Sign-Sync

Software Requirements Specification Document (Demo 2)

Member	Student Number
Michael Stone	u21497682
Matthew Gravette	u23545977
Wessel Johannes van der Walt	u22790919
Jamean Groenewald	u23524121
Stefan Muller	u22498622

Introduction

Sign Sync is a real-time translation system designed to bridge communication between spoken English and American Sign Language (ASL). The system allows users to input spoken or typed English and receive an accurate visual translation in ASL, using a combination of natural language processing, speech recognition, and gesture playback. It is developed as part of the COS301 Capstone Project to assist the Deaf and Hard-of-Hearing community.

User Characteristics

Deaf or Hard-of-Hearing users: Require real-time sign language output from spoken or written English.

Hearing individuals: May use the system to learn or communicate via ASL.

Interpreters: Professionals who need assistance in translating speech quickly.

Educators or Students: Learning environments using ASL as a teaching aid.

System Administrators: Maintain the deployed system and monitor performance.

User Stories

Deaf or Hard-of-Hearing Users

- View spoken language translated into sign animations.
- Use sign language to communicate back via webcam input.
- Adjust avatar display or gloss format for clarity.
- Access help or FAQs without requiring external assistance.
- Enable accessibility features (e.g., high-contrast mode, larger text, slowed animation).

Hearing Users

- Speak naturally and have their speech translated into signs.
- Read or hear signed responses translated to text or audio.
- Type messages and see the corresponding ASL gloss and animation.
- Use the system as a teaching or demonstration tool for learning ASL.
- Select different visual styles or avatars for sign output.

Administrators / Researchers

- Monitor system performance and translation accuracy.
- Gather flagged feedback data for AI retraining.
- Manage user access and customization settings.
- Review usage logs and system health metrics.
- Push updated models or gloss rules into the pipeline.

Educators / Interpreters

- Demonstrate real-time speech-to-sign translation in classrooms or training.
- Share a live session with multiple learners.
- Export gloss translations for lesson plans.
- Annotate signs or glosses for specific learning contexts.

Functional Requirements

R1: Text-to-Sign Translator

R1.1: Capture text input from user.
R1.2: Translate English text to Sign gloss.
R1.3: Search word definition for appropriate sign.
R1.4: Display sign through avatar.
R1.5: Handle grammar transformations for accurate ASL structure

R2: Sign-to-Text Translator

R2.1: Capture webcam input and extract hand keypoints.
R2.2: Classify sign gesture sequences using trained AI models.
R2.3: Convert recognized signs to sign gloss.
R2.4: Convert sign gloss to English.
R2.5: Show real-time visual feedback of detected signs.

R3: Feedback and AI Improvement System

R3.1: Allow users to flag inaccurate translations.
R3.2: Log flagged data.
R3.3: Retrain AI models periodically using collected data.
R3.4: Provide an admin dashboard to review flagged entries and retraining logs

R4: User Interface

R4.1: Display live avatar animations based on translation output.
R4.2: Show translated text and/or play voice feedback.
R4.3: Offer accessibility options (high-contrast mode, font scaling, voice personas).
R4.4: Provide a help menu with tutorials and FAQs.
R4.5: Support multi-device layouts (mobile/tablet/desktop).

R5: User Management and Settings

R5.1: Authenticate users.
R5.2: Store and retrieve user preferences and settings.
R5.3: Allow users to login.
R5.4: Allow users to register.
R5.5: Allow role-based access for admins, researchers, and general users.

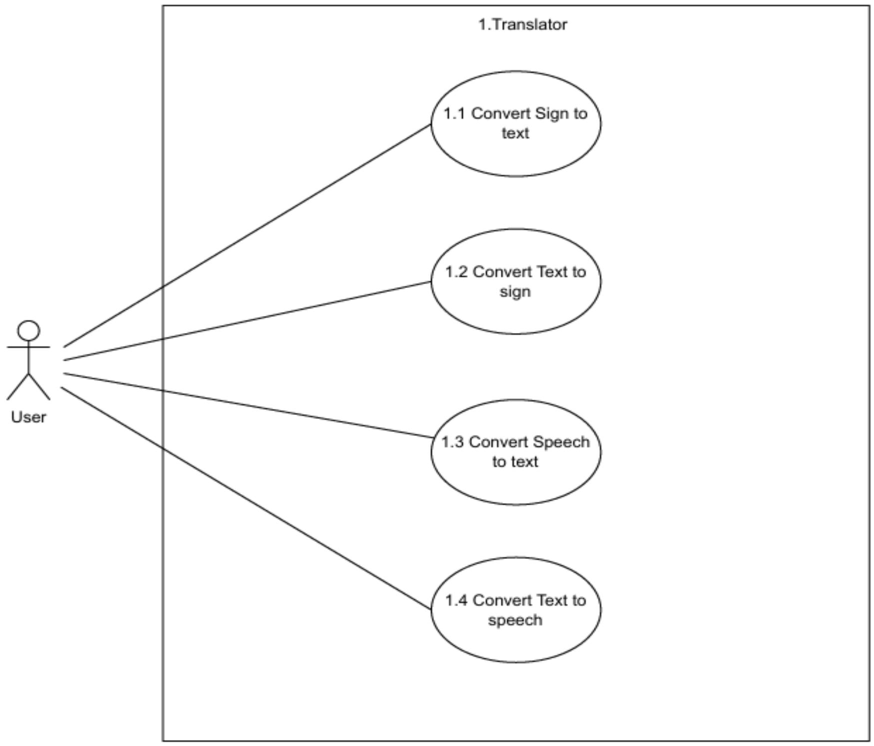
R6: Speech to Text

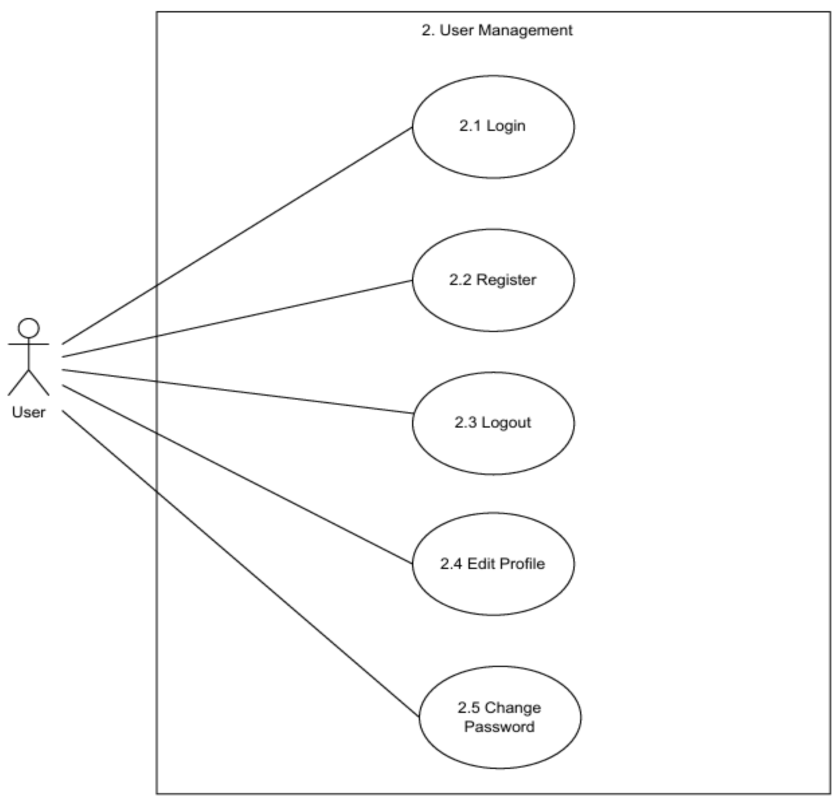
- **R6.1**: Capture user speech.
- **R6.2**: Convert speech to text.
- **R6.3**: Display text on screen.
- **R6.4:** Send text into ASL gloss converter pipeline.

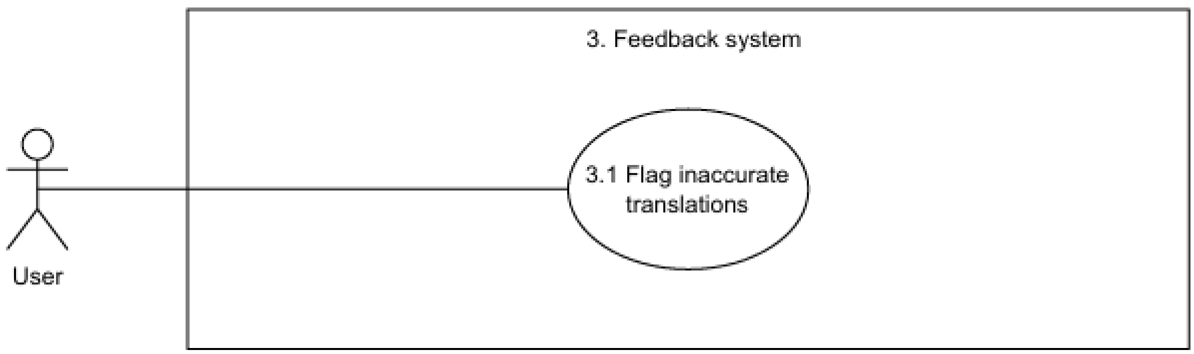
R7: Text to Speech

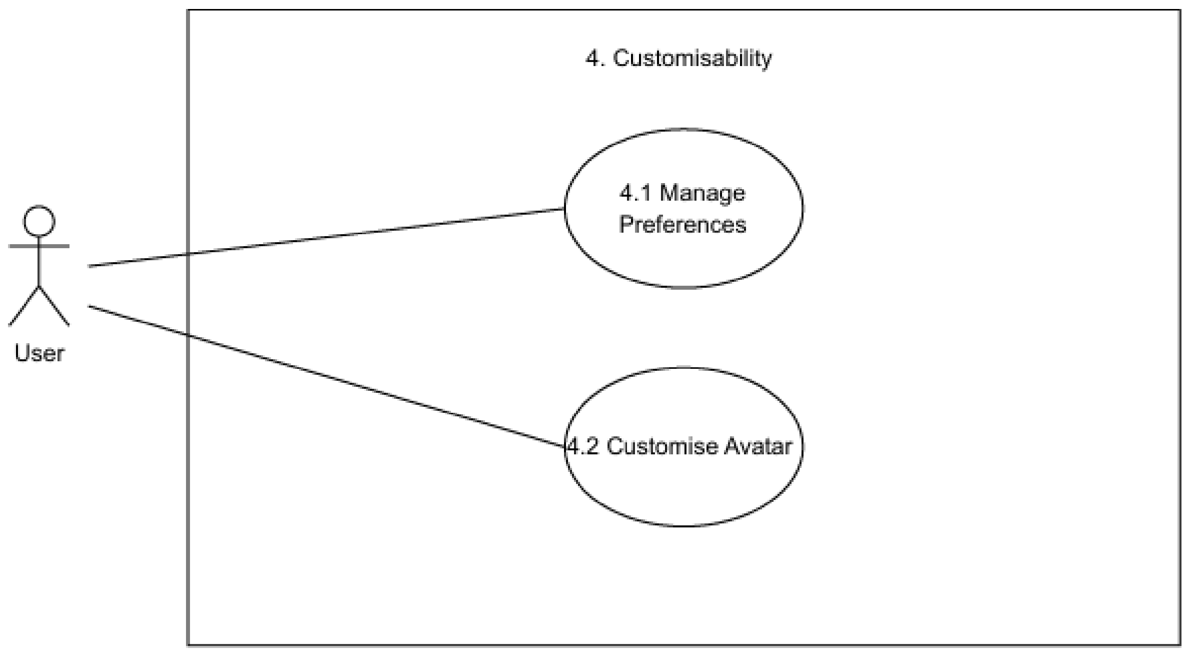
- **R7.1**: Capture text input by user.
- **R7.2**: Convert text to speech.
- **R7.3**: Play speech for the user to hear.
- **R7.4:** Support voice personalization settings.

Use Case Diagrams

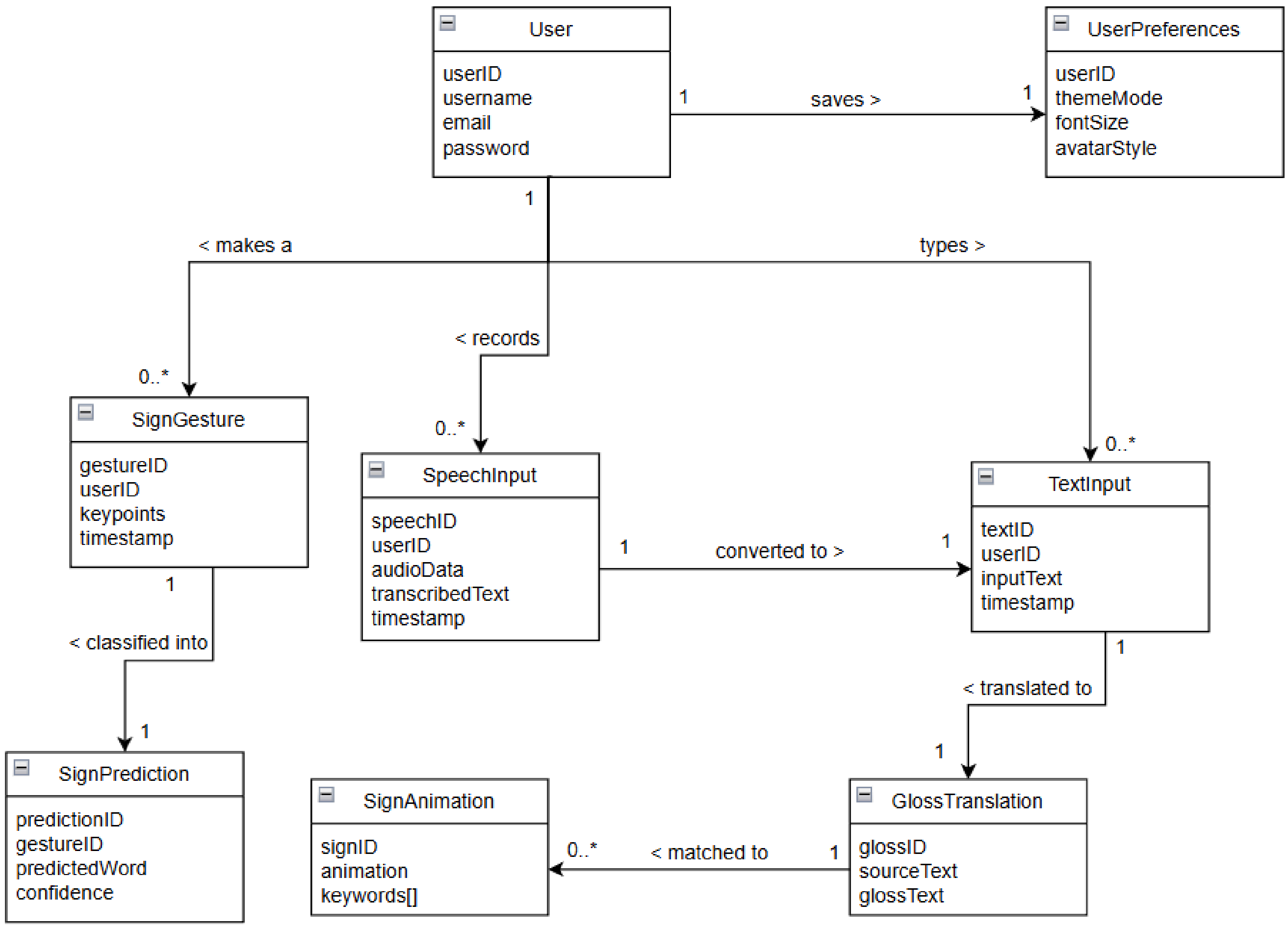




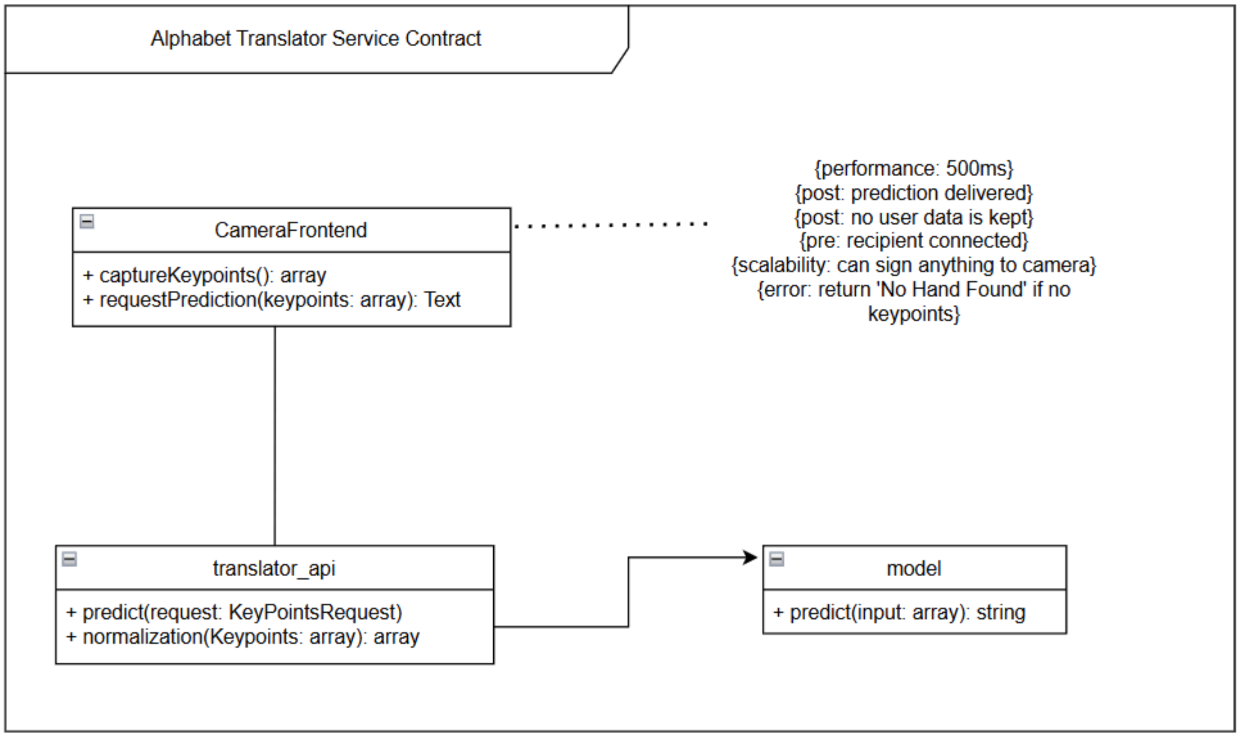

3. Feedback system
3.1 Flag inaccurate translations
User

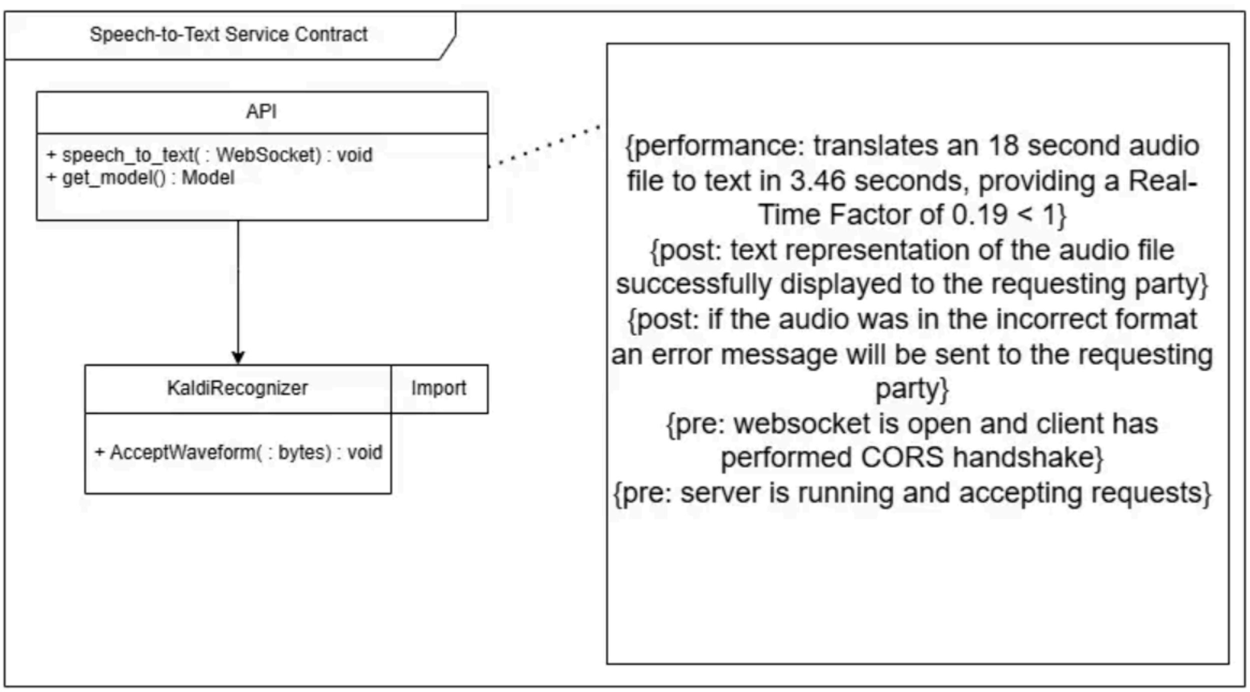

4. Customisability
4.1 Manage Preferences
4.2 Customise Avatar
User

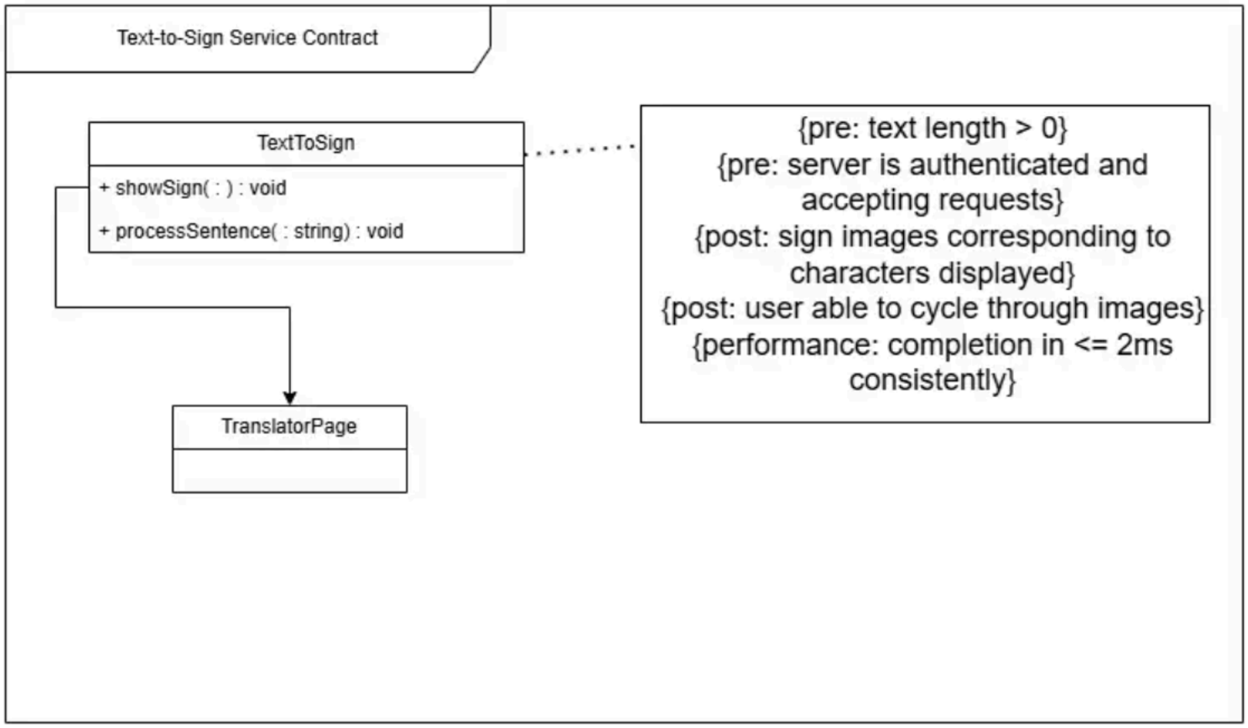
Domain Model



Service Contracts (Graded in demo 3)






Text-to-Sign Service Contract
TextToSign
+ showSign(:) : void
+ processSentence(: string) : void
TranslatorPage
{pre: text length > 0}
{pre: server is authenticated and accepting requests}
{post: sign images corresponding to characters displayed}
{post: user able to cycle through images}
{performance: completion in <= 2ms consistently}

Architectural Requirements

Sign-Sync: Architectural Requirements Document

Technology Choices

Frontend Framework

Framework	**Pros**	**Cons**
Angular	• Full-featured MVC framework • Large enterprise support	• Steep learning curve • Heavy bundle size
React	• Component-based • Huge ecosystem and community • Easy Websocket integration	• State management can be difficult • Setup can be tedious
Svelte	• Compiles to vanilla JS • Fast performance	• Less enterprise adoption • Smaller ecosystem

Choice:
React was selected due to its modular structure, vibrant and large ecosystem and ease of integrating real-time features such as websockets. This aligns well with the microservices architecture and enables a maintainable, scalable frontend.

Backend Language

Language	Pros	Cons
Python	• Large AI/ML ecosystem • Simple, readable syntax • Strong library support	• Slower runtime • Not ideal for multi-threading
JavaScript (Node.js)	• Full-stack JS • Large NPM ecosystem	• Difficulty in debugging • Complex async handling
Go	• Excellent concurrency • Fast Performance	• Limited AI/ML libraries

Choice:
Python was chosen for backend services, especially AI-related modules, due to its excellent support for ML and NLP libraries, such as spaCy and Vosk. While it is not the fastest, its developer productivity and expressiveness make it ideal for rapidly developing and deploying independent services. This aligns perfectly with the microservices architecture.

API Framework

Framework	Pros	Cons
ExpressJS	• Minimal and flexible • Well-established • Fast setup	• Requires manual validation • Not type-safe
FastAPI	• Fast, async support • Easy validation with Pydantic • Auto-generated docs	• Lacks some mature integrations • Still relatively new
Flask	• Lightweight • Simple for quicks APIs • Mature and stable	• Not async by default • Less scalable for real-time

Choice:
FastAPI was chosen as our API framework since it supports our microservice architecture with its async design, fast performance and modular structure. Each microservice can be independently built and deployed using this framework which ensures scalability and maintainability.

Database

DB	Pros	Cons
MongoDB	• NoSQL, flexible schema • Document-oriented (therefore great for JSON data)	• Less suitable for relational data • Data consistency is not always guaranteed
PostgreSQL	• Strong ACID compliance • Complex querying	• Requires fixed schema • Slightly more setup for scaling
Firebase Realtime DB	• Real Time sync • Easy to use • Scales well	• Less control over backend logic • No relational structure

Choice:
MongoDB was chosen due to its document-oriented structure which fits well with storing user preferences and data and loosely structured data. It also complements a microservices setup by being easy to scale independently per service.

Speech Recognition

Model	Pros	Cons
Mozilla DeepSpeech	• Open source • Good accuracy • Active community	• Large models • High resource usage
Vosk	• Free • Fast and multilingual • Real-time • Raw byte streams	• Limited documentation • Smaller community
Google Speech API	• Very high accuracy • Robust language support	• Cloud-only • Latency • Usage cost

Choice:
We chose Vosk because it runs offline, supports real-time transcription and integrates easily into independent microservices without relying on external APIs. This is crucial for maintaining modularity and reducing latency in a distributed architecture.

NLP Processing

Model	Pros	Cons
spaCy	• Lightweight • Pretrained models • Easy to integrate	• Limited deep semantic analysis
NLTK	• Rich library for NLP education/research	• Slower • Outdated for production systems
HuggingFace Transformers	• State-of-the-art models • flexible	• Heavier • Complex integration

Choice:
spaCy was chosen for its speed and simplicity which is ideal for real-time language processing within our NLP microservice. Its modularity ensures each NLP-related function can scale and update independently in the overall architecture.

Gesture Recognition

Model	**Pros**	**Cons**
TensorFlow (TCN)	• Great for temporal sequences • Memory efficient	• Steeper learning curve • Requires model tuning
PyTorch (LSTM)	• Dynamic graph • Easy debugging	• Slower in production • Less optimised for mobile
MediaPipe	• Fast • Easy gesture pipelines	• Limited customisation • Black-box components

Choice:
TensorFlow with Temporal Convolutional Networks (TCNs) was chosen due to their strong performance in recognising sequences, such as gestures. These models are containerised and deployed as an isolated microservice which aligns well with our architecture's need for scalable, efficient model inference.

Hand Recognition

Model	Pros	Cons
OpenCV	• Lightweight • Cross-platform • Integrates well with Python	• Requires manual tuning • No built-in hand detection
MediaPipe	• Fast • Pretrained hand landmark detection	• Harder to customise • Black-box components
OpenPose	• Highly accurate for full body/hands	• Heavy • GPU-dependent • Harder to deploy at scale

Choice:
OpenCV and MediaPipe as they are easy to integrate. They are flexible and lightweight which makes it ideal for our hand recognition microservice. It enables fine-tuned control and, when containerised, it integrates smoothly into the microservices environment without excessive resource demands.

<u>Hosting</u>

Service	**Pros**	**Cons**
Amazon Web Services (AWS)	• Highly scalable and battle-tested • Offers free-tier services (EC2, S3, Lambda) suitable for MVP deployments • Excellent integration with Docker, API Gateways, and CI/CD tools	• Complex initial setup • Steeper learning curve for new developers • Cost increases quickly beyond the free tier
Google Cloud Platform (GCP)	• Excellent for containerized deployments (e.g., Cloud Run, GKE) • Great NLP/AI service integrations if needed in future • Free-tier credits for students and education teams	• Fewer community resources/tutorials compared to AWS • Region-specific performance may vary
Microsoft Azure	• Strong enterprise integrations and CI/CD via GitHub Actions • Azure App Service is simple for deploying Python + React apps • Offers educational credits for students	• Documentation is sometimes inconsistent • Slightly more expensive for persistent container hosting than GCP

Choice:
Still being discussed with the client.

Deployment Plan/Model (Demo 3)

The Sign Sync system is designed to run as a containerized, microservice-based application, which allows us to deploy each translation component independently. The deployment model is cloud-ready and optimized for scalability, portability, and modularity.

- **Frontend** (React) — served via Azure App Service or S3 + CloudFront (AWS)
- **Backend Microservices** (FastAPI) — containerized and deployed using Docker on Microsoft Azure or AWS
- **Database** (MongoDB) — hosted via MongoDB Atlas or Azure Cosmos DB with Mongo API
- **WebSocket-enabled Speech Service** — exposed through secure public endpoints
- **CI/CD Pipeline** — automated builds and deployments via GitHub Actions, with deployment scripts targeting Azure or AWS services

Live Deployed System (Demo 4)

Work towards in demo 3