Production Systems and Information Engineering
Volume 13 (1), pp. 46-63
doi: https://doi.org/10.32968/psaie.2025.1.5

Real-Time, low audio latency based AI-Powered application architecture design

PÉTER MILEFF
University of Miskolc, Hungary
Institute of Information Technology
peter.mileff@uni-miskolc.hu

Abstract:
This paper presents the design and implementation of a mobile application that provides users with an interactive conversational experience powered by OpenAI's language model. A key feature of this application is its real-time text response streaming, coupled with synchronized audio synthesis using Azure's text-to-speech (TTS) services. The architecture includes a Node.js backend server that handles OpenAI communication in streaming mode, sentence segmentation for response buffering, and a dedicated, multithreaded audio service for efficient TTS conversion. Parallelized webSocket communication enables high throughput real-time coordination between the backend and the audio service. This paper explores the system's architecture, implementation challenges, performance evaluation, and potential applications in education, accessibility, and virtual assistants.

Keywords: OpenAI, low latency audio, parallel processing, websocket

1. Introduction

Over the past decade, the rapid advancement of artificial intelligence (AI) and natural language processing (NLP) technologies has significantly redefined the landscape of human–computer interaction. One of the most prominent breakthroughs in this domain is the emergence of large-scale transformer-based language models, such as OpenAI's GPT series, which are capable of generating coherent, context-aware responses to open-ended user input. These models have demonstrated state-of-the-art performance across a wide variety of tasks, ranging from language translation and summarization to question answering and creative content generation. Their availability via APIs has enabled developers to integrate intelligent conversational capabilities into mobile and web-based applications with relative ease.

Despite these advances, most AI-powered chat interfaces today remain largely text-based. While textual interaction is sufficient for many scenarios, it is not always the most effective or inclusive form of communication. In particular, voice-based interactions offer substantial advantages in terms of accessibility, engagement, and naturalness. Voice responses are crucial for users with visual impairments, for hands-free operation in mobile environments, and for creating more immersive digital assistants that closely mimic human conversational patterns. Furthermore, multimodal systems—those

combining text, voice, and visual feedback—have been shown to enhance user retention, learning outcomes, and emotional connection with the application.

However, delivering real-time, voice-enhanced conversations with AI presents several architectural and technical challenges. Unlike static or pre-generated content, chatbot responses are typically generated dynamically, and in many cases, incrementally through streaming APIs. This means that a system aiming to synthesize speech from AI-generated text must be capable of handling partial or segmented content, initiating audio generation while the conversation is still ongoing. Achieving low-latency speech synthesis in such a streaming environment, while ensuring proper sequencing and voice consistency, requires a highly optimized and parallelized infrastructure.

Equally important is the need for scalability and responsiveness. In a mobile context, users expect near-instantaneous feedback, both in textual and audio forms. The system must be designed to support multiple concurrent conversations, with efficient management of resources, minimal delays, and accurate alignment between spoken and written output. These requirements are particularly critical when targeting general-purpose use cases such as digital personal assistants, educational tutors, or voice-based customer service agents.

To address these needs, we propose a modular and high-performance architecture that combines OpenAI's advanced text generation capabilities with Azure's high-quality text-to-speech (TTS) service, orchestrated through a distributed backend infrastructure that supports real-time, sentence-level audio synthesis.

2. Related work

The field of conversational AI has experienced rapid growth over the last decade, driven by advances in natural language processing (NLP), deep learning, and speech synthesis technologies. This section reviews key developments in conversational agents, text-to-speech systems, multimodal interfaces, and architectural solutions for real-time AI applications.

2.1 Conversational Agents and Language Models

Traditional chatbots were based on rule-based or retrieval-based approaches, which relied on predefined scripts or matching heuristics to simulate dialogue (e.g., ELIZA, AIML-based systems). While these systems had limited flexibility, they laid the groundwork for more dynamic approaches.

Recent years have seen the emergence of generative language models such as OpenAI's GPT-3 and GPT-4, Google's PaLM, and Meta's LLaMA, which leverage transformer architectures to produce highly contextual and coherent responses. These models have demonstrated state-of-the-art performance on a wide range of NLP benchmarks and are now widely used in research and commercial applications. The ability to stream responses token-by-token from these models (e.g., via OpenAI's streaming API) allows for more responsive interaction, a feature crucial for real-time applications. Notable platforms such as ChatGPT, Bing Chat, and Claude AI have demonstrated the potential of these models in production environments. However, these platforms typically focus on desktop or web use cases, and often do not provide audio synthesis as an integrated feature, especially not in a streaming context.

2.2 Text-to-Speech Synthesis (TTS)

Text-to-speech systems have evolved from concatenative and parametric models to neural architectures. WaveNet, developed by DeepMind, introduced a new paradigm in speech synthesis by producing natural-sounding audio using deep generative models. Building upon this, commercial services such as Microsoft Azure's Neural TTS, Google Cloud Text-to-Speech, and Amazon Polly now offer high-quality, low-latency speech generation in multiple languages and voices.
Azure's TTS in particular is designed for scalable cloud deployments and provides SDKs that support parallel processing. Previous works (e.g., [1], [2]) have shown the advantages of multithreaded TTS pipelines in reducing latency for dynamic dialog generation. However, many implementations assume pre-processed, non-streamed text input, limiting their real-time applicability in streaming LLM contexts.

2.3 Multimodal and Assistive Applications

Several systems have been developed to support multimodal interaction. For example, Google Assistant and Apple Siri combine speech recognition, NLP, and TTS to offer a fully voice-driven interface. Similarly, research systems such as Microsoft's XiaoIce and Facebook's BlenderBot integrate vision, speech, and language capabilities to create engaging interactions.
While these systems provide advanced capabilities, they are generally closed-source or tightly integrated with proprietary ecosystems. Moreover, they do not provide modular backend architectures that can be independently deployed or customized, limiting their use in research and domain-specific applications.
In the accessibility domain, multimodal systems have shown substantial benefits for users with disabilities. The use of text-plus-audio interaction has been found to improve comprehension for users with dyslexia [3], and to provide essential support for visually impaired users navigating digital content [4]. However, integrating this functionality into custom mobile applications still requires significant engineering effort.

2.4 Real-Time Streaming Architectures

Real-time NLP systems require low-latency communication between various components, especially when handling streaming data. Several studies (e.g., [5], [6][11]) have explored the use of WebSockets and microservice-based designs to achieve scalability and responsiveness in AI-driven systems [12][13].
Recent work on streaming LLM integration (e.g., [7]) demonstrates the feasibility of parsing token streams into semantically meaningful segments in real time. However, few studies have addressed the challenge of synchronizing such dynamic text generation with concurrent, parallel TTS synthesis.
While substantial research has been done in the domains of conversational AI, TTS, and multimodal applications, the integration of real-time streamed language model output with parallel, sentence-based audio synthesis remains a relatively unexplored area. Our work addresses this gap by providing a modular, efficient architecture suitable for mobile deployment, with a focus on both user experience and backend scalability.

3. Importance of Low-Latency Audio Conversion

Low-latency audio conversion plays a central role in the effectiveness and usability of multimodal conversational systems, especially those intended for mobile and real-time applications. In the context of AI-driven chatbots, where text responses are generated dynamically, the ability to transform those responses into natural-sounding speech with minimal delay is critical for delivering a seamless and human-like user experience. This section outlines the functional and technical importance of minimizing audio synthesis latency and highlights its impact on perceived system responsiveness, user satisfaction, and accessibility.

3.1 Real-Time Interaction and User Expectations

Modern users expect conversational applications to respond almost instantaneously, particularly when they resemble human-to-human interactions. Any perceptible delay between the display of a chatbot's text response and the corresponding voice playback can disrupt the flow of interaction and reduce user immersion. According to cognitive studies on dialogue systems, latencies above 300–500 milliseconds are often noticed by users, while latencies above 1 second can result in frustration or abandonment.

In applications like virtual assistants, language tutors, or assistive technologies for visually impaired users, low-latency audio feedback is essential. These users rely heavily on audio output not just as an optional feature, but as the primary means of consuming content. Therefore, ensuring that synthesized audio is delivered promptly after the text is generated is not a mere enhancement—it is a core usability requirement.

3.2 Audio Conversion as a Parallel Process

One of the main challenges in low-latency systems is the inherently sequential nature of text generation, especially when models like OpenAI's GPT are used in streaming mode, producing output token by token. This makes it difficult to wait for the full response before initiating audio synthesis without significantly delaying the voice output. To mitigate this, the system must adopt a sentence-level streaming strategy, where partial responses are segmented into sentences and sent immediately for audio processing.

Low-latency audio conversion thus becomes a pipeline problem: as soon as a complete sentence is available, it should be synthesized and returned in parallel with the ongoing text generation. The faster each sentence can be converted to audio, the more natural and responsive the interaction becomes. This architecture imposes strict demands on the audio synthesis pipeline, which must be designed for concurrency, parallelism, and thread-safe sequencing to ensure that audio segments are processed and returned in the correct order without delay.

4. System Architecture

In any complex software system, the architecture serves as the foundational blueprint that governs how components interact, scale, and perform. For real-time, multimodal applications—such as the mobile chatbot system described in this paper—architectural design is not just a matter of organization; it directly impacts critical system qualities

including performance, responsiveness, scalability, reliability, maintainability, and user experience. The success of such a system depends not only on the capabilities of individual technologies like OpenAI's language models or Azure's Text-to-Speech (TTS) service, but also on how effectively these technologies are integrated and orchestrated at the architectural level.

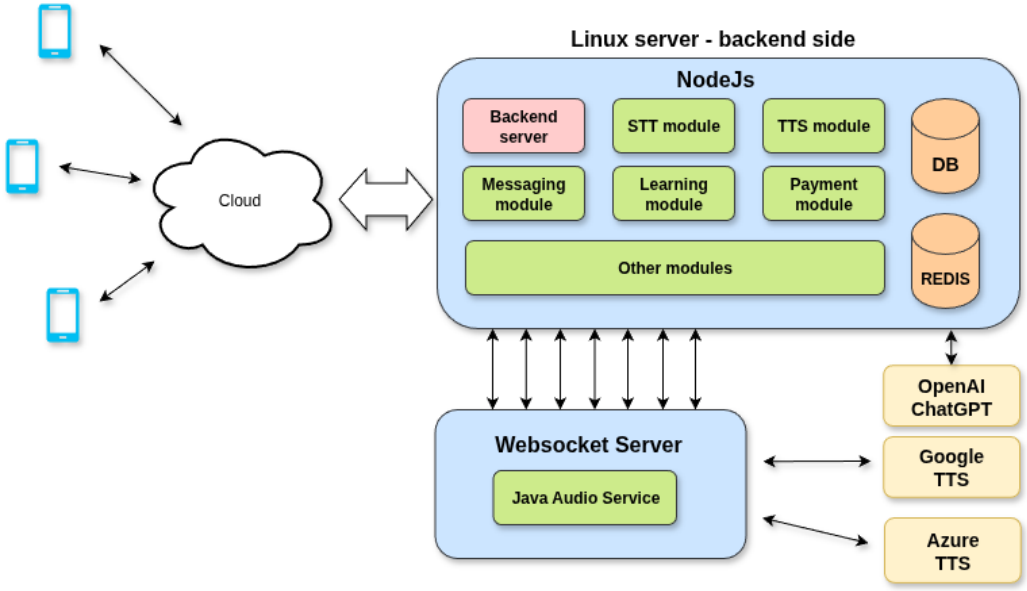

Figure 1. System Architecture Diagram

The architecture of the system is composed of the following core components:

- **Mobile Frontend:** Built using Flutter, the mobile app provides the user interface for text/audio input and audio playback.
- **Node.js Backend:** Central manager which contains several modules (Learning, Payment, etc) to handle application logic. It manages OpenAI requests in streaming mode and coordinates audio response generation.
- **OpenAI Integration:** Utilizes the OpenAI API with streaming enabled to capture responses token-by-token.
- **Audio Service (Azure TTS):** TTS service from Azure's neural voice models.
- **Audio Service (Google TTS):** TTS service from Google's neural voice models.
- **Audio Conversion Service:** This module provides a high speed, parallelized TTS audio conversion
- **WebSocket Server:** Maintains low-latency, bidirectional communication between the backend and audio service.
- **Database:** MySQL relational database to server application logic
- **Redis:** used for NodeJS side worker threads

The primary goal of the application is to support English teaching for students with audio content as well. This paper deals only with the technical background. The normal flow of application usage is the following: users open the app and initiate a conversation with the AI assistant. The application contains a variety of pre-

integrated topics within the domain of English learning, but it also provides a so-called "Ask Anything" mode using a highly customized english learning assistant prompt, where users can freely interact with the assistant.

It is crucial to clearly distinguish between these different areas, as this is the only way to provide the AI model with a precise — sometimes quite lengthy — prompt that enables fine-tuning. For example, to ensure that responses remain within the proper domain, that the response length stays within acceptable bounds, and so on.

Users can communicate with the AI assistant either via text or voice. These messages are sent to the NodeJS backend over a WebSocket connection. WebSockets are preferred over standard HTTP requests because they allow for a bidirectional communication channel between client and server [9]. This enables the efficient and immediate delivery of responses from OpenAI, which operates in streaming mode. Further details on this are discussed in later sections.

The NodeJS-based backend contains several modules responsible for the application's functionality. This article focuses exclusively on the OpenAI integration and the audio processing pipeline. After initial filtering and transformation, user requests arriving at the backend are forwarded to the OpenAI service, which processes them in the aforementioned streaming mode.

The data arriving from OpenAI is collected by the backend and sent sentence-by-sentence to the audio service via a dedicated WebSocket connection. The audio service filters the incoming sentences and forwards them in a parallelized environment to Azure/Google TTS services for audio conversion. In this context, parallelization means that the processing of every sentence is performed by a separated thread. Threads are making audio conversion requests to Azure/Google TTS and waiting for the response. Our implementation uses mainly Azure for audio conversion, but for some specific languages, Google TTS is used for better results. From the technical perspective, the process for generating audio from text is the same for both providers.

Since the audio conversion time of the sentence-segmented OpenAI responses can vary, an important challenge arises: sending back the completed audio parts to the backend in the correct order. While waiting for the audio responses, the backend already starts sending the streamed text segments back to the user via WebSocket, thereby improving the user experience.

In order for the above architecture and operation to function efficiently, an appropriate cloud infrastructure is essential. Viable alternatives include Amazon AWS, Google Cloud [10], UpCloud, etc. The current architecture was developed using Google Cloud.

5. Implementation Details

Modern AI applications increasingly rely on a composition of distributed services, often provided by third-party cloud APIs. In our case, natural language generation (via OpenAI) and speech synthesis (via Azure) are both handled by external providers. Each service introduces latency, rate limitations, and failure modes that must be managed explicitly. A well-designed architecture provides the coordination layer that:

- Buffers and synchronizes partial responses,

- Handles service-specific constraints and retries,
- Enables parallelism without sacrificing ordering or context,
- Decouples responsibilities across services and components.

Failing to properly design this coordination layer can result in bottlenecks, redundant processing, cascading failures, and unpredictable user-facing delays. In our system, the architectural separation between the backend and audio services, combined with session-specific WebSocket channels and multithreaded processing, ensures that each sentence is treated as a unit of work that can be routed, processed, and sequenced independently.

The subsequent sections have presented all aspects of the implementation in detail.

5.1. OpenAI Streaming Mode for Text Generation

Due to the goal and nature of the project, it was necessary to choose a modern AI model capable of fully implementing the personal assistant functionality across various domains. In the context of this project, we chose the widely popular OpenAI. Naturally, the decision was also influenced by prior experience with using the OpenAI model.

OpenAI's large language models (e.g., GPT-3.5, GPT-4) are capable of producing coherent and contextually rich natural language responses from user prompts [7]. A key feature relevant to real-time conversational applications is the streaming mode, which allows for incremental delivery of generated tokens over a persistent connection, rather than waiting for the entire response to be computed before transmission.

In streaming mode, the model begins emitting output tokens (typically words or subword units) as soon as the generation process starts. This is accomplished through an HTTP-based Server-Sent Events (SSE) protocol, where each token is transmitted as a data frame as it becomes available. The client can thus consume partial results in real time, enabling immediate user feedback.

Advantages for Real-Time Applications

- **Reduced Latency:** Users can see the response unfold gradually, which significantly improves perceived responsiveness. This is particularly important in spoken interfaces where audio playback can begin before the full message is generated.
- **Progressive Processing:** The application backend can parse and process parts of the response as they arrive (e.g., sentence-by-sentence), allowing for concurrent actions like text-to-speech conversion.
- **Improved UX:** Streaming mimics human conversation more naturally than fully buffered responses, which can appear delayed or abrupt.

5.1.1 Token Framing and Response Format

The Node.js backend initiates a streaming request to the OpenAI API when a user submits a message into the App. OpenAI responds in a *token framing* format. Token framing refers to the structured delivery of output one token at a time over a persistent connection using the *Server-Sent Event*s (SSE) protocol. Each token is encapsulated in a small JSON payload known as a *data frame*, which allows developers to handle the generation stream incrementally.

Structure of a Token Frame:

```
data: {"choices":[{"delta":{"content":"Hello"},"index":0}]}
data: {"choices":[{"delta":{"content":"!"},"index":0}]}
data: [DONE]
```

Hundreds of token frames may arrive from the OpenAI server in response. These are collected by the backend and then forwarded to the audio service in the appropriate format.

5.1.2 Real-Time Parsing and Aggregation

To achieve efficient, low-latency audio generation in a multi-user mobile application, our system is designed to exploit **true parallelism** by coupling OpenAI's **streamed token output** with a **sentence-level segmentation and dispatch mechanism**. This design choice is essential due to the sequential nature of token generation from large language models, where complete responses are produced incrementally. OpenAI's tokens arrive sequentially and represent partial linguistic units such as subwords, words, punctuation, or whitespace. In order to transform the generated text into a suitable input for text-to-speech (TTS) synthesis, the backend system aggregates these tokens into **complete sentences** using runtime heuristics. Sentence boundaries are detected primarily through the occurrence of terminal punctuation symbols (e.g., `.`, `!`, `?`) combined with syntactic context rules to ensure linguistic completeness.

Once a sentence boundary is identified, the sentence undergoes a filtering process that removes non-speakable artifacts such as:

- Emojis
- Special symbols (e.g., #, @, ~),
- Markdown formatting characters (e.g., *, _),
- Control characters and unsupported Unicode blocks.

This ensures that the resulting audio stream remains semantically relevant and phonetically clean for real-time auditory presentation. For example Google’s TTS service is sensitive and gives 403 HTTP errors for non-normal sentences like empty spaces, underline character, etc.

5.2. Text-to-Speech Audio Service

Because NodeJS is not specifically designed for parallelism, it quickly becomes clear that if we want to make the most of the parallelization opportunity and build

an audio service based on truly parallel processing, it is advisable to put the service in an independent software component, as we called *Audio Service*. Our implementation was based on Java technology.

The Audio Service is designed as a multithreaded daemon that listens for incoming WebSocket connections from the backend. Each connection represents a unique user session, allowing the service to isolate and manage sentence queues per session.

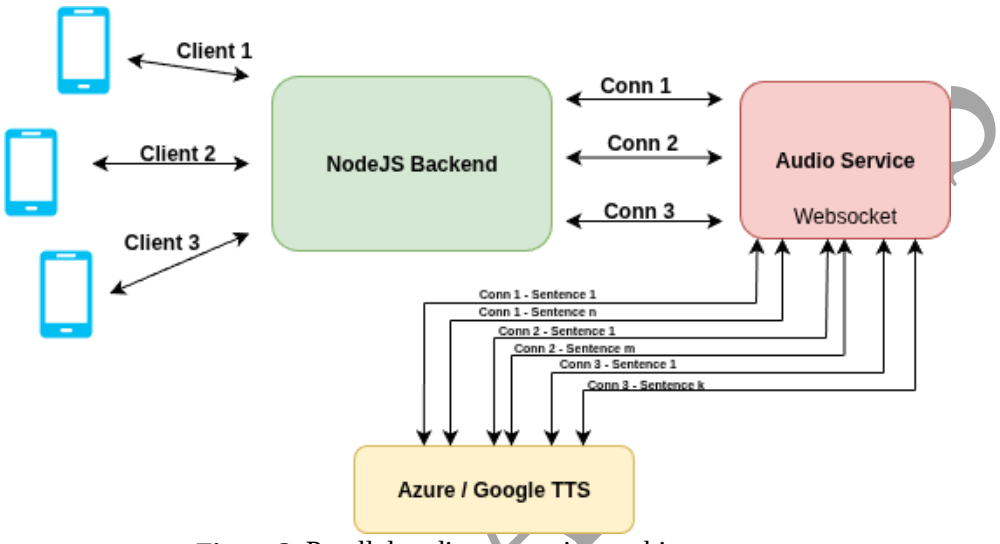

Figure 2. Parallel audio processing architecture

So the normal way of communication is the following: the user asks something from the application. The application communicates with the backend, prepares and makes a request towards OpenAI. OpenAI gives responses (token frames) as streaming mode, the backend collects sentences from these. As one sentence is identified and ready, it is immediately sent to the audio server for processing and to the client mobile app, while the remaining part of the OpenAI streaming is continued to be parsed. The end of the OpenAI response is identified by the end token signal (`data: [DONE]`).

During this process the backend listens to answers from the audio server as well in order to apply true parallelism. Because if the OpenAI answer is long for the user question, and the first sentence was short, it is not uncommon that the audio of the first sentence is already ready while OpenAI still sends the remaining text part to the backend. So we can only reach a good user experience, if the backend immediately sends the ready audio parts back to the client while the remaining will come later.

Why collecting sentences is a good idea: we need to provide some basic unit for the TTS services which is large enough to convert and small enough to not have “long” conversion time at Azure/Google.

5.2.1 Serving multiple client request parallel via websocket

One of the key problems was to eliminate every bottleneck from the whole process and keep implementation as simple as possible. To achieve maximum performance,

we must think in terms of threads. Since the input data arriving at the audio service always comes as separate sentences—each representing an independent processing unit for the TTS providers—this naturally forms the basis for dividing the workload into processing units.

WebSocket server implementations generally operate in a very similar manner, offering a few functions to the programmer for handling the socket. These typically include: *onStart*, *onOpen*, *onClose*, *onMessage*, and *onError*. Nevertheless there are *synchronous* and *asynchronous* WebSocket implementations. In a synchronous WebSocket implementation, the communication flow is blocking. When a message is received or sent, the thread waits until the operation is complete. In an asynchronous implementation, messages are sent and received using non-blocking, event-driven mechanisms. The server can perform other tasks while waiting for I/O operations to complete.

The first identified bottleneck was the backend side, because Node.js is traditionally known for its single-threaded, event-driven architecture, built on top of the *V8 JavaScript engine* and the *libuv* library [8]. This design excels at handling I/O-bound tasks using non-blocking asynchronous operations. However, CPU-bound operations can block the main event loop, degrading overall system responsiveness. So when many users communicate with the AI assistant there is only one channel towards OpenAI and the Audio service.

To address this limitation, Node.js introduced the *worker_threads* module in version 10.5.0 (and stabilized in 12.x), enabling developers to offload CPU-intensive or blocking operations to separate threads.

How worker threads are used in our environment: Each user request, which needs OpenAI answer, is handled by worker threads. Every user request opens a new websocket connection to the Audio service for sending the sentences. When the sentences are ready, the backend closes the websocket connection. So many channels can be open towards the Audio Service, which has a huge positive performance impact. And besides, this approach helps to keep the Audio Server implementation more simple, because all the sentences belonging to one user request which needs to be converted to audio are handled within one separated websocket connection. This makes it easier to handle threads and preserve proper audio parts order.

5.2.2 Parallel Audio Processing

Our parallel audio processing implementation does not use any library for handling or scheduling threads. Everything is built from scratch. The websocket server works like a Singleton. Even though every user request has a unique websocket connection from the backend, the server has only one main controller class. This means that any array/structure is needed for handling audio processing threads to be a global array. We introduced a global Vector array for incoming *TextStream*s. *TextStream* class is a structure, which holds all the convertible text input parts (sentences) from a unique user. It has a unique *clientID* which identifies the client. So if many users have a conversation with the AI assistant, there are multiple connections from the backend to the audio service. Sentences are coming continuously from the backend because of the OpenAI streaming mode and based on the clientID, the websocket server knows that the newly incoming text belongs to which structure in the global array.

When a message arrives to the websocket server, the server creates a new thread in order to make a non-blocking environment and allow the server to handle more messages. This thread manages the audio conversion process. The audio conversion starts with a text filtering process and after that the preferred TTS service provider is chosen based on the language. Our implementation supports English and Indonesian language using Google TTS for Indonesian language and Azure TTS for English conversion.

By assigning each sentence to a dedicated thread, the system allows multiple sentences to be processed concurrently. This design leverages cloud concurrency limits efficiently and ensures maximum throughput.

5.2.3 Audio Ordering and Delivery

Our goal was to improve user experience as much as possible. For example, the easiest way would be to wait for all the text input from OpenAI and convert them sentence by sentence and send them back to the client. But with this approach there will be a longer delay between the arriving text and the arriving audio on the user side. So, a more appropriate, more satisfactory approach is, to send back immediately the audio part when it is ready. However, this resulted in an additional problem: ensuring correct audio order which comes from the asynchronous nature of audio TTS processing.

Example: there is only one user, which asks something more complicated from the AI assistant and the test result from OpenAI has multiple sentences. Longer sentences may take more time to synthesize than a shorter one, even if it was sent earlier. In this case, we cannot send back the finished audio immediately, because the finished sentence is not the next in order.

In our implementation, each *TextStream* (identified by *clientID*) structure has a text part array representing the sentences. The input text elements in this array are in order, because as they come from the backend, they are added to the end of this array. In order to keep proper audio ordering in the sending back process, we need a special algorithm to check if there is any part ready to send back.

In order to control the proper audio ordered communication, we have two options:

1) Introduce some kind of manager thread globally, which always checks the sendable audio parts. It runs in short time periods. Thread safety is very important in this solution, because of getting status information from other running threads.

2) Build control logic into the working threads: every audio processing thread has a built in option for checking audio ready parts. The algorithm reads the array of the input texts (which is already in order because it is coming from the backend) and checks if there is anything to send back to the backend.

In our implementation the second option was chosen. The algorithm to determine to send anything back is the following:

```
Function isAnythingToSend() -> TextPart or null
    Lock this function to ensure thread safety (synchronized)
```

```
    Set minimumNotSentIndex to a very large number (e.g., 99999)

    For each index i from 0 to size of textParts - 1:
        If textParts[i] is not already sent:
            If i < minimumNotSentIndex:
                Set minimumNotSentIndex = i

    Set audioReadyIndex to a very large number (e.g., 89999)

    For each index i from 0 to size of textParts - 1:
        If textParts[i] is audio ready AND not already sent:
            If i < audioReadyIndex:
                Set audioReadyIndex = i

    If minimumNotSentIndex == audioReadyIndex:
        Return textParts[audioReadyIndex]

    Return null
```

In this algorithm we look for the text part, where the audio is ready, it is not sent already and it is the next in the order. By this method the problem is not solved fully, because if the sendable audio part is not itself (the same thread), then the last audio part will not be sent to the backend. Therefore an extension is needed, which makes possible to handle the audio ordering logic properly:

```
// Websocket server onMessage function
Function onMessage(connection, message):

    // Parse incoming message
    ParseMessage(message)

    // Retrieve or create a text stream instance for this client
    textStream ← addToGlobalStreamArray(message.clientId)

    // Start a new background thread to process the request
    Start new Thread ← run() {

     // Processing request: filtering, send to TTS providers
    processor.processRequest(textStream.latestSentence)

        // Loop to send audio responses back in order
        Repeat:
            partToSend ← textStream.isAnythingToSend()

            If partToSend is not null:
                Print "Sending audio part [index] to client [clientId]"
                connection.send(partToSend.responseData)
                partToSend.markAsSent()
                partToSend.clearMemory()
        Until partToSend is null
  }

End function
```

Because the socket server has a global array for tasks, the *addToGlobalStreamArray()* function has two functions: if the message is coming

from a client (clientID) that is not exist in the array, then it creates a new *TextStream* class for the client and add to the end of the array. If a *TextStream* with the clientID already existed in the array, then it gives the reference of this class back. It means that a new sentence has arrived for an existing client connection. This method ensures that the request from the same client uses the same TextStream class holding all the user related OpenAI answer sentences in one place.

The last infinite loop is the heart of the audio ordering preserve and sending process. The loop is inevitable because even if the socket server gets the sentences from backend in a proper order, the audio conversion time can be different. Therefore, there may be cases where a later audio conversion in a queue is completed sooner than a sentence that arrived earlier from the backend. Let's take the following case: we have three sentences (from one client). The first one is the longest and the second one is shorter than the third. These three sentences mean three threads in the audio server. During the conversion the second sentence will be ready at first. The *isAnythingToSend()* function in the above algorithm will give back a NULL response, because we cannot send back the second sentence, while the first one is not ready yet. So the second thread is terminated. Let's assume that the audio conversion of the first sentence is finished. Now we have two sentences (1 and 2) ready. The infinite loop helps to send both sentences back, because the *isAnythingToSend()* function will give back the first sentence id (because it is ready now), which can be sent back. But because of the loop, the isAnythingToSend*()* function runs again and identifies that the second is also ready, so it can be also sent back. After sending back the two audio parts, the first thread terminates. Finally when the third part is ready, the *isAnythingToSend()* function will give back the third part ID, which is sendable immediately and thread terminates.

5.2.4 TTS provider limitations

Cloud-based Text-to-Speech (TTS) services, such as those provided by Microsoft Azure Cognitive Services and Google Cloud Text-to-Speech, offer high-quality, multilingual, neural voice synthesis capabilities. These platforms enable developers to convert textual information into natural-sounding speech with minimal local computation. However, in real-time or high-throughput applications—such as chatbots responding with audio in parallel—these services impose rate limits and architectural constraints that must be carefully managed.

In our case we had a 100 request / minute / project rate limit on both providers. It is obvious that if 50 users are using the application simultaneously and every user has at least 2 sentences answer from the OpenAI, then we are in trouble, the rate limit was reached easily. To overcome this the following trick was made.

Both providers allow to create more projects and limitations are regarded to projects. Based on this, the idea was to make several projects on both providers. Each project is identified by an unique API key. If we make a simple algorithm, which rotates the API keys during audio conversion, it will give enough time between the requests to not run into rate limitations. Of course this method is not enough to handle thousands of users simultaneously, that needs unique pricing from providers.

In our implementation we used a simple “round-robin” like algorithm to rotate between keys. Every TTS service request is performed with another key. 32 projects were created on both providers. If we rotate keys between them, it pushes

the rate limit (32*100 = 3200 simultaneous requests) far enough to make a usable mobile application. Maybe a more intelligent solution can be the following: if we count the request numbers on each project / minute, then we know when we reach the rate limit. A smart algorithm can predict the usage before reaching the rate limit and can delay the audio conversion in a given little amount of time to not reach the limit. Based on the current number of simultaneous users the algorithm could predict how much time delay it should insert before every TTS request.

6. Performance Evaluation

To assess the efficiency, responsiveness, and scalability of our mobile application's backend architecture, we conducted a series of performance evaluations focusing on the **OpenAI streaming pipeline**, the **multithreaded audio generation service**, and **end-to-end latency** experienced by users. The primary goal was to validate that the system supports **real-time, parallel audio synthesis** across multiple concurrent user sessions, while maintaining low latency and ordered delivery of audio responses.

Properly testing such a complex system is not an easy task. As a first step, a Python based, multithreaded backed service test environment was built. The supported main functionalities of the environment are:

- Configurable multithreaded requests: simulate parallel mobile app usage simulation by supporting parallelized requests to the backend. The number of simultaneous requests are configurable to help making wider test cases
- Monitoring outgoing and incoming websocket messages: manage requests, pairing and validate messages, count and alert on missing parts
- Measure different types of requests times: Summarize and visualize results.

The tests were conducted on a Google Cloud-based machine running Debian Linux, equipped with a 4-core CPU and 8 GB of RAM.

6.1. First approach

Due to the complexity of the problem, the research progressed step by step. Initially, we aimed for a working but as simple a solution as possible, with the primary goal of gaining experience. The first, less efficient approach included the following architecture and characteristics:

Backend side: In the initial implementation, user requests coming from the mobile app and are going towards OpenAI. The streaming mode responses returned from OpenAI were immediately sent back to the mobile application via a WebSocket connection. During this time, in the audio domain, however, the backend waits for all parts of the response to arrive from OpenAI, and then sends the complete response over a WebSocket to the Java-based audio service. In this solution, there was only a single permanent WebSocket connection between the backend and the audio service.

Multithreaded Audio Service: The backend sends the full text, meaning it contains all the sentences intended as a response to the audio server. The main task

of the audio service was to split and filter the received text into individual sentences. These sentences were then sent to the TTS (Text-to-Speech) providers simultaneously. Each sentence was converted into a separate thread. Once all the audio files are ready controlled by our special algorithm (5.2.3), the service sends the result back to the backend and the backend sends the audio as whole back to the user.

The test results obtained using the above solution are as follows:

Table 1. First results with 50 simulated users. Time was measured in seconds.

Statistics	Min	Max	Average
First message received duration	6.95	71.44	35.84
Last message received duration	7.71	71.67	36.14
First audio received duration	13.23	74.52	41.48
Last audio received duration	13.39	74.65	41.57

In the measurements, we recorded four cumulative data points: the average time of arrival for the first and last messages from OpenAI, and the average time required for the audio conversion of the first and last sentences. The results clearly show that performance is unconvincing, making the service nearly unusable even with just 50 users.

6.2. Second approach

In the second approach, our goal was to eliminate the bottleneck occurring with responses from OpenAI. While the previous solution waited for all incoming sentences before sending them in a batch for audio conversion, this version allows the backend to send the data to the audio server sentence by sentence. As the OpenAI streaming response continuously arrives, the backend sends the chunks immediately back to the client in order to have better user experience. But meanwhile, the backend forms sentences from the incoming text chunks, and these sentences are sent immediately to the audio service via a WebSocket channel. It is important to emphasize that, in this case, there is still only a single connection between the backend and the audio server.

The test results achieved with this approach are as follows:

Table 2. Second results with 50 simulated users. Time was measured in seconds.

Statistics	Min	Max	Average

First message received duration	1.84	30.59	12.26
Last message received duration	2.57	31.02	12.54
First audio received duration	2.42	31.04	15.72
Last audio received duration	3.62	31.20	16.04

Although the results have been visibly improved and the application may have become more usable, unfortunately, the performance is still not sufficient to ensure an optimal user experience.

6.3. The final approach

With the experience and measurements gained from the previously developed models, we were able to eliminate all bottlenecks. Based on this, the architecture presented in Section 5.2.1 was developed, according to which, for *n* number of users, *n* number of WebSocket connections are established between the backend and the audio server. On the backend side, worker threads support NodeJS parallelism, while on the Java audio side, the structure described in Section 5.2.3 was implemented. In the current implementation, the performance improvement is primarily expected due to the advantages arising from multiple WebSocket connections. So the main new feature regarding the previous second approach is, that there are many numbers of websocket connections between the backend and the audio server. This number equals the number of clients who are using this part of the application. This modification removed the last bottleneck between the backend and the audio server.

Table 3. Third results with 50 simulated users. Time was measured in seconds.

Statistics	Min	Max	Average
First message received duration	0.80	1.56	1.14
Last message received duration	1.72	2.71	2.13
First audio received duration	1.37	2.30	1.74
Last audio received duration	2.17	3.47	2.68

It is clearly visible that the results far exceed those of previous attempts. Naturally, the architecture can be further scaled if needed. However, they also demonstrate that without proper parallelization, the results will not be satisfactory. With

parallelization, however, the system's complexity increases significantly. Tests were also conducted with 100-300 users, where naturally the numbers were somewhat higher. Nevertheless, the tests clearly show that a 4-core computer does not constitute a bottleneck, as the CPU load on the cores is not significant since most of the work is handled by the TTS providers.

7. Conclusion

In this paper, we presented the design and implementation of a mobile application that combines natural language understanding and speech synthesis to provide an interactive, multimodal user experience. The architecture leverages OpenAI's streaming API to deliver real-time, sentence-by-sentence text generation, and integrates a custom-built, multithreaded audio service that uses external TTS capabilities to generate speech output in parallel. A key technical contribution of our solution lies in its ability to perform true parallelism through the use of dedicated WebSocket channels for each user session. This design enables concurrent and scalable audio processing, avoiding the limitations of single-threaded execution environments such as Node.js. Performance evaluations demonstrated that the architecture achieves low-latency response times, effective concurrency management, and high audio output consistency, even under increased user load. This work highlights the viability of integrating advanced AI services in mobile applications to deliver fluid and personalized user interactions.

References

[1] Shen, J., Pang, R., Weiss, R. J., et al., Natural TTS synthesis by conditioning WaveNet on mel spectrogram predictions. ICASSP, 2018
[2] Tan, X., Ren, Y., He, D., Qin, T., Zhao, Z., & Liu, T. Y., FastSpeech 2: Fast and high-quality end-to-end text to speech. Proceedings of ICLR, 2021
[3] Schneps, M. H., Thomson, J. M., & Sonnert, G., E-books and the visual advantage for struggling readers. PLOS ONE, 2013
[4] Sánchez, J., & Aguayo, F., Mobile learning for visually impaired people. Proceedings of the 4th International Conference on Universal Access in Human-Computer Interaction, 2006
[5] Suh, S., Hwang, J., et al., Serving LLMs at scale: Streaming, caching, and system optimizations. arXiv preprint arXiv:2106.05350, 2021
[6] Kleppmann, M., Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media., 2017
[7] OpenAI., Streaming API Documentation. https://platform.openai.com/docs/guides/text-generation/streaming, 2023
[8] Tilkov, S., The Node.js Event Loop: Explained., https://nodesource.com/blog/the-node-js-event-loop-explained, 2019
[9] IETF. The WebSocket Protocol (RFC 6455)., https://datatracker.ietf.org/doc/html/rfc6455, 2011
[10] Google Cloud, Cloud Text-to-Speech Documentation, https://cloud.google.com/text-to-speech/docs, 2025
[11] Deshpande, K., Jain, A., Abhinav, Mishra, S., Goel, S., Garg, R., Empowering Real-Time Communication: A Seamless Chatting System Using Websocket. In: Hassanien, A.E., Anand, S., Jaiswal, A., Kumar, P. (eds) Innovative Computing and Communications. ICICC 2024. Lecture Notes in Networks and Systems, vol 1039. Springer, Singapore. 2025, https://doi.org/10.1007/978-981-97-4152-6_29

[12] Sathishkumar M, Mr Karthikeyan S, REAL TIME CHAT APPLICATION AWS WEBSOCKET API, International Journal on Science and Technology, Volume 16, Issue 2, 2025, https://doi.org/10.71097/IJSAT.v16.i2.6306
[13] K.E. Ogundeyi C Yinka-Banjo, WebSocket in real time application, Nigerian Journal of Technology, Vol. 38 No. 4, 2019, https://doi.org/10.4314/njt.v38i4.26