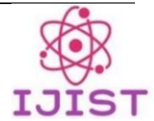
OPEN ACCESS
International Journal of Innovations in Science & Technology



A Hybrid WebSocket–HTTP Architecture for Low-Latency Cloud AI Integration in Assistive Communication Systems

Abuzar Soomro, Aqsa Younus, Bhave Sagar, Kehkashan Asma Memon, Saba Baloch, Bilal Khan

Mehran University of Engineering and Technology.

*Correspondence: abuzarsoomro3@gmail.com

Citation | Soomro. A, Younus. A, Sagar. B, Memon. K. A, Baloch. S, Khan. B, "A Hybrid WebSocket–HTTP Architecture for Low-Latency Cloud AI Integration in Assistive Communication Systems", IJIST, Vol. 8 Issue. 3 pp 1192-1206, June 2026

Received | April 19, 2026 **Revised |** May 22, 2026 **Accepted |** May 29, 2026 **Published |** June 11, 2026.

Cloud-based Artificial Intelligence (AI) services offer speech recognition, synthesis, and image text extraction services for use in assistive communication technology. However, there are acute challenges in integrating these services in resource-constrained embedded systems such as high latencies that interfere with real-time communication, complex authentication protocols, communication protocol overheads, and frequent API changes that jeopardize system maintainability. In this paper, a hybrid WebSocket-HTTPS architecture is introduced which is designed to match communication protocols with cloud AI service patterns, evaluated on an ESP32-S3 microcontroller, where streaming and transactional cloud AI interactions are separated, with a WebSocket bridge server handling provider-specific authentication and protocol management for the streaming service, and the transactional service accessed directly via the HTTPS protocol. The WebSocket bridge server abstracts the complexities of cloud communication protocols and allows the removal of authentication from embedded firmware and shields against API modifications. With the proposed architecture, the speech-to-text latency over WebSocket for Deepgram Nova-2 averaged 976 ms, while Azure Cognitive Services averaged 1768 ms, both of which are far below the conversational disruption threshold of 2000 ms. In contrast, HTTPS speech-to-text resulted in a mean latency of 2574 ms, about 1.5 times longer, mainly due to the TLS handshake overhead per connection (around 574 ms) and the need to upload full audio data to the cloud before processing could begin. Meanwhile, transactional services like Text-to-Speech (TTS) and Optical Character Recognition (OCR) achieved mean latencies of 1045 ms and 1888 ms respectively, when accessed directly over HTTPS, thus showing that this protocol can be used for stateless request-response interactions without WebSocket relay. This work provides a practical architectural methodology balancing real-time performance and implementation simplicity, particularly for resource-limited assistive devices, that can enable more sophisticated cloud-based AI functions.

Keywords: Assistive Technology; Cloud AI Integration; Edge-Cloud System; WebSocket Communication; Embedded System

Introduction:

Cloud-based Artificial Intelligence (AI) has been demonstrated to be the foundation of modern assistive communication systems. Cloud-hosted services can provide capabilities far beyond the means of low-power embedded hardware in the case of users with hearing, speech, or visual impairments [1]. Recent advances in AI-enabled assistive technologies, such as augmentative and alternative communication (AAC) devices, gesture recognition, and intelligent speech interfaces, **have** also illustrated the power of cloud AI in conjunction with specially designed embedded hardware [2]. However, combining cloud AI with assistive devices is not an easy task because of the sensitivity to latency, resource usage, and complexity in protocols, and the combination of these issues is what the current methods have been unable to handle simultaneously.

A study of the dynamics of human conversation has revealed that a response delay exceeding 2 seconds significantly interrupts the conversation [3]. Modern psycholinguistic studies have also demonstrated that the mutual interruption of speech in a conversation is strictly dependent on the natural turn taking, which is in the order of 100-300ms; even the smallest of misalignments in such a cloud-based transcription service is noticeable and problematic for the conversation [4]. These delays make real-time interaction a type of passive communication to deaf and hard-of-hearing individuals, who depend on speech-to-text transcription. Such timing factors suggest that end-to-end latency architectural decisions directly affect whether the cloud-based AI-assistive systems will fulfill natural communication or create accessibility barriers.

Embedded assistant devices work within tight limits, such as a small amount of memory and reduced networking stacks [5]. Although the functionality of the modern cloud AI SDKs is rich, they have large memory footprints on microcontroller-based systems, and frequently do not have much room left to execute application code. A recent body of research examining large language model deployment on edge and IoT hardware shows a wide variance of memory, latency, and energy consumption across device tiers, and that workload-aware evaluation of protocols is critical in the case of distributed computation across the device–edge–cloud continuum [6]. The constraints encourage the thin-client design which aims at simplifying the complexity of devices, yet the majority of the current integration models are server-grade hardware-based and do not consider these constraints.

Recent edge-cloud studies indicate that achieving integration success requires, protocols and application requirements must be aligned selectively instead of using a uniform approach to communication [7][8]. The current strategies, however, tend to implement the same protocol strategies in all types of cloud services, with either the result of unnecessary complexity in simple transactional operations or poor performance of latency-sensitive streaming jobs. Empirical studies conducted on microcontrollers akin to the ESP32 with WebSocket and other solutions have shown that WebSocket is significantly more efficient than polling-based solutions for streaming workloads, with lower round-trip time, while conventional HTTPS is more suitable for short transactions [9]. The disjunction, however, is not in the availability of competent cloud services but in the lack of an architecture that integrates protocol awareness, simplified firmware, and real-time performance on small embedded hardware.

This paper will deal with these restrictions by proposing a hybrid architecture which isolates continuous and transactional cloud interactions, uses a WebSocket bridge server to mediate streaming services, hiding provider complexities, and permits embedded devices to interact with transactional services using lightweight HTTPS calls.

The specific goals of this work are: (i) to measure the end-to-end latency of streaming transcription using an ESP32 microcontroller via a dedicated WebSocket bridge server, and conclude that it is feasible to achieve sub-2000 ms mean partial latency for streaming

transcription through the ESP32; (ii) to minimize the complexity of the embedded firmware by providing a (middleware) layer for handling provider-specific authentication and protocol functions, leaving only the embedded client to handle the complexities of doing lightweight HTTPS and WebSocket calls; (iii) to validate the selection of the protocol (WebSocket streaming vs HTTPS polling) empirically by streaming transcription across an ESP32, and concluding that it is feasible to use WebSocket streaming over an ESP32 for latency-sensitive tasks; and (iv) to demonstrate that a single embedded firmware design can interoperate with multiple cloud AI providers without vendor-specific SDK dependencies on the ESP32.

Novelty Statement:

The novelty of this research is the implementation of a hybrid WebSocket–HTTPS architecture for the ESP32 microcontroller to create real-time cloud-based integration with AI for assistive communication, while the embedded hardware conditions are tightly constrained. Compared to previous studies which required server class machines or vendor SDKs with high memory usage or introduced a protocol strategy without a clear embedded validation, this study demonstrates that streaming transcription can be done within mean partial latencies of 976 ms for Deepgram Nova-2 and 1768 ms for Azure Cognitive Services, both of which are under the 2000 ms conversational perceptibility threshold, and other tasks including TTS and OCR were performed efficiently using the standard HTTPS protocol. A WebSocket bridge server is a new abstraction that eliminates provider-specific authentication and protocol details from the embedded code, thereby simplifying the embedded code and enhancing cross-provider maintainability.

Related Work:

[7] considers edge-cloud integration patterns in a set of IoT deployments and demonstrate that homogeneous protocol adoption leads to sub-optimal performance in comparison to heterogeneous approaches that **tailor** protocols to specific interaction patterns. This observation suggests utilizing a different architecture for streaming and transactional interactions, rather than using single network protocol for both. In this context, [10] outline the benefits of utilizing WebSocket protocol for real-time communication between clients and servers, especially in cases where there is a high frequency of data exchange between client and server, which is directly applicable to our streaming speech-to-text pipeline. Combined, these works prove that there is a fundamental difference between streaming and transactional interaction, and the fact that one protocol can be used to process them creates unnecessary overhead, a concept that directly inspires the hybrid design proposed in this paper.

At the architectural level, [11] concludes that hybrid cloud systems increasingly use intelligent communication strategies to meet the latency-sensitive requirements of applications, which is another motivation for the need of a hybrid approach. [12] build on this argument by showing that latency sensitive IoT applications tend to produce continuous streams of data and that software layer communication is a major cause of end-to-end delay and therefore efficient delivery of real time data is required to have a responsive system. [13] further puts these technical considerations into context as in table 2 by setting human perception limits for response latency, noting that response times greater than 300 ms are increasingly perceptible and greater than 1 second are liable to cause mental context switching. These results are united by the one consistent message that architectural and protocol choices have a direct effect on whether a system is responsive or disruptive to its users, which is especially relevant to assistive communication where the flow of conversation is as much a factor as accuracy.

From an edge computing perspective, protocol selection is also important for devices that are resource constrained. [14] surveys communication technologies and data reduction techniques across the IoT-edge-cloud continuum, concluding that communication overhead due to high data traffic is a major bottleneck, and that processing and protocol strategies at

the edge can reduce bandwidth usage and response times. [15] discuss communication protocols in more detail for constrained and edge-based IoT systems, and demonstrates that the efficiency and latency of communication protocols depend highly upon interaction patterns and data characteristics. Together, these studies support the principle that protocol selection should be specific to application needs and not applied uniformly, lending further justification to the differentiated approach taken in this work.

Table 1. Time And User Perception Thresholds [13]

Delay	User Perception
0–100 ms	Instant
100–300 ms	Small perceptible delay
300–1000 ms	Machine is working
1,000+ ms	Likely mental context switch
10,000+ ms	Task is abandoned

A frequent design of cloud-integrated embedded systems is middleware abstraction. [16] argues that middleware can provide the flexibility in IoT and Cloud of Things scenarios as it addresses conflicts between the application and the devices' communication which is a successful method of transferring complexity out of the application layer. In addition, [17] thoroughly examine IoT middleware solutions and state the necessity of offering abstraction layers and high-level programming interfaces to shield the developer against heterogeneous communication technologies and low-level system settings. Equally, [18] lists resource optimization as a design requirement in the IoT middleware, dedicated to the effective management of computation, memory, storage, and energy in resource-constrained systems, like microcontrollers. All these works put in place a clear justification of the WebSocket bridge server proposed in this paper which exactly fulfills this middleware purpose of soaking up provider-specific complexity in order to have lean and stable embedded firmware.

In assistive communication, speech recognition has changed significantly due to the incorporation of the cloud, but usability is no longer dictated by the raw algorithmic accuracy, but more by the timing, presentation, and quality of execution. [19] have discovered that conventional accuracy measures like word error rate are weak predictors of real understanding in live TV captioning whereas timing and presentation are the predictive variables. Equally, [20] indicated that delays in captioning of approximately two seconds or more were found to have a considerable negative effect on understanding and involvement in an educational or work environment. When combined, these studies show that latency optimization cannot be considered as an after-thought in engineering, but as a key component of accessibility.

Hybrid edge-cloud systems have been successfully implemented at the system level in latency-sensitive workloads in speech and assistive communication. [21] presented an edge-centric speech-to-text system which used WebSocket streaming and offline ASR and achieved sub-second latency, a good example of hybrid design improving end-to-end responsiveness. [22] have shown that hybrid communication strategies between protocol selection and workload characteristics can enhance system performance, facilitating streaming of data over persistent connections and request-response operations to support control functions. Comparative studies show that hybrid deployments perform better than cloud or edge systems in latency-sensitive applications [23] while systematic reviews of WebRTC technologies confirm the benefits and challenges of low-latency, persistent, bidirectional communication outside the scope of conventional audio-video streaming [24]. The combination of these results indicates that, contrary to the complexity of the AI system, practical system design can define the usability and accessibility of real-time assistive communication systems.

The review studies discussed above each focus on independent aspects of a cloud-integrated communication system, but none of them has been able to converge on the unique

combination of real-time streaming performance, embedded hardware limitations, and middleware abstraction that defines the system described in the current work. In order to give these distinctions a tangible form, Table 2 places the proposed architecture, and the closest related previous works, in three dimensions: the communication protocol used, target hardware platform, and the reported latency.

Table 2. Comparison of Proposed Architecture with Related Works

Study	Protocol Used	Platform/Hardware	Latency Achieved
[7]	Heterogeneous (Protocol-specific)	IoT Devices	Not Measured
[12]	Various IoT Protocols	IoT Devices	Review-based, not measurement
[21]	WebSocket	Edge Server	Sub-second
[22]	Hybrid (Persistent + Request-Response)	Cloud Server	Not Measured
Proposed Work	Hybrid WebSocket + HTTPS	ESP32 Microcontroller	976–1768 ms (streaming), 1.0–1.9s (TTS/OCR)

This comparison raises a number of observations. To begin with, although [21] can achieve sub-second latency with audio streaming via WebSockets, their system uses locally run speech recognition offline, a tradeoff between cloud scalability and network reliance. The suggested design offers similar latency to streaming and the accuracy and versatility of commercial cloud AI services, which indicates that cloud reliance does not have to be compromised at the expense of real-time performance when protocol choice is managed with caution. Second, both [22][7] support the idea of the hybrid and heterogeneous communication strategies respectively and in their turn present the theoretical basis of the approach used in this study, but neither of them substantiates the choices by the latency measurements of the physical embedded hardware. The current work fills this gap directly by applying and testing a hybrid protocol system on ESP32 microcontroller in conditions of a real operation.

Combined, the comparison reveals that although this and previous works concentrate on these challenges individually, none combines them into a single implemented and validated system, which is exactly the gap that this work aims to address.

Methodology:

System Architecture:

The proposed architecture includes three key parts which are embedded client devices (ESP32 microcontrollers), WebSocket bridge server, and cloud service endpoints such as Azure Cognitive Services and Deepgram. These elements are used to isolate device-level communication, protocol processing, and cloud processing.

The ESP32 devices are lightweight clients, which are employed to receive input data and send it to cloud services. Direct integration with various cloud APIs would add complexity to firmware since embedded platforms are limited in their resources. Hence, the communication tasks are separated on the layers to make the implementation of the devices easier.

The service types of cloud AI in the system adhere to two interaction patterns. Streaming applications, e.g., real-time speech-to-text (STT), need continuous two-way connections in which audio is sent continuously and the results of the transcription are returned at the same time. Conversely, transactional services, like OCR and single-utterance TTS, are implemented in a stateless request response model, in which a full input is provided, and one output is returned. Such separation, as can be seen in Figure 1, guarantees that the

device layer is used to process acquisition, the communication layer is used to process protocol-specific interaction, and the cloud layer is used to process service requests accordingly. Transactional services can be implemented using HTTPS since it aligns with this communication pattern.

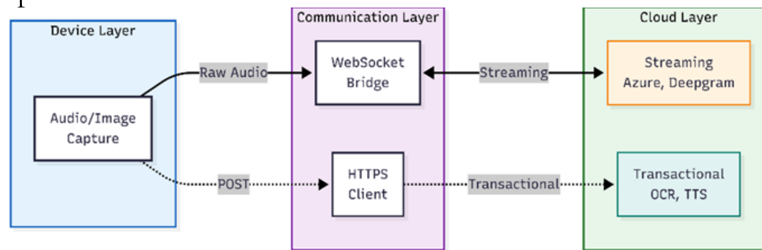


Figure 1. System architecture with streaming and transactional service patterns.

To support reproducibility and clarify the two distinct communication patterns present in the architecture, each interaction type is further documented with a dedicated flow diagram. Figure 2 presents the streaming WebSocket flow and Figure 3 presents the transactional HTTPS flow. Each diagram captures the complete data flow across device, communication, and cloud layers, including protocol transitions, request and response paths, and explicitly labeled performance measurement points used during evaluation.

WebSocket Server:

The WebSocket server is an intermediary between cloud services and ESP32 microcontrollers, which manages the in-between communication protocols. ESP32 connects with the server and transmits audio to it. This audio is then sent by the server to the chosen cloud service provider with authentication and protocol-specific needs being handled there.

Streaming authentication protocols are designed differently by various cloud providers. Deepgram and Azure utilize API keys for authentication and to establish client-side connections. These provider specific needs are met by the bridge server which provides a standard WebSocket interface to the ESP32 devices.

The bridge server is implemented in Python 3.11 with asyncio, websockets library, and aiohttp, and is deployed on the local LAN to mimic a real-world edge deployment. High resolution, monotonic server-side timestamps are obtained using time.perf_counter() function.

Figure 2 shows the entire data flow and all measurement points in the WebSocket streaming path. The flow is as follows:

Audio capture:

The ESP32-S3 records audio via the onboard PDM MEMS microphone (I2S interface) at 16 kHz, 16-bit mono in 320-byte chunks (10 ms per chunk). A dual-core FreeRTOS architecture runs a capture task on Core 0 and a network worker on Core 1 with a 120-slot audio queue separating the two cores.

Frame transmission:

Each chunk has a 4-byte big-endian chunk ID and is transmitted as a binary WebSocket frame. Each chunk ID has an incremental numbering system, which enables the server to create the per-chunk arrival time registry.

Bridge relay (Deepgram):

When each frame is received, the bridge stores

$$chunk_{arrival_{registry[chunk_{id}]}} = time.perf_{counter}(\ )$$

The only timing anchor for Deepgram latency is from this registry.

Bridge relay (Azure):

When Azure's bridge receives the first audio byte, it records the time at time.perf_counter() function. Subsequent latency runs this baseline, adding per-utterance offset and duration from Azure Speech SDK.

Partial result (Azure):

$latency_{ms} = perf_{counter(x)} - (stream_{start_{time}} + offset_s).$

Partial Result (Deepgram):

$latency_ms = perf_counter() - chunk_arrival_registry[last_chunk_id - chunk_delta], \text{ where } chunk_delta = phrase_len_sec / 0.010.$

Final result (Azure):

$latency_ms = perf_counter() - (stream_start_time + offset_s + duration_s)$, with the 500 ms silence window added.

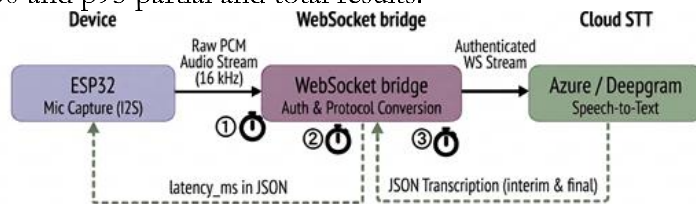
Final Result (Deepgram):

$latency_ms = perf_counter() - chunk_arrival_registry\ [last_chunk_id]$

Device receipt:

The JSON result is received by the ESP32 on Core 1. The latency_ms field is computed by the server, not the device. The partial result set is discarded if it is stale.

Note: The two providers employ different types of latency anchors and the numbers cannot be compared directly. While Azure is based on wall clock speech timing through the SDK offset and duration, Deepgram is based on per-chunk arrival timestamps. Both are self-consistent in their clock. On the end of the session, the bridge provides a summary JSON containing mean, p50 and p95 partial and total results.



Number	Description	
①	stream_start_time	Azure: perf_counter() on first audio byte
②	chunk_arrival_registry	Deepgram: perf_counter() per chunk
③	now	perf_counter() at recognition callback

Figure 2. WebSocket Bridge Server Architecture and Measurement points for Streaming Services (STT).

Direct HTTPS for Transactional Services:

Azure HTTPS STT, Azure TTS, and Azure OCR are accessed directly over HTTPS without the use of the WebSocket bridge, since these services are not persistent and only utilize a single request-response.

For OCR, the images are posted in binary form, and the service provides the structured text asynchronously, while TTS (single utterance) requires the payloads to be Speech Synthesis Markup Language (SSML) payloads per request. Relaying such services via the WebSocket bridge would not provide any functional value but would increase the number of relay hops. This corresponds to the idea that the protocol used for any service should depend on the pattern of interaction of that service and not be used by all services in the system.

Five timestamps are acquired for each iteration and are recorded to measure the duration of each phase of the connection lifecycle as shown in Figure 3. The total TCP connection overhead and TLS handshake overhead is performed on each iteration and is intentionally added to accurately model the actual per-request cost in a stateless deployment:

t_start: Captured using esp_timer_get_time() function before client.connect(). Contains DNS resolution, TCP setup and TLS handshake.

t_connected: after a client.connect() has been performed. The interval (t_connected − t_start) is used to isolate the TLS overhead.

t_req_sent: Captured when the last byte of the request body is sent. The interval (t_req_sent − t_connected) corresponds to the data transmission time.

t_first_byte: When the first response byte is received. The time t_first_byte-t_req_sent is the cloud-side processing latency.
t_last_byte: Captured when the entire response is received. The end-to-end latency is given by the difference $\Delta t = t_{last_{byte}} - t_{start}$.
Radio state is allowed to settle with a 500 ms cooldown between iterations.

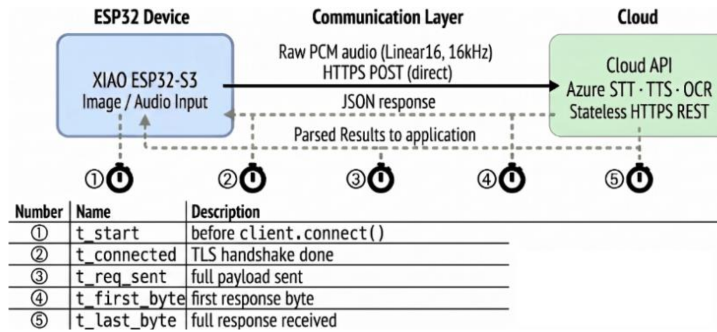


Figure 3. Transactional HTTPS flow and measurement points

Experimental Setup:

All experiments were performed under repeatable and controlled conditions with fixed audio and/or image payloads (where applicable) to avoid input variation.

Hardware & Software:

Component	Specification
Microcontroller	ESP32-S3 dual-core Xtensa LX7, 240 MHz, 512 KB SRAM + 8 MB PSRAM
Microphone	Onboard PDM MEMS (I2S), Linear16 PCM, 16 kHz mono
Connectivity	2.4 GHz Wi-Fi 802.11 b/g/n, power save disabled
Firmware	Arduino/ESP-IDF v5.x, dual-core FreeRTOS; DMA: 8×256 frames
Bridge server	Python 3.11, asyncio, websockets, aiohttp (hosted on local LAN)
Timing Functions	esp_timer_get_time() on device, time.perf_counter() on server

Cloud & Test Configuration:

Parameter	Azure STT	Deepgram STT
Service, Region	Azure Cognitive Services, Southeast Asia	Nova-2, Global
Mode	Continuous, 500 ms silence timeout	Streaming, Voice Activity Detection
Audio format	Linear16, 16 kHz, mono	Linear16, 16 kHz, mono
Test duration	120s, live mic, 320-byte chunks	120s, live mic, 320-byte chunks
Statistics	mean ± SD, min, max, p50, p95, p99	mean ± SD, min, max, p50, p95, p99

The fixed audio buffer was used for HTTPS testing to isolate protocol and network latency from recording variability. The bridge server was hosted on the local LAN to reflect a realistic edge deployment.

Benchmark Configurations and Measurement Approach:

The goal is to evaluate architectural viability for a conversational assistive application, and not to compare/benchmark between providers. The device-side clock and the server-side clock are not synchronized; all latency intervals are measured in the same clock domain to eliminate cross-clock subtraction and drift errors. The computed $latency_{ms}$ is also reported on each JSON result and logged by the ESP32, without cross-domain arithmetic. Percentiles are calculated using rank-order index truncation (floor method) for all services.

Streaming STT (Azure and Deepgram) used identical ESP32 firmware over the WebSocket bridge, with 120-second live microphone sessions and 320-byte chunks at 16 kHz. Azure silence timeout was set to 500 msec and Deepgram adopted Voice Activity Detection (VAD)-based segmentation. The latency is calculated server-side using perf_counter() function, which returns mean, p50 and p95 for partial and final results. Note that Azure WS partial latency is based on the speech start to first visible text delay the user perceives, meaning that the p95 is increasing with the length of the utterance and not with the network latency overhead; this can result in a higher latency with longer utterances. HTTPS Speech-to-Text (STT) employed a fixed 3s PCM buffer (96,000 bytes) stored in PSRAM and reused in 30 iterations with a cooldown period of 500ms. The latency is measured on device using esp_timer_get_time() function.

TTS submitted 10 SSML phrases in 50 trials (500 trials total); each phrase had 5 warmup trials, and the inter-trial delay was 300ms. Latency is measured on device from POST to HTTP 200 (micros()).

OCR was simulated on a PC using a Python/OpenCV pipeline, with images preprocessed to 640×480 grayscale JPEG at quality 55. Both submission latency and total latency (including 50 ms polling) were captured over 50 trials per image, with 5 warmup trials discarded, timed via perf_counter() function.

Formal latency definitions are given in Table 3.

Table 3. Latency definitions by service and clock domain.

Service	Metric	Formula	Clock
Azure STT	Partial latency	$perf_counter() - (start + offset)$	Server
Azure STT	Final latency	$perf_counter() - (start + offset + duration)$	Server
Deepgram STT	Partial latency	$perf_counter() - registry[last_id - delta]$	Server
Deepgram STT	Final latency	$perf_counter() - registry[last_id]$	Server
HTTPS STT	End-to-end	$t_last_byte - t_start\ (incl. TLS)$	Device
TTS	API response	$t_response - t_post$	Device
OCR (request)	Submission	$t_202 - t_post$	PC
OCR (total)	End-to-end	$t_result_ready - t_post\ (incl. polling)$	PC

Result and Discussion:

This section provides latency data for all the services and communication protocols tested, classified by services and interaction pattern. Streaming STT reports mean, p50, and p95. All other services report means, SD, min, max, p50, p95, and p99; OCR also includes a 95% confidence interval. Warmup iterations are excluded throughout, with p95 and p99 being emphasized as they are more likely to cause high-latency spikes that are more harmful to assistive applications than an elevated mean.

All the latency results for the services evaluated are summarized in Table 4.

Table 4. Cross-Service latency summary.

Service	N	Mean (ms)	p50 (ms)	p95 (ms)	Key Note
Deepgram WS (Partials)	439	976	1002	1599	Utterance-accumulation anchor
Deepgram WS (Finals)	388	8	7	16	Near-zero; chunk-arrival anchor
Azure WS (Partials)	90	1768	1514	3809	Wall-clock speech-onset anchor

Azure WS (Finals)	26	659	649	852	Incl. 500 ms silence window
Azure HTTPS STT	302	2574	2535	2900	Fixed 3s buffer; per-request TLS
Azure TTS	497	1045	1033	1239	POST to HTTP 200; SD 134 ms
Azure OCR (Total)	250	1888	1719	2879	Incl. async polling at 50 ms
Azure OCR (Request)	250	929	924	1037	POST to HTTP 202 only

WebSocket Streaming STT:

Experiments were conducted with both Deepgram Nova-2 and Azure Cognitive Services using the WebSocket bridge over a 120-second live microphone channel. Results are presented separately by provider as they have differing structures in their latency anchoring strategies.

Deepgram Nova-2 partial results had a mean of 976 ms (p50 1002 ms, p95 1599 ms), using the arrival timestamp of the last chunk of audio for each phrase as the anchor. The latency is thus a measure of phrase length, not network overhead. Azure partial results had a mean of 1768 ms (p50: 1514 ms, p95: 3809 ms), with the wall-clock onset of each phrase being used as a reference point, due to the SDK offset field which measures elapsed time from when the user started speaking to when the partial callback was fired. This is the ideal operationally correct anchor for assistive use which is the time gap between speech start and the onset of visible text.

Because of this structural difference, absolute partial latency values from different providers cannot be compared. However, as shown in figure 4, both are below the conversational perceptibility threshold of 2000 ms on mean. Final results, as mean 8 ms, p95 16 ms and 659 ms for Azure (reflecting the embedded 500 ms silence segmentation window) — are not designed for cross-provider comparison, as they are structurally different.

The near-zero final latency for Deepgram (mean 8 ms, p95 16 ms) stems from its chunk-arrival anchor, where the final callback is triggered when the last audio chunk is received, and the measured interval may not be interpreted as an indication of network or recognition performance.

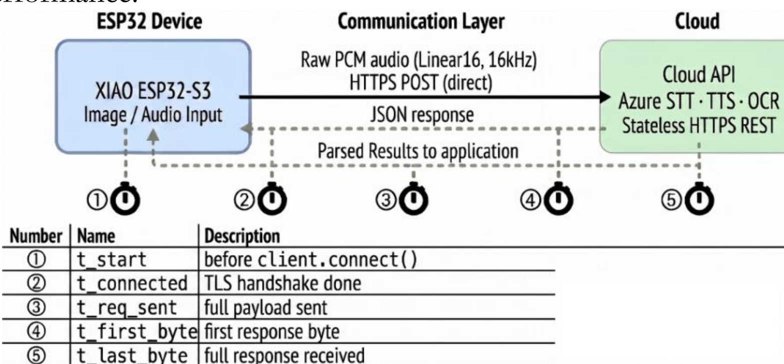


Figure 4. WebSocket STT partial latency distributions for Deepgram and Azure.

Protocol Comparison: WebSocket vs. HTTPS:

Both transports used the same underlying recognition engine, and the PCM buffer (96,000 bytes) was fixed for 30 iterations of HTTPS to isolate per-request protocol overhead.

As Figure 5 shows WebSocket partials' mean (1768 ms) outperform HTTPS (2574 ms) by approximately 1.5×. The breakdown of the HTTPS total shows that the per-request TLS handshake (~574 ms), TCP connection and upload (~1775 ms), and cloud processing (~225 ms) all contribute to the overhead, with the final one being uniform across both transports, indicating that the overhead is entirely protocol-driven.

The comparison also flips at p95: Azure WS partials are 3809ms while HTTPS is 2900ms, as show in table 5. This is not a sign of WebSocket tail-latency issues. The p95 inflation in WebSocket partials is a direct consequence of the phrase-onset anchoring (e.g., a 4-second utterance would add ~4000 ms to the anchor offset by definition) as longer utterances lead to more elapsed time between speech_start_wall_clock and stream_start_time, as is observed in the bridge implementation:

$$speech_{start_{wall_{clock}}} = stream_{start_{time}} + offset_{seconds}$$

The WebSocket p95 isn't a protocol ceiling, it's a sentence-length ceiling, and cannot be directly compared to the HTTPS p95, which is a fixed per-request round-trip ceiling. Moreover, the fixed 3-second HTTPS buffer is a lower limit; if the utterances were variable length, the upload time would increase based on the size of the buffer, extending the gap even further in favor of WebSocket.

Table 5. WebSocket vs. HTTPS (same Azure backend)

Metric	Azure WS STT (Partials)	Azure HTTPS STT
Mean latency	1768 ms	2574 ms
p95 latency	3809 ms	2900 ms
TLS overhead	None (persistent)	~574 ms per request
Mean ratio	1× baseline	~1.5× higher
p95 source	Utterance-length accumulation	Fixed per-request round-trip

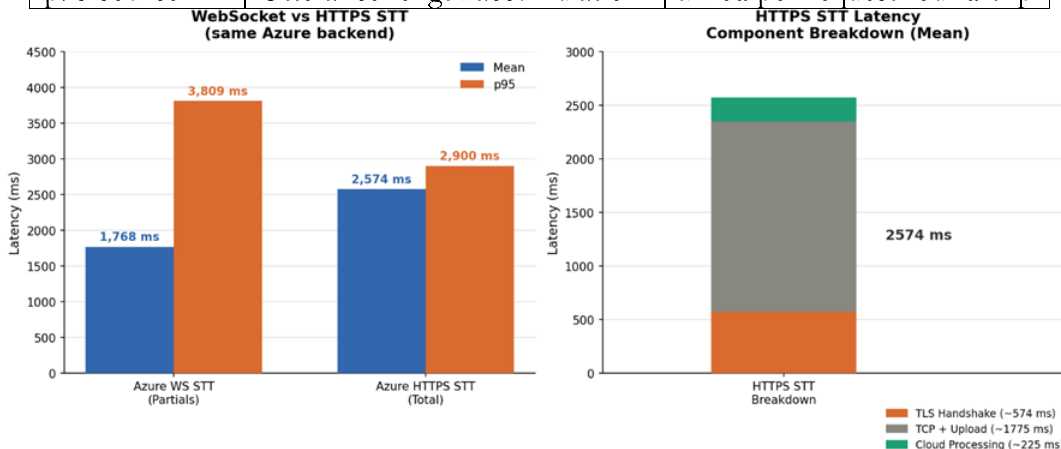


Figure 5. WebSocket vs. HTTPS STT latency distributions (left). HTTPS component breakdown (Right).

In the case of embedded assistive hardware, it is strongly preferred to maintain a WebSocket connection for continuous speech recognition. A minimum average of ~2.5 seconds per request (with p95 being ~2.9 seconds under controlled, fixed input) is a factor to be considered when designing a system that must use HTTPS.

Transactional HTTPS Services

Azure TTS was consistent with a mean of 1045 ms (SD 134 ms, p50 1033 ms, p95 1239 ms) and 497 trials, showing predictable response behavior. The narrow SD shows that single-utterance TTS works well for stateless HTTPS, where each request is independent and the use of a persistent connection does not provide any advantage.

Azure OCR total latency recorded a mean of 1888 ms (SD 403 ms, p50 1719 ms, p95 2879 ms). The wider distribution is due to the asynchronous polling overhead that is an expected part of Azure's HTTP 202 accepted pattern. Request-only latency (POST to HTTP 202) is much more consistent at mean 929 ms (SD 62 ms, p95 1037 ms); Figure 6 shows that this variability comes from the polling phase. Applications should strive to achieve total delays of less than 2.9 seconds for the OCR pipelines at p95. Routing either service through the WebSocket bridge would be redundant in terms of service.

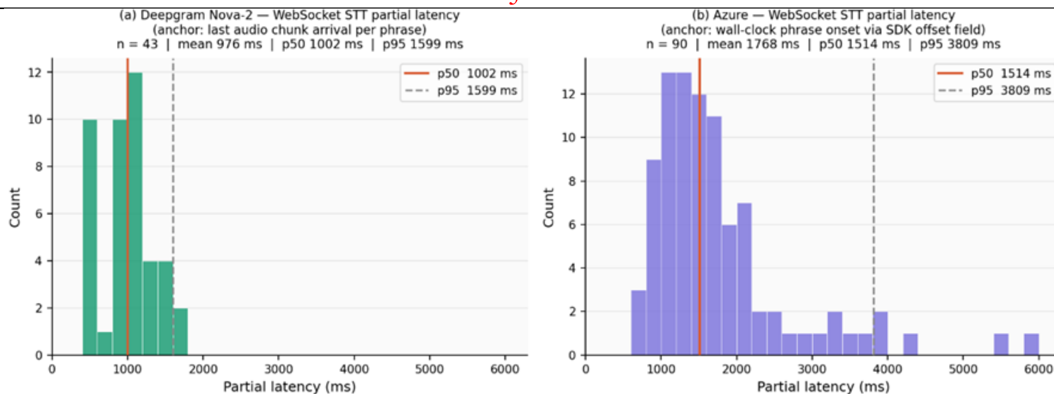


Figure 6. TTS and OCR latency distributions, showing total vs. request-only for OCR.

Cross-Service Synthesis: Protocol Must Match Interaction Pattern:

The obtained results validate a two-protocol architecture. Figure 7 summarizes that streaming STT over WebSocket delivers mean partial latencies of 976 ms (from Deepgram) or 1768 ms (from Azure), both of which are below the perceptibility threshold; when streaming via HTTPS, mean latency is increased by 1.5× to 2574 ms, due entirely to per-request TLS overhead. The apparent p95 reversal in favor of HTTPS, detailed in Table 5, does not conflict with this result: It is an accumulation of utterances at the phrase onset anchor, not a disadvantage of the protocol. Services like TTS (mean 1045ms, p95 1239ms) and OCR (mean 1888ms, p95 2879ms) are performant over direct HTTPS and do not rely on WebSocket relay.

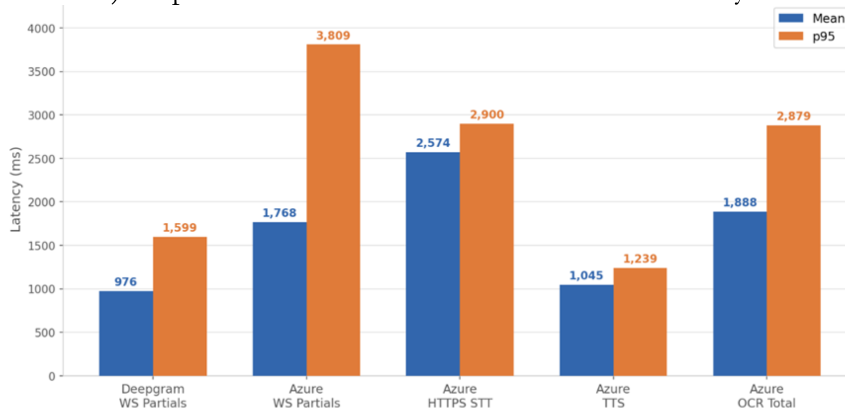


Figure 7. Cross-service mean and p95 latency for all services under evaluation.

Thus, protocol selection is an engineering need for an embedded assistive hardware system under real-time conditions, and it directly impacts whether the system is perceptible or not.

Recommendations and Future Directions:

Recommendations for Deployment:

Multi-provider evaluation should include adding other cloud AI providers to Azure and Deepgram testing, with interoperability and benchmarking across various ecosystems. Scalability needs to be evaluated by running multiple ESP32 devices connected together to the bridge server, and checking the throughput and resource consumption under multi-concurrent operation. There should be multilingual support beyond English to tackle the latency and accuracy issues in low-resource languages. The bridge server should be deployed with more robust security features, such as encrypted WebSocket tunnels and handling authentication tokens. Lastly, longitudinal studies with end-users are needed to assess maintainability, firmware update cycles and user satisfaction in the long term.

Future Directions:

There are several avenues for future research. A degraded or unstable network condition evaluation would set the limits of resilience of the architecture outside the controlled

LAN environment of this test. The bridge abstraction should be evaluated for cross-language (Multilingual STT) and cross-provider (multi-provider failover) support. Studies using target users (people with hearing or speech loss) over longer periods of time are required to confirm the value of the latency parameters obtained and prove that they correspond to a significant advance in communication fluency. Finally, there is a need to formalize security aspects such as handling of authentication tokens, encrypted WebSocket tunnels and mechanisms for the firmware update before implementation in an actual assistive setting.

Conclusion:

The paper presents a hybrid WebSocket–HTTPS architecture to incorporate cloud AI services into resource-constrained assistive communication devices. The architecture explicitly associates the choice of communication protocol with the interaction pattern of each service – for instance, persistent WebSocket connections for latency sensitive streaming applications, and standard HTTPS for stateless transactional applications. A WebSocket bridge server manages provider-specific authentication and protocol processing, reduces complexity of embedded firmware from the cloud API, and provides better long-term maintainability.

Mean partial latencies over websocket were 976 and 1768 ms for Deepgram Nova-2 and Azure Cognitive Services respectively, both below the 2000 ms threshold for conversational perceptibility, whereas mean latency measured over HTTPS was 2574 ms, or about 1.5× the latency, mostly in terms of per-request TLS handshake overhead. These values are within acceptable human perception limits for interactive conversation, resulting in meaningful real-time performance in embedded hardware by the protocol-matching approach. The architectural decision to avoid the bridge in request-response interactions was confirmed by transactional services achieving mean latencies of 1045 ms for TTS and 1888 ms for OCR over direct HTTPS, making complete responses within reasonable delays based on the use-case of each service.

The main value of this work is related to integration methodology instead of algorithmic innovation. The findings indicate that attentive coordination between service properties and communication protocol, coupled with the layered abstraction among device and cloud, is a useful design methodology to deploy advanced AI functions on assistive embedded platforms. The greater adoption of these standardized abstraction layers would contribute to a decrease in barriers for development, reduce integration costs, and increase the interoperability between assistive technology ecosystems, increasing access to capable communication devices for those who need them the most.

Acknowledgement: The authors would like to express their gratitude to the Department of Electronics Engineering, Mehran University of Engineering and Technology (MUET), Jamshoro, for the laboratory facilities and academic support provided to conduct the research. The authors also gratefully acknowledge Microsoft Azure and Deepgram for providing free tier API credits for the purpose of cloud service validation in this work.

Author's Contribution: Abuzar Soomro[1] conceived the system architecture, developed the ESP32 firmware, conducted experimental validation, and wrote the manuscript.

Aqsa Younus[2] assisted in the implementation of the WebSocket bridge server, performed latency measurements, and contributed to the results and discussion sections.

Bhave Sagar[3] carried out the literature review, assisted in data collection, and contributed to manuscript preparation.

Prof. Dr. Kehkashan Asma Memon[4] supervised the research, provided critical guidance on system design, and reviewed the manuscript.

Prof. Dr. Saba Baloch[5] co-supervised the research, provided feedback on the methodology, and revised the manuscript critically.

Bilal Khan[6] assisted in testing the HTTPS transactional services and supported the experimental setup.

Conflict of interest: The authors declare that there is no conflict of interest regarding the publication of this manuscript in the International Journal of Innovations in Science and Technology (IJIST).
Project details: This research was conducted as part of a Final Year Project (FYP) at the Department of Electronics Engineering, Mehran University of Engineering and Technology (MUET), Jamshoro. The project aimed at developing an accessible communication device for individuals with hearing and speech impairments. The architecture described in this paper was created and tested as a basic element of that ongoing work. The whole system is still being developed and has not yet been officially documented or published.

References:

[1] "A Survey: Embedded Systems Supporting By Different Operating Systems." Accessed: Jun. 06, 2026. [Online]. Available: https://www.researchgate.net/publication/303280521_A_Survey_Embedded_Systems_Supporting_By_Different_Operating_Systems

[2] R. Malviya and S. Rajput, "AI-Driven Innovations in Assistive Technology for People with Disabilities," pp. 61–77, 2025, doi: 10.1007/978-981-96-6069-8_4.

[3] "Computer-mediated discourse analysis: an approach to researching online communities." Accessed: Jun. 06, 2026. [Online]. Available: https://www.researchgate.net/publication/285786435_Computer-mediated_discourse_analysis_an_approach_to_researching_online_communities

[4] Antje S. Meyer, "Timing in Conversation," *J. Cogn.*, 2023, [Online]. Available: https://journalofcognition.org/articles/10.5334/joc.268

[5] Julien Mineraud, Oleksiy Mazhelis, "A gap analysis of Internet-of-Things platforms," *Comput. Commun.*, vol. 89–90, 2016, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366416300731

[6] E. Dinçer and Z. H. Kilimci, "Real-time and offline large language models on edge devices: a systematic review," *PeerJ Comput. Sci.*, vol. 12, 2026, doi: 10.7717/PEERJ-CS.3769/.

[7] Nisha Saini, Jitender Kumar, "A PRISMA-Based Systematic Review of Cloud- Edge Orchestration Using the MAPE-K Framework," *Int. J. Electr. Electron. Eng. Telecommun.*, vol. 14, no. 3, pp. 130–146, 2025, doi: 10.18178/ijeetc.14.3.130-146.

[8] J. Liu *et al.*, "Edge-Cloud Collaborative Computing on Distributed Intelligence and Model Optimization: A Survey," Aug. 2025, doi: 10.1109/COMST.2026.3669216.

[9] N. Mitrovic, M. Dordevic, S. Veljkovic, and D. Dankovic, "Implementation of WebSockets in ESP32 based IoT Systems," *2021 15th Int. Conf. Adv. Technol. Syst. Serv. Telecommun. TELSIKS 2021 - Proc.*, pp. 261–264, 2021, doi: 10.1109/TELSIKS52058.2021.9606244.

[10] "RFC 6455 - The WebSocket Protocol." Accessed: Jun. 06, 2026. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6455

[11] M. Kavis, "Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, AND IaaS)," *Archit. Cloud Des. Decis. Cloud Comput. Serv. Model. (SaaS, PaaS, IaaS)*, pp. 1–199, Jan. 2014, doi: 10.1002/9781118691779.

[12] Mohd Tamizan Abu Bakar, "Latency Issues in Internet of Things: A Review of Literature and Solution," *Int. J. Adv. Trends Comput. Sci. Eng.*, vol. 9, pp. 83–91, 2020, doi: 10.30534/ijatcse/2020/1291.32020.

[13] "High Performance Browser Networking (O'Reilly)." Accessed: Jun. 06, 2026. [Online]. Available: https://hpbn.co/

[14] Dora Kreković, Petar Krivić, "Reducing communication overhead in the IoT–edge–cloud continuum: A survey on protocols and data reduction strategies," *Internet of Things*, vol. 31, p. 101553, 2025, doi: https://doi.org/10.1016/j.iot.2025.101553.

[15] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," *Rfc 7252*, p. 112, 2014, [Online]. Available: https://www.rfc-editor.org/rfc/pdfrfc/rfc7252.txt.pdf

[16] Amirhossein Farahzadi, Pooyan Shams, "Middleware technologies for cloud of things: a survey," *Digit. Commun. Networks*, vol. 4, no. 3, pp. 176–188, 2018, doi: https://doi.org/10.1016/j.dcan.2017.04.005.

[17] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Cla, "Middleware for internet of things: A survey," *IEEE Internet Things J.*, vol. 3, no. 1, pp. 70–95, Feb. 2016, doi: 10.1109/JIOT.2015.2498900.

[18] A. S. M. Kayes *et al.*, "A survey of context-aware access control mechanisms for cloud and fog networks: Taxonomy and open research issues," *Sensors (Switzerland)*, vol. 20, no. 9, May 2020, doi: 10.3390/S20092464.

[19] Mariana Arroyo Chavez, Molly Feanny, "How Users Experience Closed Captions on Live Television: Quality Metrics Remain a Challenge," *Conf. Hum. Factors Comput. Syst. - Proc.*, pp. 1–6, 2024, [Online]. Available: https://dl.acm.org/doi/10.1145/3613904.3641988

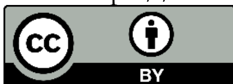
[20] Raja S. Kushalnagar, Walter S. Lasecki, "Accessibility Evaluation of Classroom Captions," *ACM Trans. Access. Comput.*, vol. 5, no. 3, pp. 1–24, 2014, [Online]. Available: https://dl.acm.org/doi/abs/10.1145/2543578

[21] Stefano Di Leo, Luca De Cicco, "Real-Time Speech-to-Text on Edge: A Prototype System for Ultra-Low Latency Communication with AI-Powered NLP," *Information*, vol. 16, no. 8, p. 685, 2025, doi: https://doi.org/10.3390/info16080685.

[22] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo and J. Zhang, "Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing," *Proc. IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019, doi: 10.1109/JPROC.2019.2918951.

[23] Umar Islam, Mohammed Naif Alatawi, Ali Alqazzaz, Sulaiman Alamro, Babar Shah & Fernando Moreira, "A hybrid fog-edge computing architecture for real-time health monitoring in IoMT systems with optimized latency and threat resilience," *Sci. Rep.*, 2025, [Online]. Available: https://www.nature.com/articles/s41598-025-09696-3

[24] Abozariba, Haitham Mahmoud & Raouf, "A systematic review on WebRTC for potential applications and challenges beyond audio video streaming," *Multimed. Tools Appl.*, vol. 84, pp. 2909–2946, 2025, [Online]. Available: https://link.springer.com/article/10.1007/s11042-024-20448-9



Copyright © by authors and 50Sea. This work is licensed under Creative Commons Attribution 4.0 International License.