Sign-Sync

Architectural Requirements Document
(Demo 4)

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.

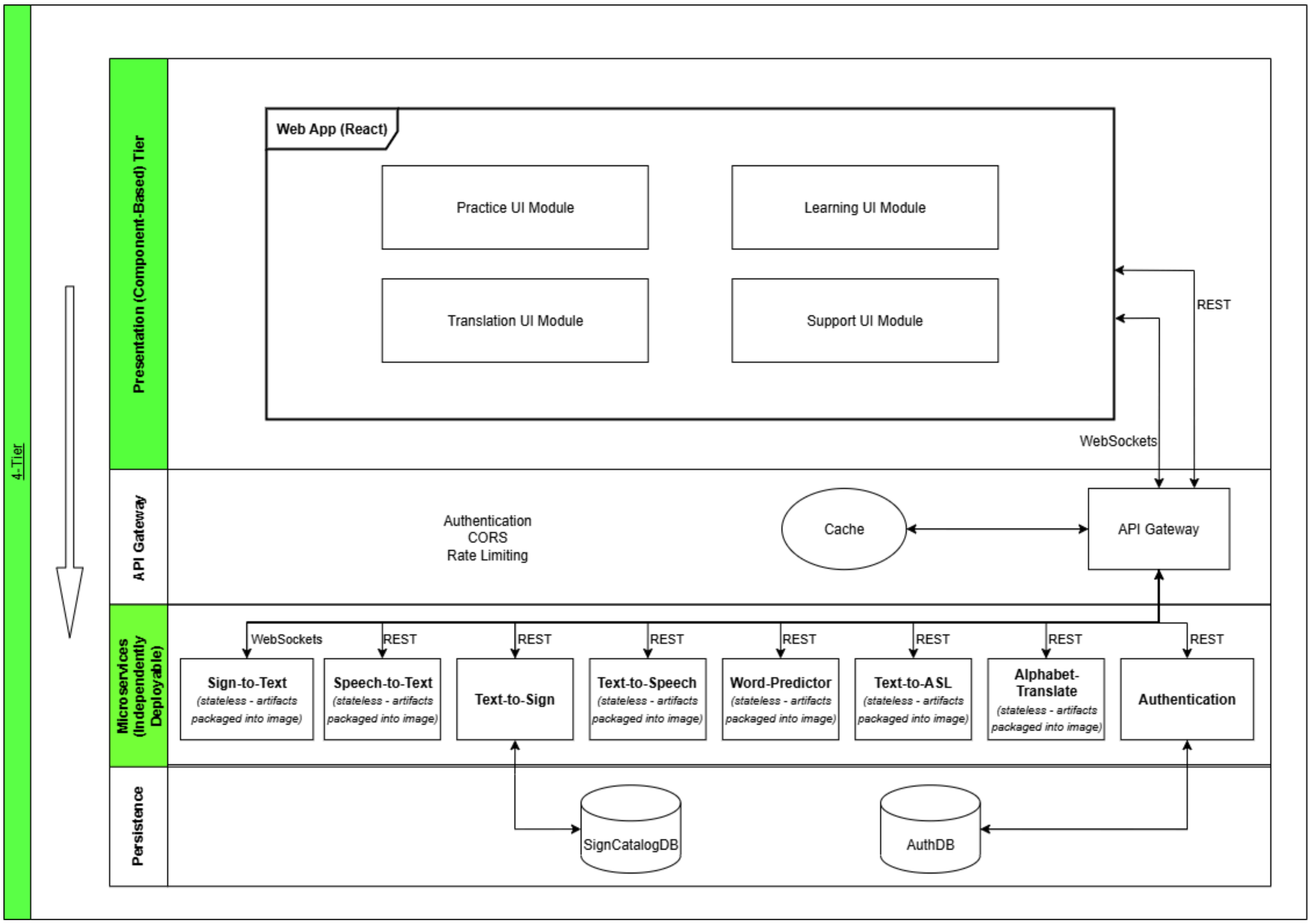
Other styles such as Monolithic or Layered architecture were considered, but they lacked the modular flexibility and deployment agility needed for a system with real-time, multi-modal translation services.

Quality Requirements

Rank	Requirement Quality	Measurement
1	Usability	System offers 2 theme modes: Light, Dark Users can select font size: small, medium, or large. As well as animation speed and voice customization. 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 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) Average response time per request must remain ≤ 2 seconds under this load
4	Security	User passwords must be securely stored using Bcrypt Database connection strings and other sensitive configuration data must be stored in environment variables. Sensitive information, such as hashed passwords must not be exposed in API responses, logs or client-side code. Verified via manual code review and security test cases
5	Maintainability	All services must follow a modular, single-responsibility architecture All APIs and use cases must be fully documented At least 90% of logic-layer functions must be covered by unit tests, measured using coverage tools CI pipelines must run on every commit to verify regressions

Architectural Design, Pattern and Diagram

Architectural Diagram:



For better view:
Apollo Projects (Sign-Sync): Architectural Diagram

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

Architecture: Component-based
Justification:

- Since microservices can scale considerably, the frontend must be able to use the plethora of services in an efficient, sustainable manner without bloating modules. The component based architecture enables separation of concerns and the following quality attributes which enables the application to harvest the potential of growing microservices:
  - Reusability
  - Testability
  - Scalability

Architecture: N-tier
Justification:

  - The above mentioned patterns can scale efficiently in their local scope, however the two subsystems (component-based frontend, microservices) must be able to scale independently of each other. Furthermore, ensuring security of large applications is difficult. Destructuring a system into logical layers, which can add accumulating layers of security, provides a sufficient foundation for security. Hence, we have a Component-based frontend layer, API gateway layer, microservices layer and persistence layer. The following quality attributes as well as separation of concerns can be realized with the N-tier architecture:
    - Security

- Maintainability
- Flexibility
- Reusability
- Scalability

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 grow without 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 • 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.

Hosting

Service	Pros	Cons
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
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

Choice:
Client, Gendac, has more experience with Microsoft Azure so could provide more insight and assistance in the creation, initialization and deployment of the system using Azure

Service Contracts

Note: Since the system is not yet deployed base URLs are omitted

<table>
<tr><th colspan="4"><u>Word Prediction Service Contract</u></th></tr>
<tr><td colspan="2">What it does</td><td colspan="2">Predicts next word following a sequence of previous words
Converts sequence of words in ASL grammar to a normal english sentence</td></tr>
<tr><td colspan="2">Protocol</td><td colspan="2">HTTP</td></tr>
<tr><td>Endpoint</td><td>Method</td><td>Input</td><td>Output</td></tr>
<tr><td>/predict</td><td>POST</td><td>(JSON)
{
sentence: String,
*add_k: Float=0.0,
*min_count: Integer=1,
*backoff: Boolean=False,
}

sentences- sequence of words, prefix of to be predicted word
add_k- degree of smoothing to better represent less seen words
min_count: minimal count to consider. Only used when add_k=0
backoff: retries with subset of prefix if not found</td><td>(JSON)
{
token: String | null,
prob: Float
}

token- predicted next word based on prefix
prob- probability score</td></tr>
<tr><td>/translate</td><td>POST</td><td>(JSON)
{
text: String
}</td><td>(JSON)
{
translation: String
}</td></tr>
</table>

API Gateway Service Contract	
What it does	• Acts as a singular entry point to all backend services hiding the frontend from the microservices allowing scalability and maintainability. Enables separation of concerns - microservices do not need to implement all the security and additional functionality. • Proxies requests from frontend to microservices • Supports http requests and websocket connections • Provides security in the form of CORS, Authentication, API keys. • Provides caching • Provides rate limiting
Protocol	HTTP / Websockets
Methods	GET, POST, PUT, DELETE, PATCH, OPTIONS
Endpoint	Service
/api/auth	Authentication Service
/api/speech	Speech-to-Text Service
/api/asl	Text-to-ASL-gloss Service
/api/alphabet	Alphabet-translate Service
/api/word	Word Prediction Service
/api/sign	Text-to-Sign Service
/api/stt	Sign-to-Text Service

<table>
<tr><th colspan="4"><u>Text-to-Sign Service Contract</u></th></tr>
<tr><td colspan="2">What it does</td><td colspan="2">Takes in an ASL gloss and returns animation names. If the sign for the corresponding ASL gloss is not in the database, an array of the letter-animations is returned (essentially spelling out the word in signs). Otherwise, the animation name for the sign is returned.</td></tr>
<tr><td colspan="2">Protocol</td><td colspan="2">HTTP</td></tr>
<tr><td>Endpoint</td><td>Method</td><td>Input</td><td>Output</td></tr>
<tr><td>/getAnimation</td><td>POST</td><td>(JSON)
{
word: String (ASL gloss)
}</td><td>(JSON)
Sign supported:
{
response: String (animation name)
}

Sign not supported:
{
response: String[] (alphabet animation names)
}</td></tr>
</table>

<table>
<tr><th colspan="4"><u>Text-to-ASL-gloss Service Contract</u></th></tr>
<tr><td colspan="2">What it does</td><td colspan="2">Converts normal english text to an ASL word/gloss</td></tr>
<tr><td colspan="2">Protocol</td><td colspan="2">HTTP</td></tr>
<tr><td>Endpoint</td><td>Method</td><td>Input</td><td>Output</td></tr>
<tr><td>/translate</td><td>POST</td><td>(JSON)
{
sentence: String
}</td><td>(JSON)
{
source: String,
gloss: String
}

source- Origin of gloss prediction.
(Default= database
Fallback 1= template
Fallback 2= model)
gloss- predicted ASL gloss</td></tr>
</table>

Speech-to-Text Service Contract			
What it does		• Converts english speech to text	
Protocol		HTTP	
Endpoint	Method	Input	Output

/api/upload-audio	POST	(multipart/form-data)
		
		Form with a .wav/.raw audio file appended
			(JSON)
			{
			“text”: String (empty if unrecognized)
			}

Sign-to-Text Service Contract			
What it does		• Converts a sign in the form of live keypoints (coordinates) into an ASL gloss (text). • A Bidirectional GRU model is used for predictions. This model is trained on multiple recordings. • A sliding window buffer is used to accumulate per frame keypoints for predictions. • It emits the top-k class probabilities continuously and displays them when reaching a certain probability threshold.	
Endpoint	Protocol/Method	Input	Output

/v1/session/start	HTTP: POST	None	(JSON)
			{
			session_id: String,
			model: String,
			expected_F: Integer,
			T: Integer,
			J: Integer,
			channels: String,
			labels: String[]
			}
			
			session_id - sessionID for websocket connection
			model - model to be used. “bigru” only currently supported model
			expected_F- total features

			per frame **T** - frames per prediction **J** - joints per frame **channels** - says what kind of joint features model expects (e.g. “xyz”) **labels** - list of words the model can recognize (from label_map.json)

/v1/session/stop	HTTP: POST	{
		session_id: string
		}
			{
			ok: true
			}

/v1/stream/(session_id}	Websocket	{
		type: string
		}
		
		type - “clear_sentence” in order to clear sentence and reset state, “undo” to remove last committed word
		
		{
		pose33: [],
		left21: [],
		right21: []
		}
		
		pose33 - array of arrays for each body landmark position
		left21 - array of arrays for each left hand landmark position
		right21 - array of arrays for each right hand landmark position
			{
			type: “prediction”,
			topk: [{}, {}, {}],
			stable: boolean,
			idle: boolean,
			filling: boolean
			}
			
			topk - array of top 3 predictions (a label which is the word and probability/confidence)
			stable - if the top-1 word is stable before committing
			idle - true if too few landmarks are presented
			filling - true while sliding window is getting ready
			
			{
			type: “word_event",
			label: string,
			confidence: number
			}
			
			label - the committed word
			confidence - level of confidence return from model
			
			{
			type: “sentence”,
			text: string
			}
			
			text - consecutive

			committed words combined into a sentence
			
			{
			type: “error”,
			msg: “invalid session”
			}
			
			Sent if the session_id doesn’t match or is unknown. Closes the socket afterwards

Auth-service Service Contract			
What it does		• Handles registration, login, deregistration and user preferences such as light mode or dark mode, speech speed etc.	
Protocol		HTTP	
Endpoint	Method	Input	Output

/register	POST	(JSON)
		{
		email: String,
		password: String
		}
			(JSON)
			Success:
			Code- 200
			{
			status: “success”,
			message: “signup successful”
			}
			
			Failure:
			Code- 500
			{
			message: “Error signing up user”
			error: “Some error message”
			}

/login	POST	(JSON)
		{
		email: String,
		password: String
		}
			(JSON)
			Success:
			Code- 200
			{
			status: “success”,
			message: “Login successful”,
			user: ?

			}
			
			Incorrect password:
			Code- 401
			{
			message: “Incorrect password”
			}
			
			Failure:
			Code- 500
			{
			message: “Error logging in”
			error: “Some error message”
			}

/deleteAccount/{userID}	DELE
	TE
		None	(JSON)
			Success:
			Code- 200
			{
			status: “success”,
			message: “User account deleted successfully”
			}
			
			User not found:
			Code- 404
			{
			message: “User not found or already deleted”
			}
			
			Failure:
			Code- 500
			{
			message: “Error deleting user”,
			error: “Some error message”
			}

/preferences/{userID}	GET	None	(JSON)
			Success:
			Code- 200
			Example:
			{
			status: "success",
			preferences: {
			displayMode: "Dark Mode",
			fontSize: "Medium",
			preferredAvatar: "Zac",
			animationSpeed: 1,
			speechSpeed: 1,

			speechVoice: "George"
			}
			}
			
			User not found:
			Code- 404
			{
			message: “User not found”
			}
			
			Failure:
			Code- 500
			{
			message: “Error fetching preferences”,
			error: “Some error message”
			}

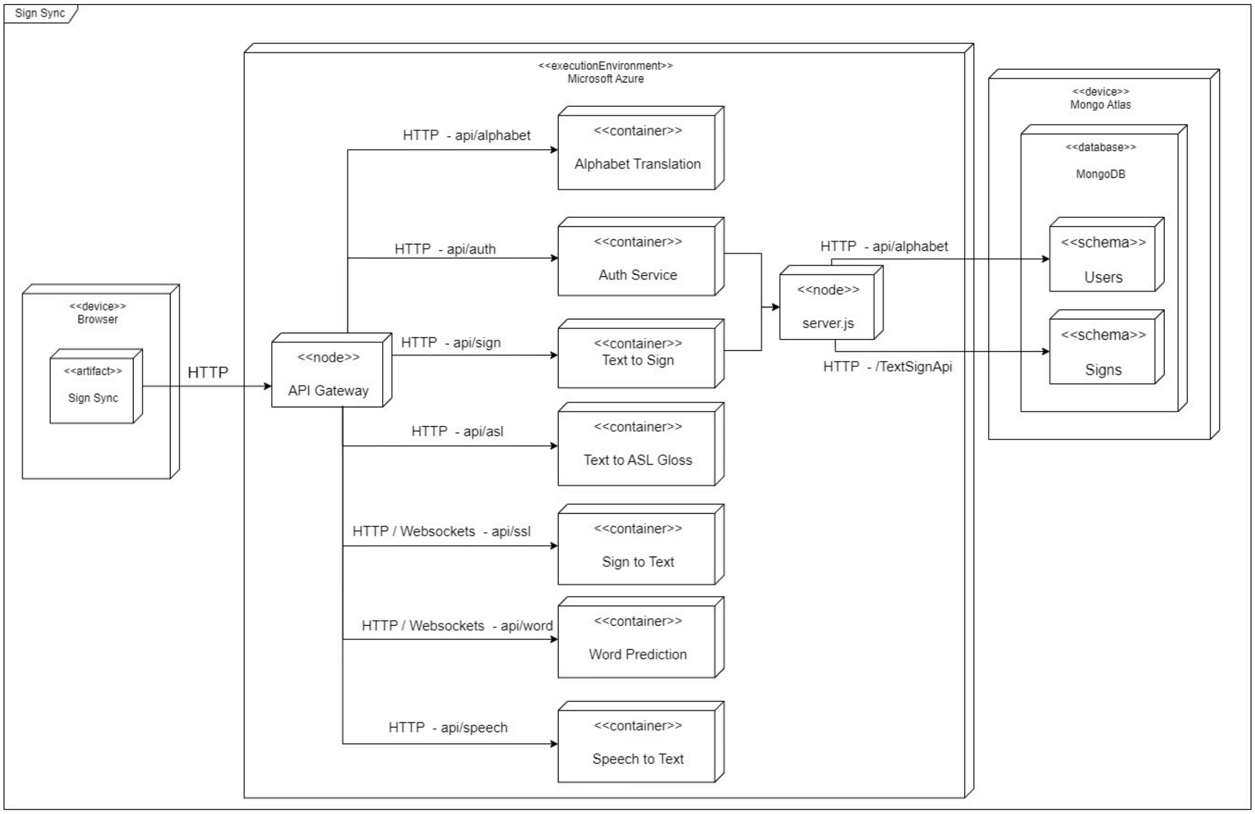
/preferences/{userID}	PUT	(JSON)
		Example:
		{
		status: "success",
		preferences: {
		displayMode: "Dark Mode",
		fontSize: "Medium",
		preferredAvatar: "Zac",
		animationSpeed: 1,
		speechSpeed: 1,
		speechVoice: "George"
		}
		}
			(JSON)
			Success:
			Code- 200
			{
			status: “success”,
			message: “Preferences updated”
			}
			
			User not found:
			Code- 404
			{
			message: “User not found”
			}
			
			Failure:
			{
			message: “Error updating preferences”,
			error: “Some error message”
			}

<table>
<tr><th colspan="4"><u>Alphabet-translate Service Contract</u></th></tr>
<tr><td colspan="2">What it does</td><td colspan="2">Translates sign language gestures for alphabet letters to text
Uses a multilayered neural network</td></tr>
<tr><td>Endpoint</td><td>Method</td><td>Input</td><td>Output</td></tr>
<tr><td>/predict</td><td>POST</td><td>(JSON)
{
keypoints: Float[21][3]
}

keypoints- a 2D array containing 21 subarrays or x,y,z coordinates extracted via Mediapipe</td><td>(JSON)
{
prediction: String
}

prediction- predicted letter of alphabet</td></tr>
</table>

Deployment Model



For better view:
Apollo Projects (Sign Sync) - Deployment Model Diagram

Sign sync is a web app that will be deployed via Microsoft Azure on to the internet. The system as a whole is composed of 3 different architectures, Component Based (Frontend), Microservices (Backend) and N-Tier for the full stack. For the issue of deployment, only N-tier and Microservices are relevant.

The services that comprise the backend are each individually dockerised in its own container and uploaded to Azure, where the API Gateway will then provide a singular point of access to the frontend web application.

The target environment is Cloud-Based, due to the deployment being hosted on Azure and the database is hosted by MongoDB Atlas.