# Sign-Sync

## Architectural Requirements Document
## (Demo 2)

APOLLO PROJECTS

GENDAC SOFTWARE ENGINEERING

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

## Architectural Design Strategy

Sign Sync follows a decomposition-based architectural design strategy. This strategy was chosen to support the system's scalability, modularity, and maintainability, given the diverse range of translation modes involved:

- Sign to Text
- Sign to Speech
- Text to Sign
- Speech to Sign

By decomposing the system into independent components, each translation mode can be developed, tested, and improved in isolation without affecting other parts of the system. For example, the sign recognition module can evolve independently from the speech-to-text module, enabling focused development and easier debugging.

This modularity also allows different teams or developers to work on separate components simultaneously and supports future integration of new translation types or model improvements without rearchitecting the entire system.

While other strategies such as quality-attribute-driven design and test-driven design were considered and partially influenced component decisions (e.g., CI/CD setup for maintainability and testability), they were secondary to decomposition, which remains the dominant strategy shaping the system's architecture.

## Architectural Strategy

### Microservices

The primary architectural style adopted for Sign Sync is the Microservices Architecture, which directly complements the team's chosen decomposition design strategy. Each translation function—Sign to Text, Sign to Speech, Text to Sign, and Speech to Sign—is implemented as an independent microservice with its own API and corresponding frontend React component.

### Components

- Individual translation services (e.g., gesture recognition, speech-to-text)
- Frontend clients consuming each microservice via HTTP or WebSocket
- Shared services, such as the gloss converter and avatar renderer

### Connectors

- REST APIs for synchronous communication between frontend and microservices
- WebSockets for real-time streaming (speech-to-text)
- Shared MongoDB for storing sign animations and keyword mappings

### Constraints

- Stateless microservices to allow easy deployment and horizontal scaling
- Each service encapsulates its own logic and dependencies to reduce coupling

**This architecture significantly improves both maintainability and scalability**:

- Each service can be developed, tested, deployed, and scaled independently.
- Developers can work on their assigned service without being blocked by others.
- Bug fixes or enhancements in one component have minimal risk of breaking others.

Additionally, this strategy made project coordination easier. Team members could each take ownership of one translation mode or subsystem, enabling parallel development with minimal conflict or dependency overhead.

Other styles such as Monolithic or Layered architecture were considered, but they lacked the modular flexibility and deployment agility needed for a system with real-time, multi-modal translation services.

## Architectural Quality Requirements

| Rank | Quality Requirement | Measurement |
|---|---|---|
| **1** | Usability | - System offers 3 theme modes: Light, Dark, and High Contrast<br>- Users can select font size: small, medium, or large<br>- A help menu and user guide are accessible via the main UI<br>- Meets WCAG 2.1 AA accessibility standards |
| **2** | Reliability | - AI models for Sign-to-Text and Speech-to-Text must achieve ≥ 85% accuracy on test datasets<br>- Translation outputs must be consistently correct under real-time usage conditions |
| **3** | Scalability | - The system must support ≥ 10 concurrent users submitting translation requests (speech, text, or sign)<br>- Average response time per request must remain ≤ 2 seconds under this load |
| **4** | Security | - User passwords must be securely stored using a strong hashing algorithm.<br>- Database connection strings and other sensitive configuration data must be stored in environment variables.<br>- Sensitive information, such as hashed passwords must not be exposed in API responses, logs or client-side code.<br>- Verified via manual code review and security test cases |

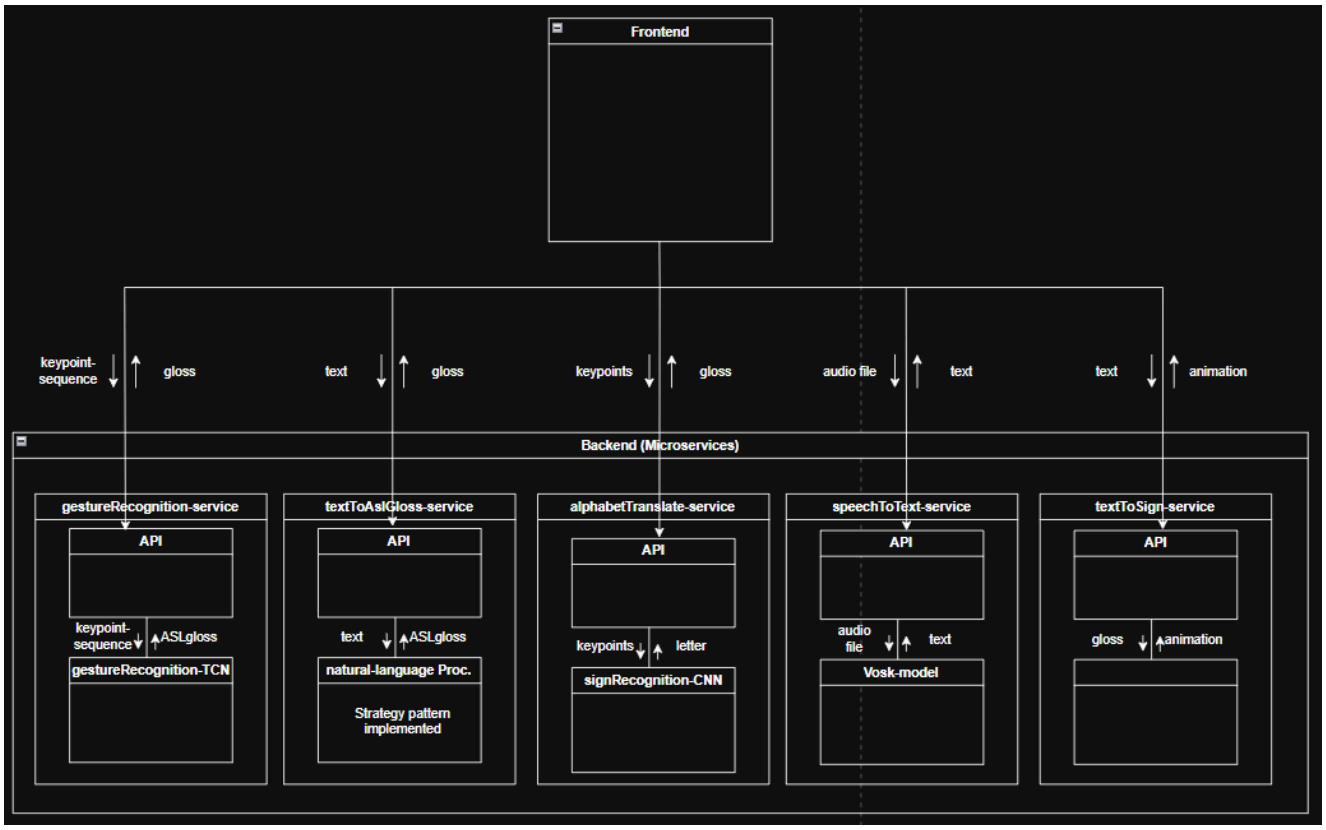
| | | |
|---|---|---|
| **5** | Maintainability | - All services must follow a modular, single-responsibility architecture<br>- All APIs and use cases must be fully documented (Swagger or markdown)<br>- At least 90% of logic-layer functions must be covered by unit tests, measured using coverage tools<br>- CI pipelines must run on every commit to verify regressions |

# Architectural Design and Pattern

## Architectural Patterns:

**Architecture**: Microservice Architecture
**Justification**:

- Each core function (speech recognition, gloss conversion, UI renderer) is isolated as a distinct service and services communicate via REST API or Websockets (real-time speech) promoting low coupling, high cohesion and:
  - Maintainability
  - Modifiability
  - Scalability (independent deployment)
  - Modularity
  - Testability
  - Reusability



## Design Patterns:

**Observer Pattern**:
**Used in:** Frontend (React) with WebSocket connection

**Purpose:** Enables real-time updates from the backend to automatically update the UI without polling.

**Example:** When a user speaks, the transcription stream from the speech-to-text service pushes data to the frontend in real-time. The UI observes changes and re-renders the output text or ASL gloss live.

**Supports:** Usability, Responsiveness

**Factory Pattern**:
**Used in:** Sign animation and avatar rendering

**Purpose:** Dynamically creates the correct sign animation (video/image/fingerspelling) based on the input gloss or keyword.

**Example:** Given a gloss term like "eat," the system uses a factory to determine whether a specific animation exists in the database or whether to fall back to fingerspelling.

**Supports:** Maintainability, Extensibility

**Strategy Pattern**:
**Used in:** Gloss conversion engine

**Purpose:** Enables flexible switching between different translation strategies — rule-based, phrase-based lookup, or machine learning fallback.

**Example:** The text-to-gloss converter first attempts a rule-based parse; if that fails, it can switch to an ML-based strategy seamlessly.

**Supports:** Flexibility, Accuracy, Maintainability

# Architectural Constraints

The architecture of Sign Sync has been shaped by practical, ethical, and technical limitations that reflect the project's inclusive mission and long-term deployment goals. These constraints guided decisions related to system modularity, privacy, accessibility, deployment, and team workflow. Each constraint listed below is directly aligned with the project’s stakeholders: Deaf and Hard-of-Hearing users, technical administrators, and future contributors.

**1. Accessibility and Inclusion Constraint**

The application must be accessible to users with diverse abilities, including Deaf and Hard-of-Hearing individuals, as well as users with visual or motor impairments.

As accessibility is a core value of the Sign Sync project, the system is designed with inclusive features from the ground up. This constraint impacts both frontend design and backend responsiveness:

- Support for theme variations including dark mode and high contrast mode
- Adjustable font size and readable typefaces
- Avatar animations for sign output, reducing reliance on text-based feedback
- Speech-to-text fallback for users with motor limitations
- Keyboard navigability and support for screen readers via semantic HTML and ARIA tags

This ensures that the system offers an equitable experience to users regardless of ability.

### 2. Privacy and Data Minimization Constraint

The system must limit data collection to what is strictly necessary for functionality and protect any sensitive information involved in translation tasks.

Because Sign Sync collects inputs like webcam footage and voice recordings, data protection is a critical architectural concern. The system adheres to POPIA (South Africa) and GDPR (EU) principles through:

- Secure password storage using hashing algorithms (e.g., bcrypt)
- No storage of raw audio/video inputs unless explicitly enabled for training or feedback
- Anonymization of any user feedback data used for AI model retraining
- Isolation of sensitive configuration data using environment variables
- Clear separation of frontend and backend concerns to reduce exposure risks

### 3. Platform Responsiveness Constraint

The system must function effectively across a range of screen sizes and device types, without sacrificing usability or performance.

Sign Sync users may access the platform via desktop or tablet interfaces. To accommodate this, the system was designed using responsive and device-independent practices:

- Responsive layout design using utility-first CSS (Tailwind)
- Component layouts that adapt cleanly between mobile, tablet, and desktop views
- Compatibility testing across Chromium-based browsers and Firefox
- Consistent avatar rendering and translation display regardless of viewport

**4. Modular Deployment Constraint**

The system must be deployable using scalable, container-friendly infrastructure and remain adaptable for future expansion.

Sign Sync is built using a microservices architecture where each translation function (e.g., speech-to-text, sign-to-text) operates as an independent service. This design introduces the following constraints:

- Backend services must be Docker-compatible for cloud deployment
- APIs must be stateless to support horizontal scaling
- Frontend and backend must communicate via well-defined interfaces (REST/WebSocket)
- Services must remain loosely coupled to support easy upgrades or substitution (e.g., swapping in a new ASL model)

This ensures that the system can scale and evolve without major architectural overhauls.

**5. Team and Time Constraint**

The system must be feasible for a small development team to build within a university semester while maintaining quality and modularity.

To accommodate the academic timeline and limited team size:

- Development followed a decomposition approach where each team member owned one microservice
- Technologies were chosen based on familiarity (e.g., React, FastAPI, Python, MongoDB)

- Components were isolated to enable parallel development without merge conflicts
- Features that require significant infrastructure (e.g., real-time avatar lip-syncing) were deferred to future phases

# Technology Choices

## Frontend Framework

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

## <u>Backend Language</u>

| Language | Pros | Cons |
|---|---|---|
| Python | ● Large AI/ML ecosystem<br>● Simple, readable syntax<br>● Strong library support | ● Slower runtime<br>● Not ideal for multi-threading |
| JavaScript (Node.js) | ● Full-stack JS<br>● Large NPM ecosystem | ● Difficulty in debugging<br>● Complex async handling |
| Go | ● Excellent concurrency<br>● 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<br>● Well-established<br>● Fast setup | ● Requires manual validation<br>● Not type-safe |
| FastAPI | ● Fast, async support<br>● Easy validation with Pydantic<br>● Auto-generated docs | ● Lacks some mature integrations<br>● Still relatively new |
| Flask | ● Lightweight<br>● Simple for quicks APIs<br>● Mature and stable | ● Not async by default<br>● 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<br>● Document-oriented (therefore great for JSON data) | ● Less suitable for relational data<br>● Data consistency is not always guaranteed |
| PostgreSQL | ● Strong ACID compliance<br>● Complex querying | ● Requires fixed schema<br>● Slightly more setup for scaling |
| Firebase Realtime DB | ● Real Time sync<br>● Easy to use<br>● Scales well | ● Less control over backend logic<br>● 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<br>● Good accuracy<br>● Active community | ● Large models<br>● High resource usage |
| Vosk | ● Free<br>● Fast and multilingual<br>● Real-time<br>● Raw byte streams | ● Limited documentation<br>● Smaller community |
| Google Speech API | ● Very high accuracy<br>● Robust language support | ● Cloud-only<br>● Latency<br>● 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<br>● Pretrained models<br>● Easy to integrate | ● Limited deep semantic analysis |
| NLTK | ● Rich library for NLP education/research | ● Slower<br>● Outdated for production systems |
| HuggingFace Transformers | ● State-of-the-art models<br>● flexible | ● Heavier<br>● 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<br>● Memory efficient | ● Steeper learning curve<br>● Requires model tuning |
| PyTorch (LSTM) | ● Dynamic graph<br>● Easy debugging | ● Slower in production<br>● Less optimised for mobile |
| MediaPipe | ● Fast<br>● Easy gesture pipelines | ● Limited customisation<br>● 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<br>● Cross-platform<br>● Integrates well with Python | ● Requires manual tuning<br>● No built-in hand detection |
| MediaPipe | ● Fast<br>● Pretrained hand landmark detection | ● Harder to customise<br>● Black-box components |
| OpenPose | ● Highly accurate for full body/hands | ● Heavy<br>● GPU-dependent<br>● 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.

**Hosting**

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

**Choice:**
Still being discussed with the client.