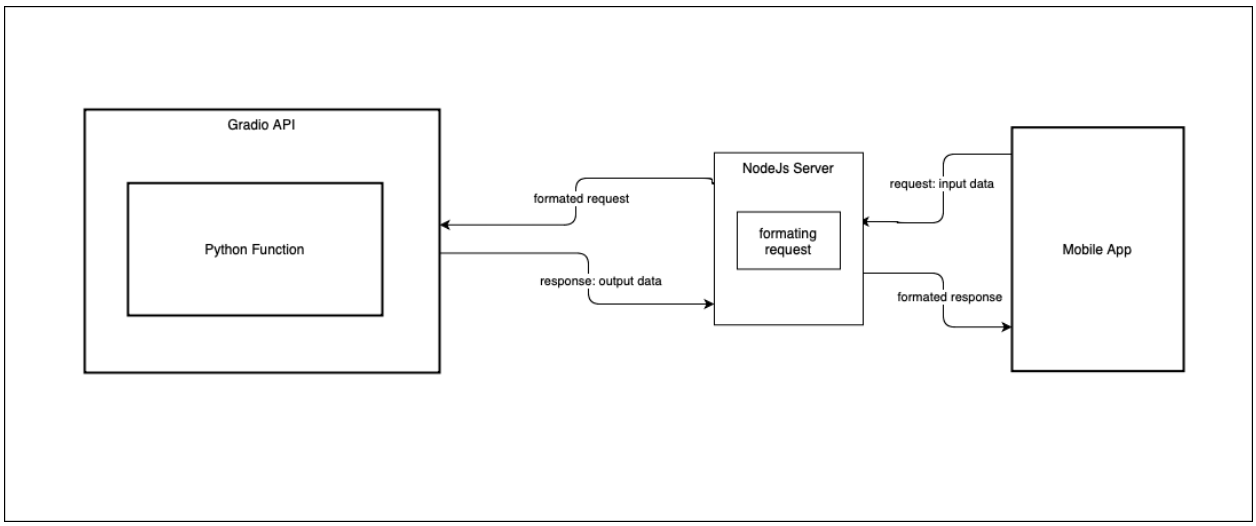


Figure 29 : Data Flow Architecture Between Gradio API, Node.js Server, and Mobile App

CHAPTER 8: User Interface / User Experience (UI/UX)

8.1 INTRODUCTION

The success of a digital application lies in its ability to combine **User Interface (UI)** and **User Experience (UX)** effectively. While UI focuses on the visual and interactive elements of the app, UX encompasses the overall feel and usability of the product. Together, they ensure that an application is not only functional but also enjoyable to use. This chapter explores how **SAHLA**, our real-time sign language conversion app, was designed with a focus on **intuitive UI** and **seamless UX** to create an inclusive and empowering experience for all users.

- **User Interface (UI): The Visual Layer**

  UI refers to the visual design of the app—the buttons, icons, colors, and layouts that users interact with. For SAHLA, the UI was designed to be **simple**, **accessible**, and **consistent**, ensuring that users can navigate the app effortlessly, regardless of their technical skills or abilities.

- **User Experience (UX): The Interaction Journey**

  UX focuses on the overall experience of using the app, from the first interaction to achieving the user's goal. SAHLA's UX was crafted to be **intuitive**, **inclusive**, and **user-centered**, ensuring that the app meets the needs of diverse users, including those with hearing impairments, and provides a seamless and meaningful experience.

By combining **UI** and **UX**, SAHLA creates a powerful tool that bridges communication gaps and empowers users to connect effortlessly. This chapter delves into the design process, showcasing how SAHLA's interface and experience were tailored to its mission of inclusivity and accessibility.

8.2 BRANDING

Branding is more than just a logo or a color scheme—it's the identity of a product and the emotional connection it creates with its users. For **SAHLA**, branding plays a crucial role in communicating the app's mission of inclusivity, accessibility, and ease of use. A strong brand ensures that SAHLA is not only recognizable but also resonates with its target audience, fostering trust and loyalty.

Brand Identity

SAHLA's brand identity is built on three core pillars:

1. **Simplicity:** Reflecting the app's ease of use and straightforward functionality.
2. **Inclusivity:** Representing the app's commitment to breaking communication barriers for all users.
3. **Innovation:** Highlighting the cutting-edge technology behind real-time sign language conversion.

Visual Elements

- **Logo:** The SAHLA logo incorporates clean, modern design with subtle elements that symbolize communication and connection (e.g., hands, speech bubbles, or waves).

Figure 30 : SAHLA Logo Design

- **Mascot:** SAHLA's mascot is a friendly, approachable character that embodies the app's mission. Designed to be inclusive and relatable, the mascot could be a hand-shaped character or an abstract figure that represents connection and communication. This mascot adds a human touch to the brand, making it more engaging and memorable.

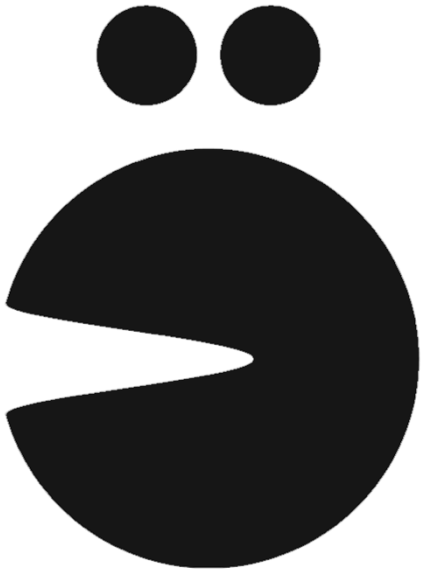


Figure** 31**: SAHLA Mascot

- **Color Palette:** Color is one of the most important and influential tools a designer has. In designs, it can set the brand tone and influence its image, draw users' attention, affect their emotions, and increase usability. However, finding the right combination of colors can be tricky and requires some basic knowledge and practice.

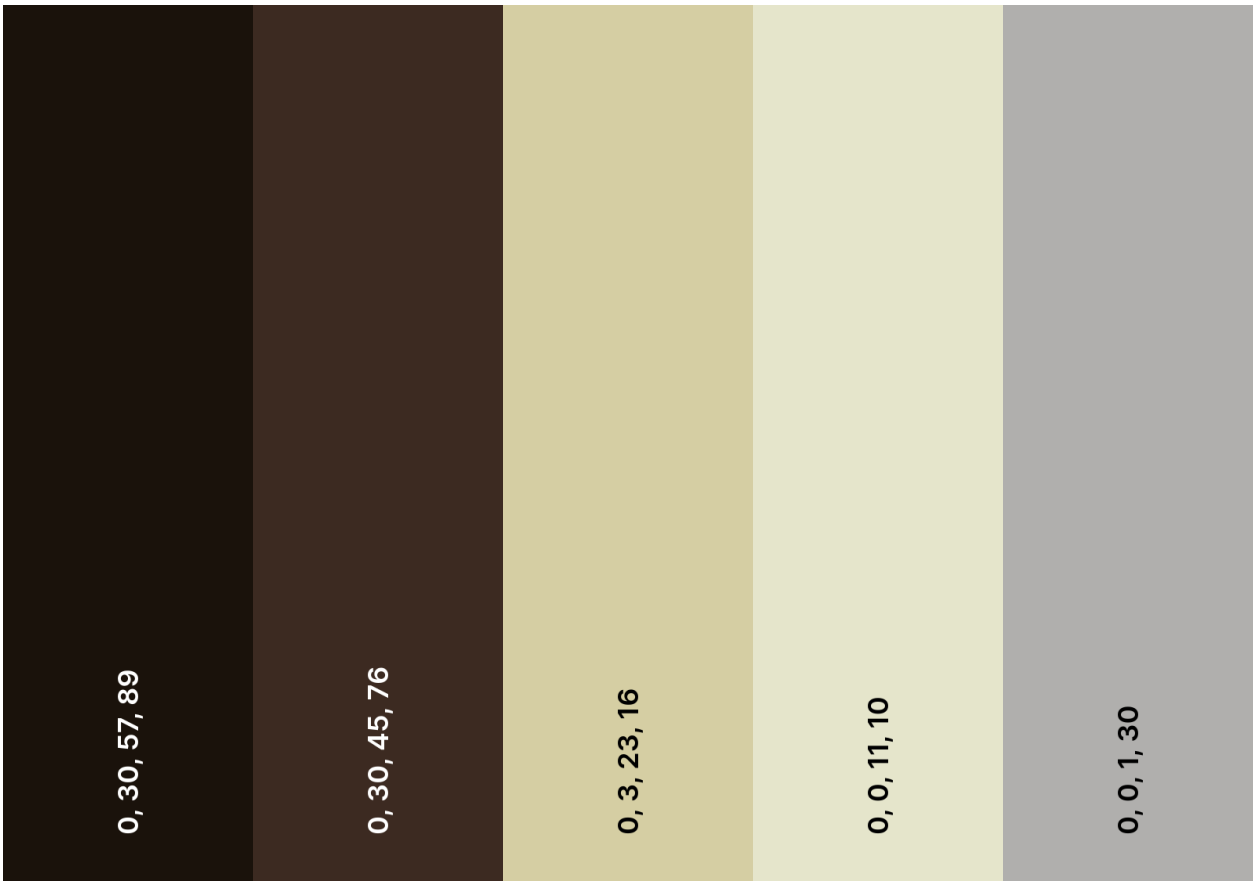


Figure 32: SAHLA Color Palette

- **Typography:** Typography is a key component in almost every digital experience. However, its complexity and industry-specific language makes it a common source of misalignment and confusion. You don't have to be a typography expert to design digital interfaces, but it's important to know some of these terms in order to have meaningful conversations with others on your team. Communicating clearly with team members about typography can help teams:
    - Increase legibility (and therefore usability) of an interface.
    - Improve visual polish and professional appearance.
    - Create a more consistent brand identity.
    - Cut down on costly revisions and iterations.

Aa

English Font

SKODA NEXT

Type	Font Size
Heading 1	20px
Heading 2	18px
Body – Large	16px
Body – Regular	14px
Body – Small	12px
Caption	12px
BUTTON	16px

Figure 33: English Typography Overview

سهلة

Arabic Font

Shareb

Type	Font Size
عنوان 1	20px
عنوان 2	18px
جسم - ضخم	16px
جسم - عادي	14px
جسم - صغير	12px
تعليق	12px
ضغط	16px

-

Figure 34: Arabic Typography Overview

- **Loading Screens Icons:**

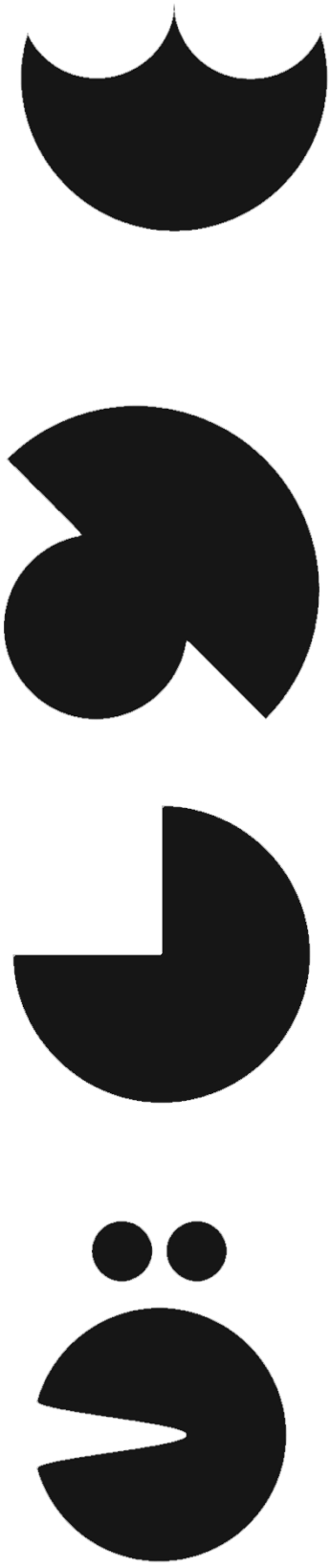


Figure 35: SAHLA Loading Screen Icons

As We see we created a pattern of loading screen icons inspired by our mascot. We have 4 different loading screen icons. Each one represents a letter from our Brand name **SAHLA** in Arabic. So, We have letters س, هـ, ل, ة. the 4 letters of our Brand name **سهلة**.

Brand Voice and Messaging

SAHLA's brand voice is **friendly**, **empowering**, and **approachable**, reflecting its mission to make communication effortless for everyone. Key messaging emphasizes inclusivity, accessibility, and the transformative power of technology.

Why Branding Matters for SAHLA

A strong brand ensures that SAHLA stands out in a competitive market while staying true to its mission. It creates a cohesive experience across all touchpoints—from the app interface to marketing materials—building trust and recognition among users. By aligning its branding with its core values, SAHLA not only attracts users but also inspires them to be part of a movement toward a more inclusive world.

8.3 USERFLOW

User flow is the path a user takes to complete a specific task within an application. It maps out every step, from the initial interaction to the final goal, ensuring a seamless and intuitive experience. For **SAHLA**, designing an effective user flow was critical to making sign language conversion simple, accessible, and efficient for all users.

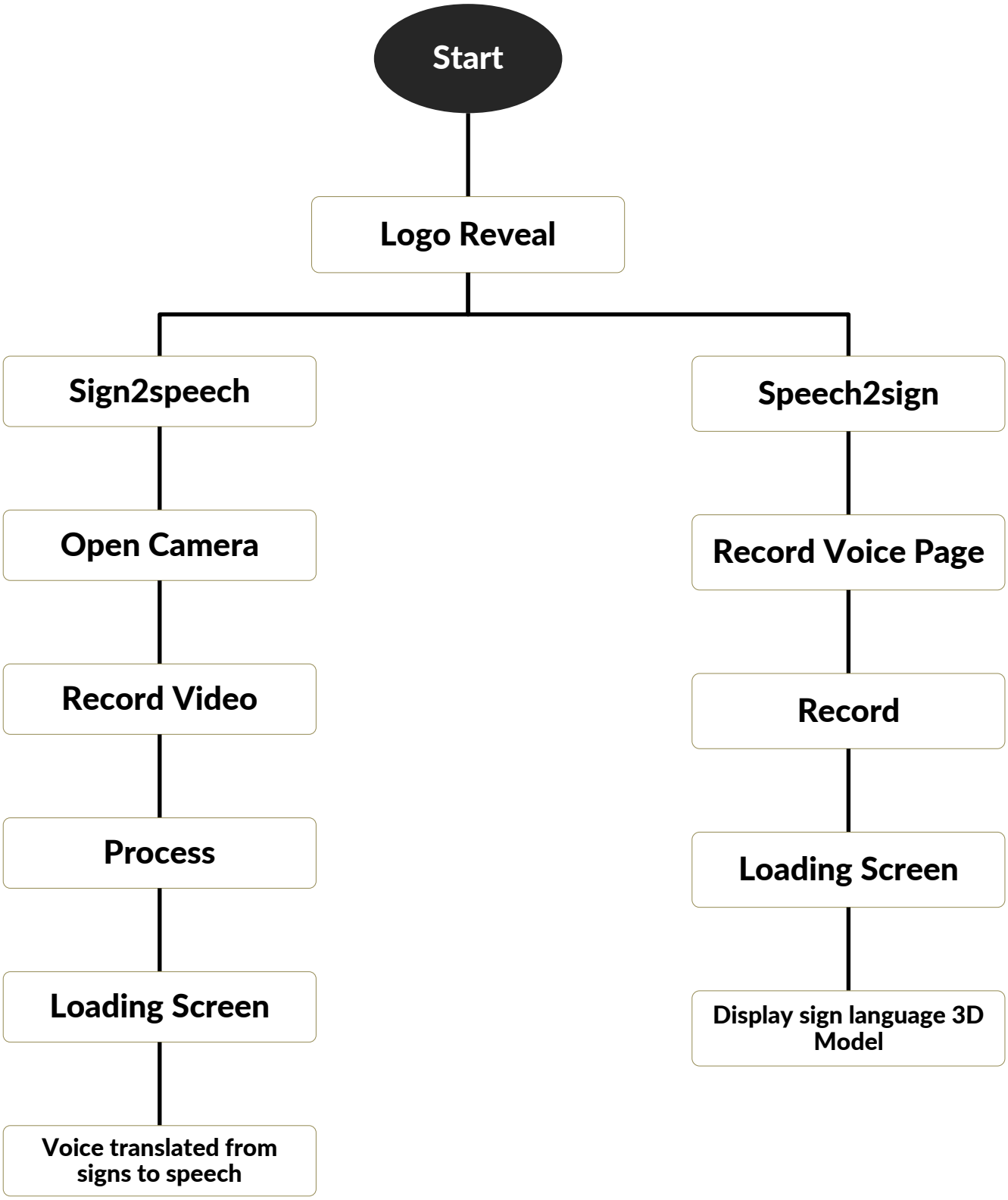


Figure 36 : Sahla Application Workflow Diagram

8.4 TOOLS USED:

The design and development of **SAHLA** relied on a suite of powerful tools to bring the app's vision to life. Each tool played a specific role in crafting the user interface, user experience, and visual identity of SAHLA, ensuring a cohesive and high-quality product.

1. Figma

- **Purpose:** UI/UX Design and Prototyping
- **Role in SAHLA:**
    - Used to create wireframes, mockups, and interactive prototypes.
    - Enabled collaborative design and real-time feedback from team members.
    - Allowed for testing user flows and refining the app's interface for optimal usability.

2. Adobe Photoshop

- **Purpose:** Graphic Design and Image Editing
- **Role in SAHLA:**
    - Used to design and edit visual assets, such as icons, illustrations, and promotional materials.
    - Helped create high-quality images for the app's branding and marketing.

3. Adobe Illustrator

- **Purpose:** Vector Graphics and Branding
- **Role in SAHLA:**
    - Used to design scalable vector assets, including the app's logo, mascot, and typography.
    - Ensured that all visual elements were consistent and adaptable across different platforms and screen sizes.

Why These Tools Were Chosen

The combination of **Figma**, **Adobe Photoshop**, and **Adobe Illustrator** provided a comprehensive toolkit for designing SAHLA's interface, user experience, and branding. Figma's collaborative features streamlined the design process, while Photoshop and Illustrator ensured that all visual elements were polished and professional. Together, these tools enabled the creation of an app that is not only functional but also visually appealing and user-friendly.

8.5 Conclusion

The design of SAHLA's User Interface (UI) and User Experience (UX) is a testament to the app's mission of inclusivity and accessibility. By combining a visually appealing UI with an intuitive and user-centered UX, SAHLA ensures that users of all abilities can interact with the app effortlessly. The emphasis on branding, user flow, and the strategic use of design tools such as Figma and Adobe Suite highlights the meticulous planning and execution involved in creating a cohesive and engaging experience. These elements come together to break communication barriers, fostering trust and usability while aligning with SAHLA's innovative goals.

As we transition into the final chapter, we delve into the deployment phase of the SAHLA app using Gradio. This chapter explores how Gradio's capabilities were leveraged to build and host interactive interfaces for our machine learning models, ensuring seamless accessibility and functionality. By integrating APIs, interactive tools, and cloud-based solutions, the deployment process bridges the gap between technical complexity and user accessibility, marking the culmination of SAHLA's development journey.

CHAPTER 9: Deployment

9.1 Introduction

In this chapter, we thoroughly explore the process of deploying our application, focusing on the steps and components required to ensure its accessibility and functionality. As we proceed, we will delve into how various elements of the system interact seamlessly to create a cohesive experience.

9.1.1 Overview of Deployment Process

When a user initiates a request, it navigates through the architecture of our application, passing through different layers and technologies. Understanding this journey is essential to grasp the deployment process and enhance its efficiency.

To provide a clear and comprehensive understanding, we include an architectural diagram that illustrates the communication between the application's components. This diagram highlights the flow of requests, showcasing how various technologies and tools work together to deliver a smooth and efficient user experience.

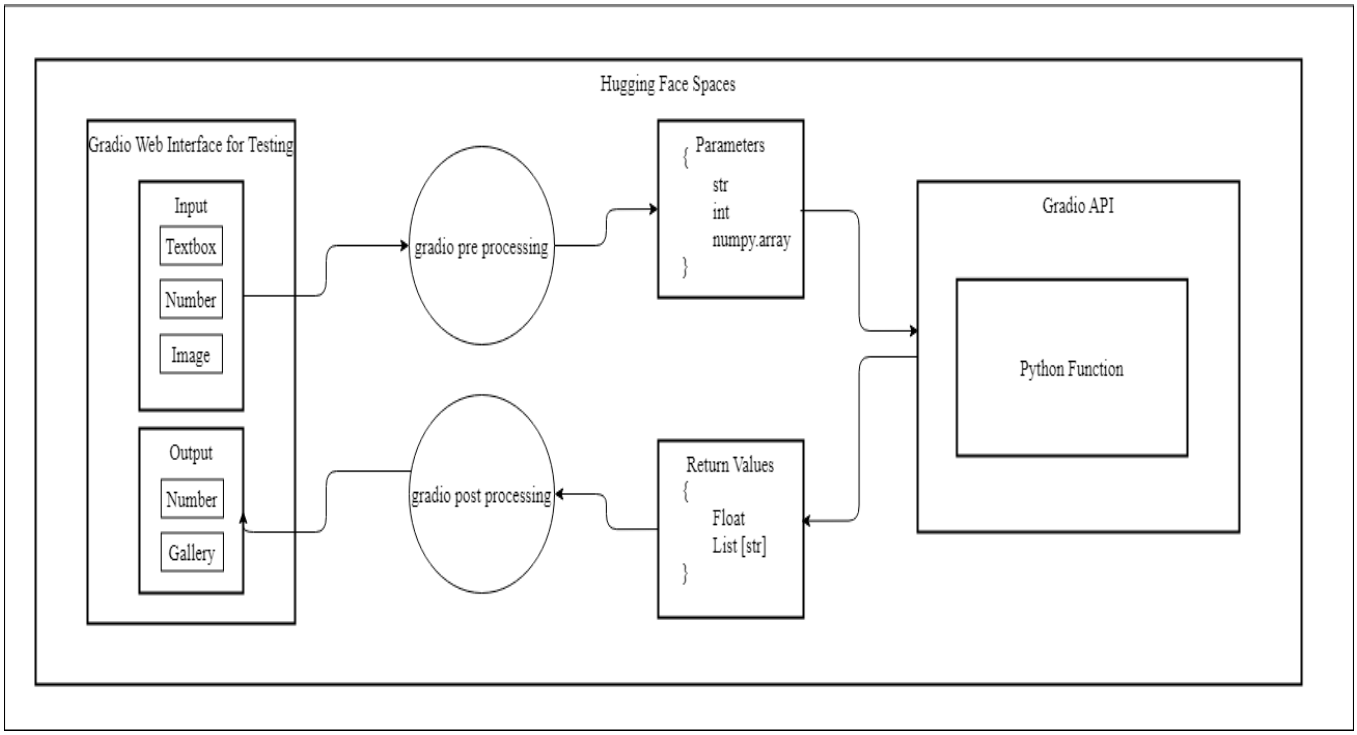


Figure 37 : Gradio Activity Diagram

Throughout this chapter, we will take a deep dive into how we built the user interface and API for our application using **Gradio**, a powerful and intuitive library for creating interactive web interfaces and APIs for machine learning models. Gradio has become a go-to tool for developers looking to bridge the gap between complex machine learning systems and end-users with minimal coding effort.

We will guide you through the process of designing the interface, showcasing how Gradio simplifies the creation of interactive components such as input forms, sliders, and real-time visual outputs. Alongside this, we will detail how we structured the API, leveraging Gradio's built-in features to expose key functionalities for seamless integration with other tools and applications.

This chapter also provides insight into the decision-making process that led us to choose Gradio. We'll discuss the factors that influenced our choice, such as ease of use, flexibility, and the ability to quickly iterate and deploy changes. By the end of this chapter, you'll have a clear understanding of how Gradio was used to enhance both the user experience and the technical integration capabilities.

9.2 Gradio

Gradio: is an open-source Python package that allows you to quickly **build** a demo or web application for your machine learning model, API, or any arbitrary Python function. You can then **share** a link to your demo or web application in just a few seconds using Gradio's built-in sharing features. No JavaScript, CSS, or web hosting experience needed.

9.2.1 Installing Gradio

Prerequisite: Gradio requires Python 3.10 or higher.

Installation: Gradio can be installed using pip, which is included by default in Python. To install it, run the following command in your terminal or command prompt**:**

```
pip install --upgrade gradio
```

Building a demo: We can run Gradio in our favorite code editor, Jupyter notebook, Google Colab, or any other environment where we write Python.

Here is a look on Gradio's first app:

```
import gradio as gr

def greet(name, intensity):
    return "Hello, " + name + "!" * int(intensity)

demo = gr.Interface(
    fn=greet,
    inputs=["text", "slider"],
    outputs=["text"],
)

demo.launch()
```

Figure 38: Example Gradio Code for Creating a Simple Web Interface

This code creates a Gradio app where the user inputs their name and selects an intensity level using a slider. The app then outputs a greeting message with the user's name, followed by an exclamation mark repeated according to the intensity value chosen.

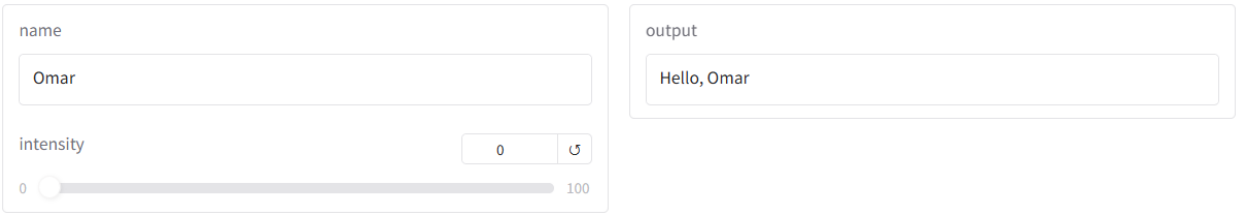


Figure 39: Gradio Interface Output Displaying Text and Slider Interaction

9.2.2 Creating APIs with Gradio

The first Gradio function we used to integrate Google's Speech-to-Text API with our custom logic for converting spoken language into 3D hand gestures representing sign language. This integration allowed us to seamlessly capture spoken input, convert it to text using Google's powerful speech recognition technology, and then process that text to generate corresponding 3D hand movements that illustrate the sign language translation. This functionality provided an interactive and dynamic way of transforming voice commands into visual sign language.

The second function, on the other hand, was dedicated to integrating our custom MediaPipe model, which was trained to recognize and translate various hand signs into text. The function utilized this model to analyze the input hand gestures, convert them to text, and then leverage the Google Text-to-Speech API to audibly communicate the translated text. This process enabled the app to provide a full cycle of sign language interaction, starting from hand gestures and transforming them into both textual and spoken responses, ensuring an effective and comprehensive communication system for users.

9.3 Sharing a demo

9.3.1 Sharing Gradio Demos Temporarily

In Gradio, sharing a demo is a straightforward process that allows you to quickly make your application accessible to others. When you launch your Gradio interface using the **demo.launch()** method, you can set the share parameter to True. This automatically generates a public URL that can be shared with anyone, enabling them to access and interact with the interface. The shared link is hosted temporarily and allows users to test and provide feedback on the application without requiring complex setup or deployment. This feature is especially useful for rapid prototyping, user testing, and collaboration, as it eliminates the need for setting up dedicated servers or hosting services. Gradio makes it easy to distribute your application for various purposes, such as sharing a model or collecting real-time feedback.

9.3.2 Deploying on Hugging Face Spaces

When sharing a Gradio demo, the provided link is temporary, hosted locally, and typically expires after 72 hours. While this is convenient for quick testing and short-term sharing, we aim to create a permanent link that can be shared publicly on the internet. To achieve this, we will deploy our Gradio app on **Hugging Face Spaces**, a platform specifically designed for hosting machine learning demos. Hugging Face Spaces offers a simple and reliable solution for creating a stable, always-accessible link, ensuring that our application is available to users without time restrictions.

```python
import gradio as gr

def greet(name):
    return "Hello " + name + "!"

demo = gr.Interface(fn=greet, inputs="textbox", outputs="textbox")

demo.launch(share=True)  # Share your demo with just 1 extra parameter 🚀
```

Figure 40: Simple Gradio Interface for Text Input and Output

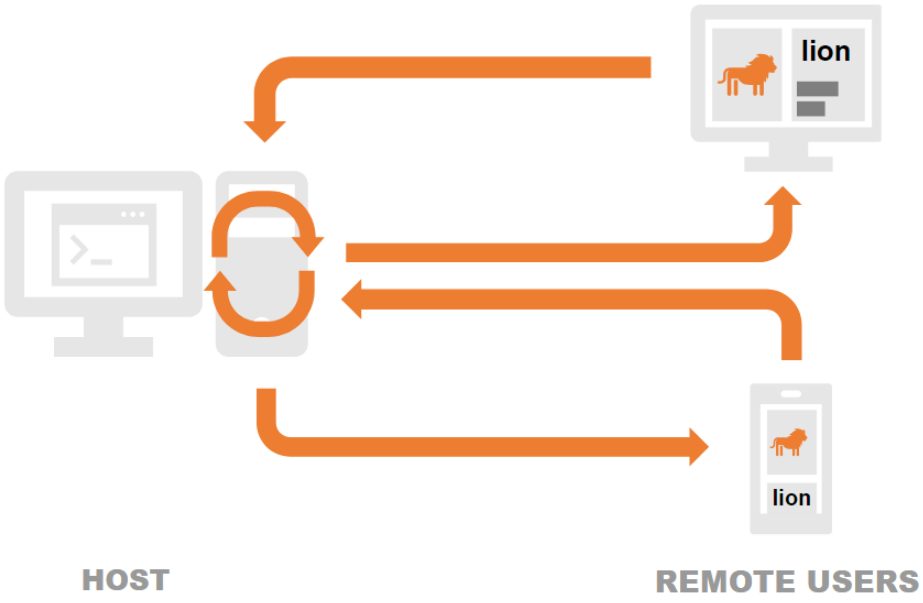


Figure 41: Interaction Flow Between Host and Remote Users in Gradio

9.4 Hugging Face Spaces

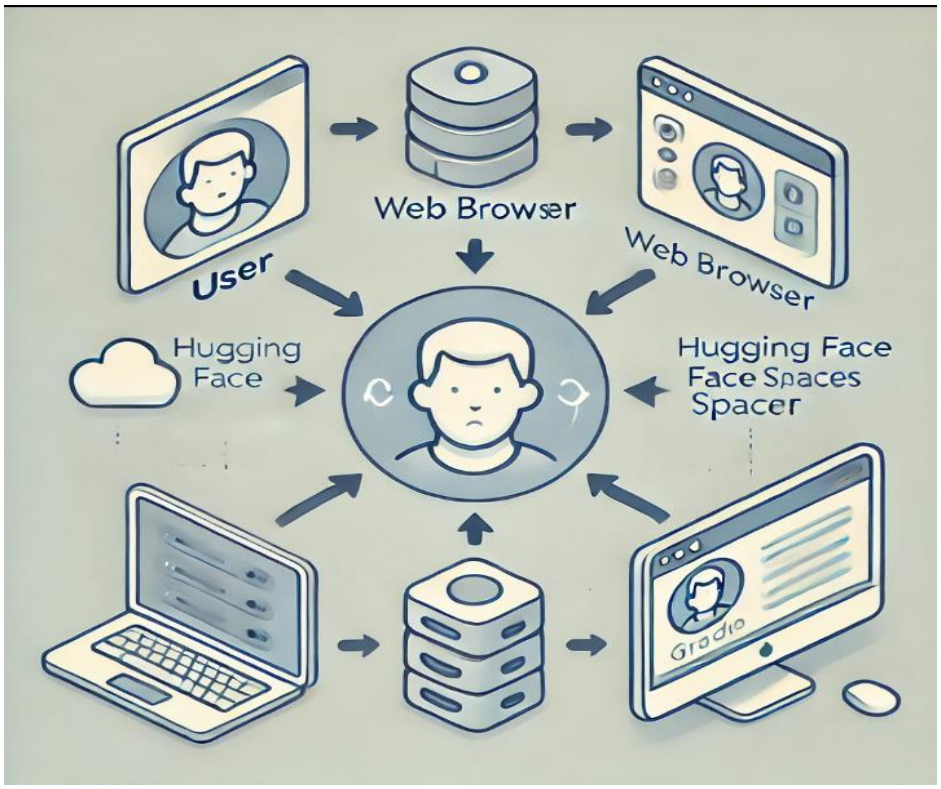


Figure 42: Ecosystem of Interactions Using Gradio and Hugging Face Spaces

9.4.1 Steps to Deploy on Hugging Face Spaces

To host and run our API permanently, we utilized **Hugging Face Spaces**, Hugging Face Spaces supports Gradio apps directly, making the deployment process straightforward and efficient. Here's how we set up our API on the platform:

We created a new repository on Hugging Face Spaces, selecting Gradio as the framework. This ensured that the platform would automatically recognize and support our Gradio app. Next, we upload the folder containing our Gradio application to Hugging Face Spaces. This folder includes all the necessary components, such as the Python script for the app, any additional files required for its functionality. By ensuring the folder is well-organized and complete, Hugging Face Spaces can seamlessly process and deploy the application. This step allows the platform to automatically install the dependencies, configure the environment, and host the Gradio app, providing us with a stable, publicly accessible link to our model. Once the files were successfully uploaded, Hugging Face Spaces automatically built and hosted the app on its

servers. It didn't take long before the app was fully set up, and as soon as the build process was completed, a permanent link to the application was provided. This link allowed us to access the app and share it with others seamlessly, making it easy to use and interact with the app instantly, no matter where we are located on the internet.

9.5 Api Page

9.5.1 Sign-to-Text Functionality

When using Gradio through an API, users can send input data such as images, videos, or text, and the model processes this input to return the result or prediction. In this context, a specific URL is used to access the hosted application, and the required data (such as a video in this case) is sent through the API interface. After the model processes the data, the result is returned and displayed to the user.

9.5.2 Sign-to-Video and Speech Functionality

When a video is uploaded, the code processes it frame by frame, detecting hand gestures that represent letters in sign language. Using the GestureRecognizer from MediaPipe, the system identifies these gestures and maps them to corresponding Arabic letters. As the video progresses, the system constructs a sentence by accumulating recognized gestures, confirming them once a specific threshold is reached. The Arabic letters are then drawn on the video, and the sentence is synthesized into speech using the Google Cloud Text-to-Speech API, providing an audio output that reads the translated text aloud.

The video is processed with a reduced frame rate for efficient gesture recognition, and the final output includes the constructed sentence both in text and speech, along with a video showing the detected gestures. Gradio serves as the interface that allows users to easily upload a video, process it, and receive the translated output in text, audio, and video formats.

This system combines multiple tools and APIs to provide a seamless experience for recognizing sign language, converting it to Arabic, and making it accessible through both visual and audio

means. It highlights the power of integrating AI models and cloud services to create a useful application for accessibility and communication.

9.5.3 Speech-to-Text Functionality

In this section, we explore how different technologies can be integrated to process and transcribe audio into text using external APIs and services. One approach involves using Gradio, a tool that allows seamless interaction with machine learning models hosted online. By leveraging Gradio, users can upload audio files, send them to a remote model, and receive predictions or results without needing to set up the backend infrastructure themselves. This simplifies the process of integrating audio-based models into applications, enabling quick and easy interaction with AI-powered services over the web.

On the other hand, Google Cloud's Speech-to-Text API offers a robust solution for transcribing audio into text. By sending an audio file to the cloud service, users can automatically convert spoken language into written text. The API is capable of processing audio in different languages and dialects, including Arabic. With its support for various audio configurations, such as sample rates and channel counts, it allows for accurate transcription in diverse conditions. This integration is valuable for applications in speech recognition, language translation, and accessibility, where converting spoken content to text can enhance user experience and interaction.

Both of these services demonstrate how modern cloud-based technologies can be combined to automate complex tasks like speech recognition and audio processing, creating tools that are accessible and easy to use without requiring in-depth technical expertise.

9.6 Conclusion

In this chapter, we have successfully deployed our sign language recognition and communication application, integrating key tools and technologies to achieve a seamless user experience. The deployment process was centered around Gradio for creating an intuitive user interface and API, and Hugging Face Spaces for hosting the application, ensuring accessibility and scalability.

We incorporated Google Cloud's Speech-to-Text and Text-to-Speech APIs to facilitate voice interaction, enabling the application to convert spoken language into text and vice versa. These components work in harmony, allowing users to interact with the system through voice commands and receive responses in a natural, auditory format.

The choice of Hugging Face Spaces as our deployment platform was strategic, offering ease of use and scalability, which are crucial for our project's growth and adaptability. This platform provided a robust environment to host our Gradio application, ensuring it remains accessible to users worldwide.

Throughout the deployment process, we encountered challenges such as ensuring smooth integration between Gradio and Google Cloud APIs. However, by meticulously configuring and testing each component, we overcame these obstacles, resulting in a cohesive and efficient application.

In conclusion, the successful deployment of our application marks a significant milestone in achieving our project's goals. It not only demonstrates the potential of combining advanced APIs and deployment platforms but also paves the way for enhancing communication accessibility for the deaf and mute community. The seamless interaction between Gradio, Hugging Face Spaces, and Google Cloud APIs underscores the power of modern technology in bridging communication gaps, setting a strong foundation for future enhancements and expansions.

REFERENCES

REFERENCES

[1] World Health Organization, "Deafness and hearing loss." https://www.who.int/news-room/fact-sheets/detail/deafness-and-hearing-loss.

[2] A. E. Morgan, S. M. El-Ghor, M. A. Khafagy, and H. S. Zaghloul, "Prevalence of Hearing Loss among Primary School Children in El-Mahalla El-Kubra District, Egypt," Egyptian Journal of Ear, Nose, Throat and Allied Sciences. Available: https://ejentas.journals.ekb.eg/article_210907_9ba04ce73e538c0536523937f4c41fef.pdf.

[3] M. Mustafa, "A study on Arabic sign language recognition for differently abled using advanced machine learning classifiers," Journal of Ambient Intelligence and Humanized Computing, 2020. doi: 10.1007/s12652-020-01790-w.

[4] M. Zaki, A. Mahmoud, S. Abd, and E. Mostafa, "Arabic Alphabet and Numbers Sign Language Recognition," International Journal of Advanced Computer Science and Applications, vol. 6, no. 11, pp. 209–214, 2015. doi: 10.14569/ijacsa.2015.061127. https://www.researchgate.net/publication/285755274_Arabic_Alphabet_and_Numbers_Sign_Language_Recognition

[5] V. N. T. Truong, C. K. Yang, and Q. V. Tran, "A translator for American sign language to text and speech," in Proceedings of the 2016 IEEE 5th Global Conference on Consumer Electronics (GCCE), 2016. doi: 10.1109/GCCE.2016.7800427. https://ieeexplore.ieee.org/document/7800427.

[6] R. Alzohairi, R. Alghonaim, W. Alshehri, S. Aloqeely, M. Alzaidan, and O. Bchir, "Image-based Arabic Sign Language recognition system," International Journal of Advanced Computer Science and Applications, vol. 9, no. 3, pp. 185–194, 2018. doi: 10.14569/IJACSA.2018.090327. https://www.researchgate.net/publication/324189192_Image_based_Arabic_Sign_Language_Recognition_System

[7] S. Hayani, M. Benaddy, O. El Meslouhi, and M. Kardouchi, "Arab Sign language Recognition with Convolutional Neural Networks," in Proceedings of the 2019 International Conference on Computer Science and Renewable Energies (ICCSRE), 2019, pp. 1–4. doi: 10.1109/ICCSRE.2019.8807586.

https://ieeexplore.ieee.org/document/8807586.

[8] Eman K Elsayed and Doaa R. Fathy, "Sign Language Semantic Translation System using Ontology and Deep Learning" International Journal of Advanced Computer Science and Applications(IJACSA), 11(1), 2020.

http://dx.doi.org/10.14569/IJACSA.2020.0110118

[9] H. Luqman and S. A. Mahmoud, "A machine translation system from Arabic sign language to Arabic," Universal Access in the Information Society, vol. 19, no. 4, pp. 891–904, 2020. doi: 10.1007/s10209-019-00695-6. Available: https://link.springer.com/article/10.1007/s10209-019-00695-6.

[10] MediaPipe Team, "MediaPipe: Main Documentation and Overview." https://ai.google.dev/edge/mediapipe/solutions/guide.

[11] MediaPipe Team, "MediaPipe Gesture Recognition: Main Documentation." Available: https://ai.google.dev/edge/mediapipe/solutions/vision/gesture_recognizer.

[12] MediaPipe Team, "MediaPipe Gesture Recognition: Python Documentation." https://ai.google.dev/edge/mediapipe/solutions/vision/gesture_recognizer/python.

[13] Google Cloud, "Speech-to-Text: API Requests Documentation." https://cloud.google.com/speech-to-text/docs/speech-to-text-requests.

[14] Google Cloud, "Text-to-Speech: Basics Documentation." https://cloud.google.com/text-to-speech/docs/basics.

[15] Sketchfab, "Robotic Hand 3D Model." Available: https://sketchfab.com/3d-models/robotic-hand-0a18fb697d1c4f9d9f743cede8593f1f.

[16] Blender, "Rigging Documentation." https://docs.blender.org/manual/nb/2.79/rigging/index.html.

[17] Blender, "Introduction to Inverse Kinematics in Bone Constraints." https://docs.blender.org/manual/en/latest/animation/armatures/posing/bone_constraints/inverse_kinematics/introduction.html.

[18] Blender, "Keyframe Animation Editing." https://docs.blender.org/manual/en/2.90/animation/keyframes/editing.html#.

[19] React Native, "React Native Main Documentation."

https://reactnative.dev.

[20] React Native, "React Native Render Pipeline Architecture." https://reactnative.dev/architecture/render-pipeline.

[21] React Native, "React Native Cross-Platform Implementation Architecture." https://reactnative.dev/architecture/xplat-implementation.

[22] Expo, "Expo Main Documentation."

https://expo.dev.

[23] TypeScript, "TypeScript Official Documentation."

https://www.typescriptlang.org.

[24] React Navigation, "Stack Navigator Documentation." https://reactnavigation.org/docs/stack-navigator/.

[25] Zustand, "Zustand Demo and Documentation."

https://zustand-demo.pmnd.rs.

[26] NPM, "React Query Package Documentation." https://www.npmjs.com/package/@tanstack/react-query

[27] Axios, "Axios Introduction Documentation."

https://axios-http.com/docs/intro.

[28] Node.js, "Node.js Official Documentation."

https://nodejs.org/en.

[29] Adobe, "Adobe XD UXP Developer Documentation." https://developer.adobe.com/xd/uxp.

[30] Figma, "Figma Help Center."

https://help.figma.com/hc/en-us.

[31] Adobe, "Photoshop User Guide."

https://helpx.adobe.com/photoshop/user-guide.html.

[32] Adobe, "Illustrator User Guide."

https://helpx.adobe.com/illustrator/user-guide.html.

[33] Gradio, "Quickstart Guide."

https://www.gradio.app/guides/quickstart.

[34] Gradio, "Sharing Your App: Sharing Demos." https://www.gradio.app/guides/sharing-your-app#sharing-demos.

[35] Gradio, "Sharing Your App: Hosting on Hugging Face Spaces." https://www.gradio.app/guides/sharing-your-app#hosting-on-hf-spaces.

[36] Gradio, "Sharing Your App: API Page."

https://www.gradio.app/guides/sharing-your-app#api-page.

[37] Hugging Face, "Hugging Face Documentation."

https://huggingface.co/docs.

[38] Expo, "Expo Documentation."

https://docs.expo.dev/.