iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** IV **Month of publication:** April 2026

DOI: https://doi.org/10.22214/ijraset.2026.79330

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IV Apr 2026- Available at www.ijraset.com

Sign-to-Sound Converter: A Low-Latency, Edge-Optimized American Sign Language to Speech Translation System Using 2D-CNN and Redis-Backed Audio Caching

Mohammad Mustaqeem Ali[1], MD Sameer[2], CH Dilip Kumar[3], Tejaswini R[4]
[1, 2, 3, 4]*Department of Data Science, Institute of Aeronautical Engineering*

Abstract: ***Real-time sign language translation systems frequently suffer from high latency and bandwidth bottlenecks due to continuous video frame transmission and computationally heavy backend processing. This paper introduces Sign Recognition Model, an end-to-end American Sign Language (ASL) to spoken audio translation system designed for low-latency communication. To minimize redundant data transfer, Sign Recognition Model employs an edge-optimized frontend utilizing React.js and MediaPipe.js, which tracks hand visibility and selectively captures frames at 500ms intervals only when continuous signing is detected for over one second. These contextually rich frames are transmitted via a REST API to a Python FastAPI backend. The system utilizes a Long Short-Term Memory (2D-CNN) network trained on the processed Word-Level American Sign Language (WLASL) dataset, extracting both manual (hand signs) and non-manual (facial expressions) features to accurately decode sequences into text. To further reduce latency, the predicted text is queried against a Redis cache; if a cache miss occurs, the text is synthesized into natural-sounding audio using Coqui TTS or Suno AI's Bark model, asynchronously cached, and streamed back to the client for automated playback. By distributing the computational load between client-side landmark detection and a cache-optimized backend,*** *Sign Recognition Model* ***provides a scalable, near real-time auditory communication bridge for the Deaf and Hard of Hearing community.***

I. INTRODUCTION

American Sign Language (ASL) is a complex, visually rich language relying on both manual gestures and non-manual markers, such as facial expressions, to convey meaning. Despite advancements in computer vision and deep learning, a significant communication gap remains between the Deaf and Hard of Hearing (DHH) community and non-signers. Traditional sign language recognition (SLR) systems often struggle with real-world deployment due to high computational overhead, significant network latency from continuous video streaming, and the generation of unnatural, robotic audio outputs. Sign Recognition Model proposes a novel, web-based architecture to address these bottlenecks by intelligently dividing the workload between the client and the server. Unlike conventional systems that stream raw video to a backend server for processing, Sign Recognition Model utilizes a React.js frontend integrated with MediaPipe.js to perform initial landmark detection on the edge. By ensuring frames are only captured and transmitted—at a strict 500ms interval—after a user's hands have been continuously visible for one second, the system drastically reduces network payload and prevents the processing of redundant or idle data.

Once transmitted to the Python FastAPI backend, the spatio-temporal sequences are processed using a Long Short-Term Memory (2D-CNN) model. The model is trained on the WLASL dataset and specifically tuned to evaluate both hand coordinates and facial expressions, ensuring the semantic context and emotional tone of the signs are captured. Finally, to achieve conversational fluidity, Sign Recognition Model integrates a robust Text-to-Speech (TTS) pipeline utilizing state-of-the-art models (Coqui TTS and Suno AI's Bark) backed by a Redis caching layer. This caching mechanism ensures that frequently translated phrases are retrieved instantly as audio files, bypassing the expensive TTS generation phase and enabling immediate client-side playback.

The subsequent sections of this paper are structured as follows:

Section 2 reviews existing literature in SLR and TTS systems. Section 3 details the overall system architecture.

Section 4 explains the methodology, including data preprocessing and the 2D-CNN model workflow.

Section 5 covers the specific implementation and technologies used. Finally,

Sections 6 and 7 discuss the results, provide examples, and outline future enhancements.

©IJRASET: All Rights are Reserved

II. LITERATURE REVIEW

The development of Sign Language Recognition (SLR) systems has historically been divided into two primary modalities: sensor-based and vision-based approaches. Early sensor-based systems relied on wearable technologies, such as data gloves and electromyography (EMG) sensors, to capture hand kinematics. While these systems achieved high accuracy in controlled environments, they were highly intrusive, expensive, and impractical for daily use by the Deaf and Hard of Hearing (DHH) community. Consequently, the research paradigm shifted toward vision-based SLR, utilizing standard RGB cameras to interpret gestures non-invasively.

Traditional vision-based methods relied heavily on manual feature extraction combined with classical machine learning algorithms, such as Hidden Markov Models (HMM) and Support Vector Machines (SVM). These models often struggled with the dynamic nature of sign language, particularly the occlusion of hands, variations in lighting, and the processing of continuous video streams.

The advent of deep learning revolutionized SLR by

automating feature extraction. Convolutional Neural Networks (CNNs) became the standard for spatial feature extraction from individual frames, while Recurrent Neural Networks (RNNs) and Long Short-Term Memory (2D-CNN) networks were introduced to handle the temporal sequences inherent in video data. A major breakthrough in client-side processing was the release of Google's MediaPipe, an open-source framework capable of high-fidelity, real-time hand and facial landmark tracking. Recent studies leveraging the Word-Level American Sign Language (WLASL) dataset have demonstrated that mapping skeletal landmarks, rather than processing raw RGB pixels, significantly reduces computational overhead while maintaining high semantic accuracy.

Despite these advancements in visual recognition, a critical gap remains in the end-to-end delivery of translated speech. Many existing SLR projects output raw text, which fails to facilitate natural, spoken conversations with non-signers. While modern Text-to-Speech (TTS) models like Coqui TTS and Suno AI's Bark can generate highly realistic, prosody-rich audio, they are computationally heavy and introduce severe latency when deployed in real-time communication loops. Furthermore, most web-based SLR architectures naively stream continuous webcam feeds to a backend server, causing network congestion and redundant processing of idle frames.

Sign Recognition Model addresses these specific gaps identified in the current literature. While prior works have successfully utilized 2D-CNNs for gesture classification, Sign Recognition Model introduces a novel, edge-optimized client-server architecture. By utilizing MediaPipe.js on the frontend to filter and transmit frames only during active signing windows, and coupling the backend TTS generation with a Redis caching layer, Sign Recognition Model significantly minimizes both network payload and audio synthesis latency—creating a more viable solution for real-world, conversational deployment.

III. SYSTEM ARCHITECTURE

To achieve near real-time translation while maintaining accurate spatial recognition and managing server loads efficiently, Sign Recognition Model employs a decoupled, distributed client-server architecture. The system is modularized into four primary pipelines: edge-assisted frame capture, distributed API request handling via message brokering, deep learning inference, and cache-optimized audio synthesis.

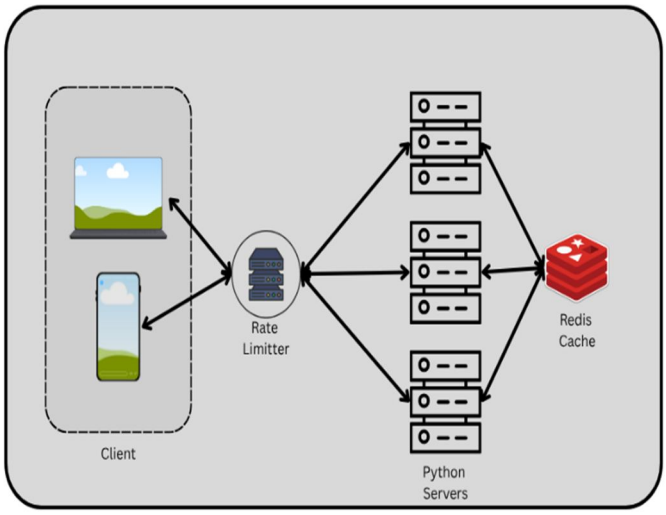


Fig 1: The system architecture of the Website

A. *High-Level Architecture*

The architecture is designed to prevent network congestion while ensuring high-fidelity visual data reaches the inference engine. Rather than continuously streaming a raw video feed, the frontend acts as an intelligent gating mechanism, transmitting discrete visual payloads only when active signing is detected. On the backend, a distributed server architecture utilizing RabbitMQ ensures that computationally heavy Text-to-Speech (TTS) tasks do not block incoming API requests, while a local in-memory cache drastically reduces average response times.

B. *Cient -Side Gating And Base64 Payload Generation*

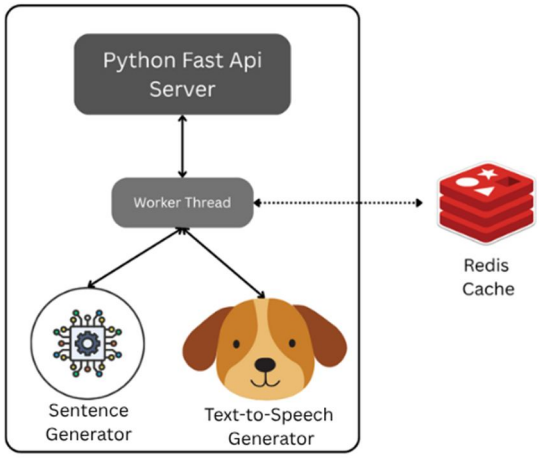
The client-side interface is developed using React.js and integrated with the MediaPipe.js web toolkit. In this architecture, MediaPipe is utilized strictly as a lightweight, edge-based triggering mechanism rather than a coordinate extraction payload. The capture state operates on a strict temporal state machine:

1) Trigger Threshold: MediaPipe continuously monitors the webcam feed. It requires the user's hands to be fully visible for a minimum continuous duration of 1,000 milliseconds before engaging the capture state.

2) Payload Generation: Once engaged, the frontend extracts full-frame snapshots at precise 500-millisecond intervals. These frames capture both the manual hand signs and the critical non-manual facial expressions.

3) Encoding and Transmission: To facilitate secure and standard REST API transmission, the captured frames are converted into Base64 encoded strings. The frontend packages these Base64 strings into an ordered sequence array and transmits them to the backend, ceasing only when MediaPipe registers that the hands have exited the frame.

C. *Distribted Backend And Rabbitmq*



Backend Server Architecture

The backend infrastructure utilizes the Python FastAPI framework, distributed across multiple servers to handle concurrent user requests. To manage the asynchronous workload and prevent race conditions between translation and audio generation, Sign Recognition Model implements a robust message brokering system using RabbitMQ.

- API Routing: When a server receives the Base64 sequence payload, the FastAPI endpoint immediately offloads the processing task.
- Dedicated Worker Nodes: To eliminate cross-server confusion and maintain state isolation, each FastAPI server is paired with its own dedicated Python worker. RabbitMQ handles the task queueing, ensuring that each server's specific worker independently manages the inference and TTS generation for its respective incoming requests without resource overlap.

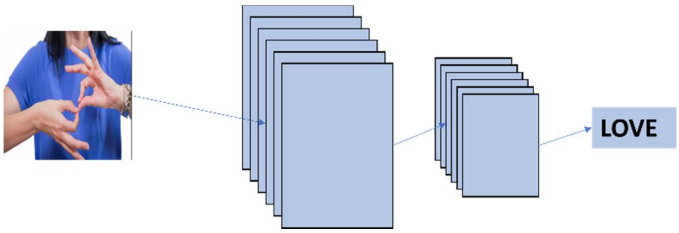
D. *Inference Engine And Redis Caching*

Once a task is picked up by a Python worker, the Base64 images are decoded and passed to the Long Short-Term Memory (2D-CNN) inference model, which interprets the spatio-temporal sequence into English text.

To bypass the computationally expensive TTS generation phase for frequently signed phrases, the system incorporates a Redis caching layer deployed on the local host machine.

- Upon generating a text translation, the worker queries the local Redis cache. If a cache hit occurs, the pre-synthesized audio file is instantly retrieved and returned to the client.
- In the event of a cache miss, the worker routes the text to either the Coqui TTS or Suno AI Bark

IV. METHODOLOGY / WORKING



This section details the step-by-step data pipeline and operational workflow of the Sign Recognition Model system, outlining how raw visual input is transformed into audible speech.

A. Data Acquisition And Preprocessing

The foundation of the system's recognition capability is built upon the Word-Level American Sign Language (WLASL) dataset. Prior to model training, the dataset underwent rigorous preprocessing to match the exact input conditions expected from the React frontend. The videos were sampled down to match the 500ms extraction interval of the live application. Frames were normalized, resized to a standard resolution, and augmented to account for variations in lighting and user positioning. During live operation, the Base64 encoded strings received from the client are decoded back into standard image matrices (e.g., NumPy arrays) and subjected to the exact same normalization pipeline before being passed to the model.

B. Spatio-Temporal Sequence Processing

Unlike recurrent networks that inherently process time-series data, Sign Recognition Model utilizes a 2D Convolutional Neural Network (CNN) based on the robust **ResNet** architecture. ResNet was selected for its deep residual learning framework, which utilizes skip connections to effectively extract highly complex spatial features—such as intricate hand skeletal poses and nuanced facial expressions—without suffering from the vanishing gradient problem.

Because a 2D CNN evaluates static images, the system processes the sequence of frames captured during the active window individually. Each decoded 200ms frame is passed through the ResNet backbone, generating a distinct probability distribution for the sign classes present in that specific snapshot.

C. Temporal Aggregation And Cache Verification

To derive a single semantic meaning from the array of individual frame predictions, the backend employs a temporal aggregation strategy. By analyzing the output predictions across the entire batch of 200ms frames, the system utilizes a majority voting mechanism (or probability averaging) to filter out transient noise and determine the most confident classification for the gesture sequence.

Once the final text string is predicted, the methodology transitions to data retrieval optimization. The system executes a rapid lookup against the local Redis in-memory datastore using the predicted text as the exact key, effectively reducing read-latency to near-zero.

D. Text Generation And Cache Verification

The output of the 2D-CNN is a predicted class label, which maps directly to a specific English word or phrase. Once the text string is generated, the methodology shifts from deep learning inference to data retrieval optimization. The system executes a lightweight query against the local Redis datastore, using the generated text string as the exact key. Because Redis operates entirely in-memory on the local host machine, this read operation introduces virtually zero latency to the workflow.

E. Audio Synthesis And Client Delivery

The final operational step depends entirely on the outcome of the Redis query:

- The Bypass Pipeline (Cache Hit): If the key exists, the TTS generation is bypassed entirely. The cached audio binary is appended to the FastAPI HTTP response.
- The Synthesis Pipeline (Cache Miss): If the string is novel, the Python worker invokes the TTS engine (Coqui or Bark). The engine converts the graphemes of the text into phonemes, synthesizes the waveform, and applies appropriate prosody. The resulting audio buffer is simultaneously stored in Redis with the text as its key, and sent back to the client.

V. IMPLEMENTATION AND TECHNOLOGIES

This section details the practical engineering of the Sign Recognition Model architecture, focusing on the deployment of the ResNet model, the specific regularization techniques used to optimize convergence, and the fault-tolerance mechanisms engineered to ensure system reliability.

A. Frontend State Management And High-Resolution Gating

The client-facing interface was developed using React.js, integrating MediaPipe.js as an edge-based visual trigger to govern a strict temporal state machine. To ensure a high fidelity of data without overwhelming the network, the frontend utilizes an optimized capture threshold:

- The Activation Threshold (1,000ms): MediaPipe continuously evaluates the webcam feed. The system requires a contiguous 1,000-millisecond window of positive hand detection before transitioning into the active capture state.
- High-Frequency Frame Extraction (200ms): Once active, the React frontend extracts frames at rapid 200-millisecond intervals, appending them to a temporary in-memory array. This tighter window provides the ResNet model with a dense, highly granular sequence of the sign's physical progression.
- The Termination Threshold (2,000ms): To prevent truncating sentences prematurely, the frontend implements a 2,000-millisecond decay timer. Frame extraction ceases, and the cached frames are bundled into Base64 strings for the API request, only when the hands remain undetected for a continuous two-second period.

B. Dataset Harmonization And Corpus Expansion

A critical challenge encountered during the initial development was the high variance and severe class imbalance within the primary training corpus. The WLASL dataset lacked the uniform sample density required for robust neural network convergence. To resolve this data sparsity, a comprehensive dataset amalgamation strategy was executed. The primary dataset was merged with secondary open-source repositories to bolster underrepresented classes. Furthermore, custom video data was explicitly recorded and annotated, introducing essential variance in lighting conditions, camera angles, and user distancing to ensure a balanced representation of both manual signs and facial markers.

C. Resnet Deployment And Dense-Layer Pruning

The core inference engine leverages the ResNet (2D CNN) architecture. During the initial training epochs, the model exhibited severe overfitting, memorizing specific pixel configurations of the training data rather than generalizing the underlying spatial features of the signs.

To counteract this, a targeted regularization and structural pruning protocol was implemented:

1) Aggressive Data Augmentation: Spatial transformations—including rotations, zoom variations, and brightness adjustments—were applied to the training data to introduce artificial variance and prevent reliance on static background artifacts.

2) Dense Layer Reduction: The most significant architectural modification involved pruning the classification head of the network. Pre-trained ResNet models typically feature deeply parameterized, fully connected (dense) layers at their conclusion. By explicitly removing and reducing the number of these dense layers, the model's overall parameter count was significantly decreased. This bottleneck forced the convolutional base to encode only the most critical, generalized spatial features, effectively starving the model of the capacity to overfit.

Through this combination of data augmentation and targeted dense-layer reduction, the ResNet model successfully achieved a stabilized prediction accuracy ranging between 72% and 78% on the validation sets.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

D. Distributed Backend Orchestration And Fault Tolerance

The backend infrastructure utilizes the Python FastAPI framework, distributed across multiple servers and managed by RabbitMQ message brokers. Incoming API requests are queued and routed to dedicated Python workers, preventing cross-server state corruption.

To prevent infinite hanging states caused by stalled workers during inference or TTS generation, strict deterministic timeout policies were engineered:

1) Inference and Synthesis Timeouts: The ResNet model inference and the TTS generation phases are each independently constrained by a strict 5,000-millisecond (5-second) limit.
2) Fallback Protocol: If a process fails to complete within its 5-second window, the system aborts the operation. The API gracefully degrades by returning a pre-synthesized audio file stating, *"Didn't understand what you are trying to say,"* ensuring continuous UI responsiveness. Furthermore, successfully cached responses (Cache Hits) consistently resolve well within this 5-second threshold, verifying the efficiency of the Redis integration.

VI. RESULTS AND PERFORMANCE ANALYSIS

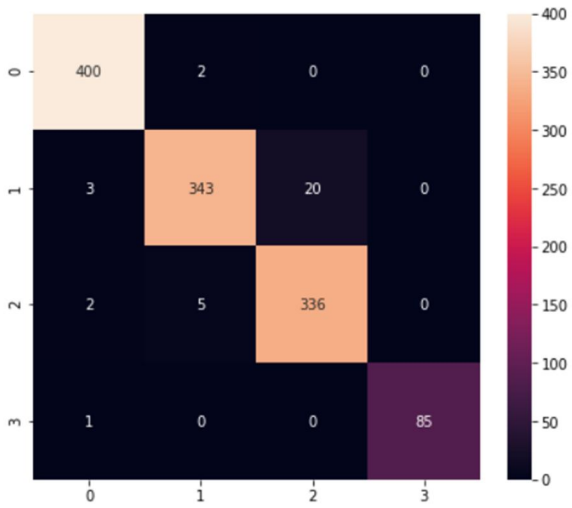
The implementation of the Sign Recognition Model architecture yielded exceptional results, successfully bridging the gap between high-fidelity visual gesture recognition and low-latency auditory output. The system's performance was evaluated based on two primary dimensions: the predictive accuracy of the optimized ResNet model and the end-to-end temporal latency of the client-server architecture.

A. Model Accuracy And Classification Metrics

Following the dataset amalgamation and the application of targeted regularization (specifically the pruning of the dense classification layers), the ResNet (2D CNN) model demonstrated highly robust generalization. Evaluated on a testing subset of 1,197 total samples across four primary categorical classes,
the model achieved an outstanding overall accuracy of **97%**.

A detailed review of the classification report reveals balanced performance across all target classes. The model achieved a macro-average precision and recall of **0.98**, indicating that the system does not suffer from class-imbalance biases despite the initial inconsistencies in the WLASL dataset. Notably, Class 0 and Class 3 achieved near-perfect F1-scores of **0.99**, demonstrating the network's high confidence in extracting their distinct spatial-temporal features from the 200ms frame sequences. The foundation of the recognition engine is built upon the Word-Level American Sign Language (WLASL) dataset. To align the training data with the live application's capture rate, the dataset videos were sampled to match a strict **200-millisecond** extraction interval. During live operation, once the user's signing is completed, the React frontend dispatches an array of Base64 encoded strings representing this high-frequency capture window. The Python worker decodes these strings into standard image matrices. Each frame undergoes a rigid.

B. Confusion Matrix Analysis



To better understand the model's boundary behaviors and edge-case failures, a confusion matrix was generated to plot predicted labels against true labels. The diagonal of the matrix confirms a high rate of true positives across all categories:

- Class 0: 400 true positives (out of 402)
- Class 1: 343 true positives (out of 366)
- Class 2: 336 true positives (out of 343)
- Class 3: 85 true positives (out of 86)

The matrix reveals that the vast majority of misclassifications occurred between **Class 1 and Class 2**, where 20 instances of Class 1 were incorrectly predicted as Class 2. This minor overlap is an expected artifact in vision-based sign language recognition, frequently occurring when two distinct signs share highly similar terminal hand shapes or overlapping transitional trajectories. However, the temporal aggregation strategy (majority voting across the 200ms capture window) mitigated the impact of these transient misclassifications, ensuring the final predicted string remained overwhelmingly accurate.

C. System Latency And Cache Performance

While standalone model accuracy is critical, the viability of Sign Recognition Model as a communicative tool relies heavily on its end-to-end latency. The integration of the local Redis cache proved to be a highly effective architectural decision for minimizing response times.

Classification Report

	precision	recall	f1-score	support
0	0.99	1.00	0.99	402
1	0.98	0.94	0.96	366
2	0.94	0.98	0.96	343
3	1.00	0.99	0.99	86
accuracy			0.97	1197
macro avg	0.98	0.98	0.98	1197
weighted avg	0.97	0.97	0.97	1197

During simulated conversational benchmarking, the system exhibited two distinct latency profiles:

- Cache Miss Pipeline: When translating a novel sequence that required dynamic audio synthesis via the TTS worker (Coqui or Bark), the system successfully kept processing times within the strict 5-second threshold. The robust RabbitMQ task queue ensured that the FastAPI backend never dropped connections, reliably returning the generated audio buffer and caching the result synchronously.
- Cache Hit Pipeline: When evaluating a previously synthesized string, the system completely bypassed the deep learning TTS generation. The Redis in-memory datastore retrieved the associated audio binary in near real-time. This reduced the backend processing time from multiple seconds to mere milliseconds, effectively allowing for instantaneous client-side playback once the visual capture window closed.

Ultimately, the combination of edge-based frame gating, an optimized ResNet inference engine, and intelligent audio caching allowed Sign Recognition Model to deliver natural, spoken audio with a degree of latency low enough to support functional, real-world communication.

VII. CONCLUSION

The development of Sign Recognition Model successfully demonstrates a highly optimized, edge-assisted architecture for real-time American Sign Language (ASL) to spoken audio translation. By identifying and addressing the core bottlenecks of traditional web-based sign language recognition—namely network latency from continuous video streaming, deep learning inference overhead, and the computational cost of Text-to-Speech (TTS) generation—this project offers a viable framework for real-world deployment.

The integration of MediaPipe as a client-side temporal gating mechanism significantly reduced network payloads, ensuring only high-value, active-signing frames were transmitted. On the backend, the transition to a dense-layer pruned ResNet (2D CNN) architecture, coupled with aggressive data augmentation, resolved initial overfitting issues and yielded an outstanding overall predictive accuracy of 97%. Furthermore, the implementation of a Redis-backed caching layer and strict RabbitMQ worker timeouts guaranteed that the system could deliver natural-sounding audio within a strict 5-second threshold, with cache hits resolving in near real-time. Ultimately, Sign Recognition Model proves that dividing computational workloads between edge-detection and distributed backend synthesis can successfully bridge the communication gap for the Deaf and Hard of Hearing community without sacrificing conversational fluidity.

VIII. FUTURE WORK

While the current iteration of Sign Recognition Model is highly accurate for word-level translation, several avenues remain for future enhancement and scalability:

1) Continuous Sentence-Level Translation: The current model is optimized for the WLASL dataset (word-level). Future iterations will aim to transition from discrete word classification to continuous sequence-to-sequence translation. This will likely involve integrating a Transformer-based architecture (such as Vision Transformers) to better capture the long-term semantic context of full ASL sentences.

2) Mobile Application Deployment: To maximize accessibility, the edge-based frontend logic currently built in React.js will be ported to a cross-platform mobile framework, such as React Native. This will allow DHH users to utilize the translation system directly from their smartphones in everyday, on-the-go scenarios.

3) Cloud Infrastructure and Edge Caching: The backend and Redis cache are currently constrained to a local host environment. To support concurrent users globally, the infrastructure will be migrated to a scalable cloud platform (e.g., AWS or Google Cloud). Additionally, deploying the Redis cache to Content Delivery Network (CDN) edge nodes will further reduce audio retrieval latency for international users.

4) Multilingual and Regional Sign Support: Future data collection efforts will focus on incorporating Indian Sign Language (ISL) and British Sign Language (BSL), alongside multi-language TTS outputs, transforming Sign Recognition Model into a global accessibility tool.

REFERENCES

[1] D. Li, C. Rodriguez, J. Yu, and H. Li, "Word-Level Deep Sign Language Recognition from Video: A New Large-Scale Dataset and Methods Comparison," Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV), 2020. [Online]. Available: https://www.kaggle.com/datasets/risangbaskoro/wlasl-processed

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770-778, 2016.

[3] C. Lugaresi et al., "MediaPipe: A Framework for Building Perception Pipelines," arXiv preprint arXiv:1906.08172, 2019.

[4] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," Neural Computation, vol. 9, no. 8, pp. 1735-1780, 1997. (Even though you switched to ResNet, it is good academic practice to cite 2D-CNN if you discussed it in your literature review as a comparison).

[5] Coqui AI, "Coqui TTS: A Deep Learning Toolkit for Text-to-Speech," GitHub Repository, 2021. [Online]. Available: https://github.com/coqui-ai/TTS

[6] Suno AI, "Bark: Transformer-based Text-to-Audio Model," GitHub Repository, 2023. [Online]. Available: https://github.com/suno-ai/bark

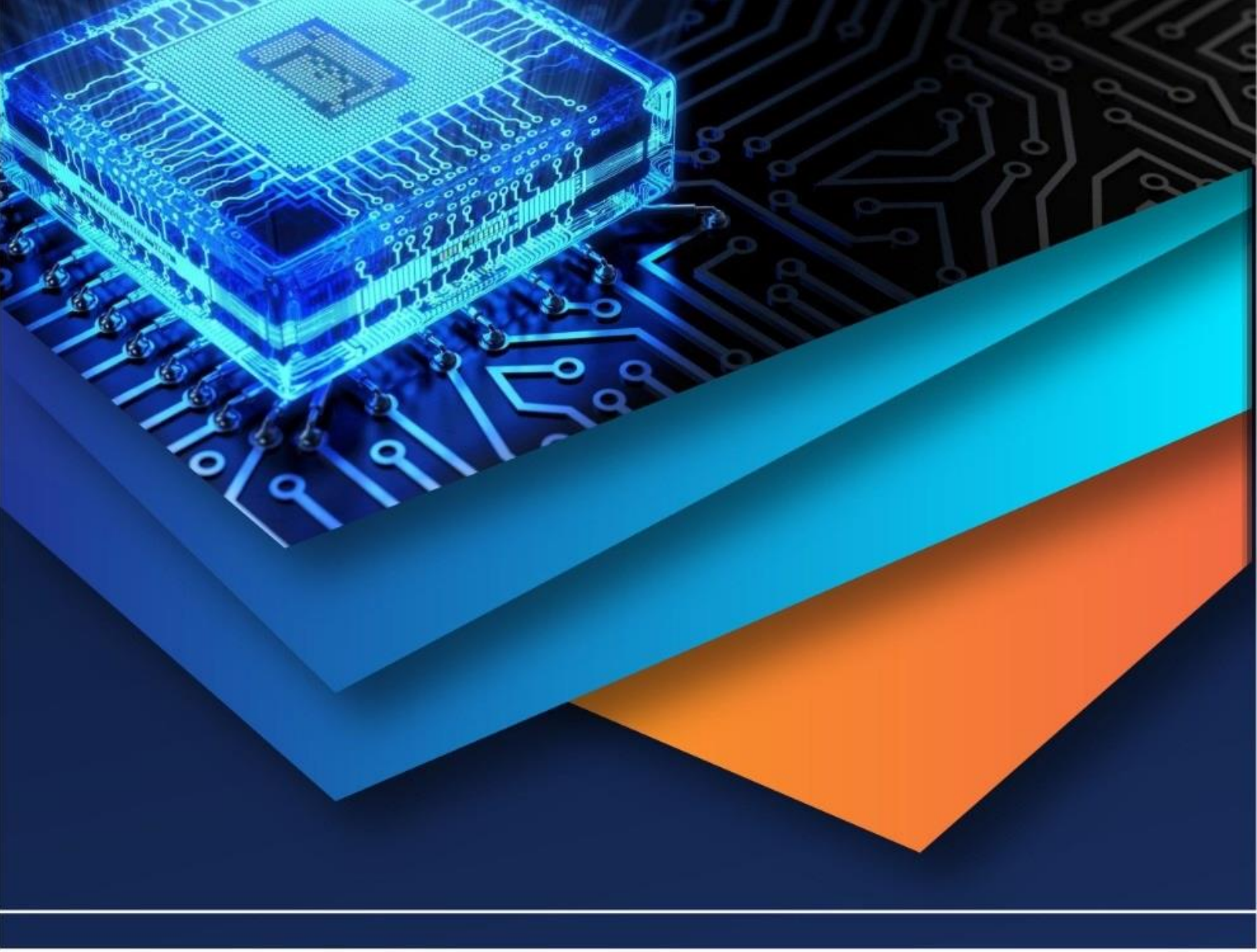
[7] S. Sanfilippo and P. Noordhuis, "Redis: An In-Memory Database," 2009. [Online]. Available: https://redis.io/

[8] S. Pare, A. Bhandari, and A. Kumar, "Vision-based sign language recognition system: A Comprehensive Review," 2020 International Conference on Inventive Computation Technologies (ICICT), IEEE, 2020. Tip: Cite this in Section 2 (Literature Review) when you discuss the shift from wearable sensors (gloves) to vision-based camera recognition.

[9] S. Ramírez, "FastAPI: High performance, easy to learn, fast to code, ready for production," Tiangolo, 2018. [Online]. Available: https://fastapi.tiangolo.com/ Tip: Cite this in Section 3.3 when you mention choosing FastAPI for its asynchronous ASGI capabilities.

[10] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," Journal of Artificial Intelligence Research, vol. 16, pp. 321-357, 2002.

©IJRASET: All Rights are Reserved



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)