SingSphere

Online Real-Time Streaming Karaoke Platform

Yin Tung Chen (yc4377) *Columbia University*
Hsin-Yuan Huang (hh3017) *Columbia University*
Chiao-Fen Chan (cc4799) *Columbia University*
Chia-Mei Liu (cl4424) *Columbia University*

***Abstract*—This paper presents SingSphere, an innovative online karaoke platform that integrates advanced cloud services to deliver a seamless and interactive singing experience. By leveraging AWS Lambda, EC2, OpenSearch, and RabbitMQ, SingSphere offers a scalable and responsive solution that transcends traditional karaoke systems. The platform focuses on social interaction, allowing users to sing synchronously with friends across the globe. We discuss the system architecture, design considerations, and the results from extensive latency performance tests. Our findings illustrate the challenges and implications of real-time audio synchronization in a distributed environment. SingSphere's approach to handling these challenges showcases the potential of cloud-based solutions in real-time entertainment applications.**

I. Introduction

Karaoke, blending singing with social interaction, is a universally popular form of entertainment. However, current digital platforms predominantly focus on the singing aspect, offering features like solo and duet recordings, music editing tools, and discussion forums, which overlook the core element of karaoke: the live, interactive experience. This gap is acutely felt among friends who are physically apart and unable to share the same physical space in a karaoke setting. Our project, SingSphere, aims to bridge this gap by enabling users to engage in a virtual karaoke environment that mirrors the experience of a physical karaoke room, placing a stronger emphasis on the social aspects of karaoke.

In an increasingly connected world, geographical separation is a common reality for many, especially due to professional and educational pursuits. This separation creates a demand for platforms that facilitate not just communication, but shared experiences. Our motivation for SingSphere stems from our own experiences of relocating and maintaining friendships across distances. The application is envisioned as a solution for individuals seeking to sustain and enrich their social connections through a shared love for music and entertainment.

While the market offers a variety of karaoke applications such as Smule, Starmaker, and SingSnap, they primarily focus on singing-related functionalities and do not fully encapsulate the social and interactive essence of karaoke. SingSphere is designed to transcend these limitations by incorporating advanced features that foster real-time social interactions. By leveraging AWS technology, we aim to create a more holistic karaoke experience, integrating aspects of gaming and social media, thereby offering a platform that is not just about singing, but about creating memorable social experiences.

SingSphere is tailored for individuals separated by distance, such as long-distance friends and family, who seek engaging ways to reconnect, as well as karaoke enthusiasts who desire a platform for showcasing their talent and engaging with fellow singers. The application also appeals to young adults and millennials, a demographic that is both tech-savvy and in constant search of innovative online entertainment avenues. This introduction outlines the concept and motivation behind SingSphere, setting the stage for a deeper exploration of its technical architecture, design considerations, and user experience in the subsequent sections of the paper.

II. Architecture

A. Component Description

CloudFront. AWS's content delivery network service that serves the frontend application with low latency and high transfer speeds.

S3 Frontend. Amazon Simple Storage Service (S3) bucket that hosts static web assets for the frontend, such as HTML, CSS, JavaScript files for the React.js-based user interface.

API Gateway. Acts as a managed gateway for API calls, routing requests from the frontend to the appropriate Lambda function, and provides an HTTPS endpoint for the frontend.

Lambda Functions.

- index-music. Takes song data for new uploads and indexes it in the OpenSearch service.
- search-music. Receives search requests from the API Gateway and queries the OpenSearch service to return song data.
- enqueue-music. Handles the enqueueing of song requests, interfacing with RabbitMQ to add songs to the queue for a specific room.

RabbitMQ. Message broker service that manages communication between services. It queues song requests and serves as a buffer to handle load and ensure message delivery.

S3 Music Bucket. Stores the MP3 files of songs and is queried when a user uploads a new song or when the Media Manager needs to access a song for streaming.

OpenSearch. Managed search service powered by OpenSearch, stores and indexes song metadata for efficient search and retrieval.

EC2 Instance. Virtual servers in the AWS cloud. Here, they host the Media Manager Room Manager and Streaming Server components and mount S3 Music Bucket component.

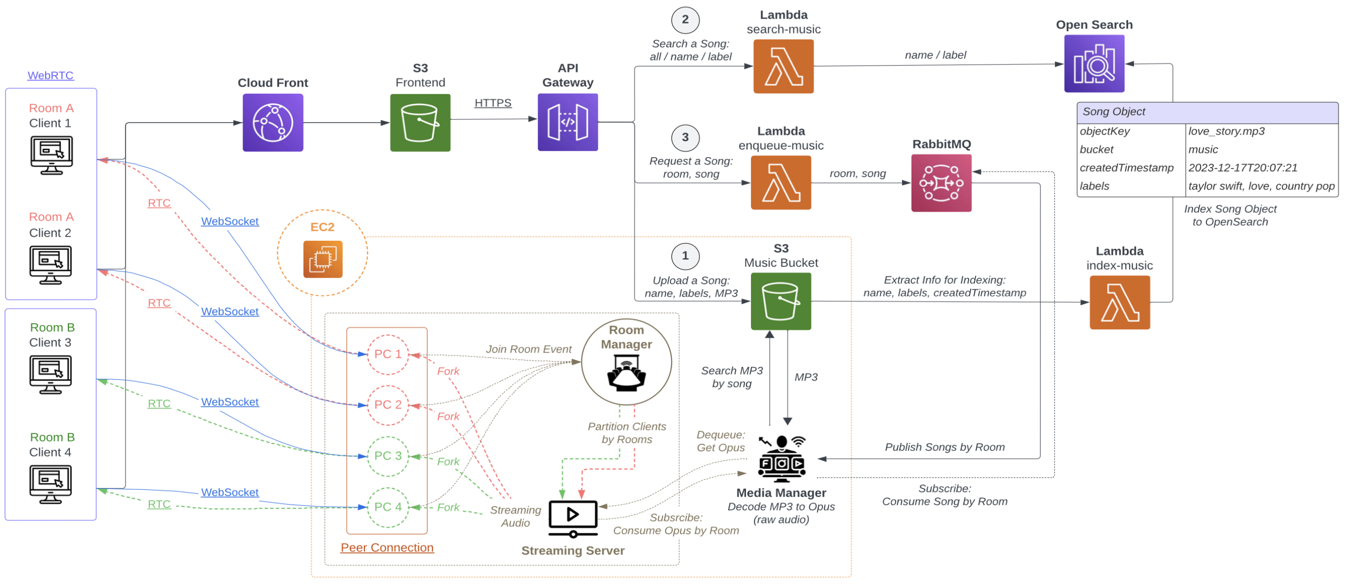


Fig. 1. Design Architecture

Media Manager. Decodes the MP3 files from the S3 Music Bucket into a streaming-friendly format and sends them to the Streaming Server.

Room Manager. A backend service running on EC2 that handles room-specific actions, such as managing user sessions and partitioning clients into rooms.

Streaming Server. Responsible for streaming audio content to clients. It interfaces with the Media Manager to receive the streaming data.

B. Protocols Description

HTTPS. The API Gateway acts as the interface for all HTTPS requests, routing actions to the appropriate Lambda functions for processing, ensuring secure communication between the client and server.

WebRTC. The Streaming Server leverages WebRTC protocols for direct peer-to-peer media streaming, enabling real-time audio exchanges with low latency, which is crucial for the karaoke experience.

WebSocket. The Streaming Server and Room Manager utilize WebSockets to handle real-time events that require persistent connections, such as signaling for WebRTC and user interaction within rooms.

AMQP. The Media Manager uses the AMQP protocol to interact with RabbitMQ, managing song requests and processing for a scalable and reliable message queuing system.

File System. The Media Manager handles file operations, reading, and writing media files to and from the S3 drive, which is crucial for managing the media that is streamed to users.

C. Data Flow

Uploading a Song

1) A user uploads a song through the frontend, which sends the file and metadata to the S3 Music Bucket via the API Gateway.
2) The upload triggers the index-music Lambda function, which processes the song's metadata.
3) $index-music$ indexes the new song in the OpenSearch service for future searches.

Searching for a Song

1) The user utilizes the frontend interface to issue a search command.
2) The frontend sends the search query to the API Gateway.
3) The API Gateway invokes the search-music Lambda function, which queries the OpenSearch service.
4) The song results are returned to the frontend for the user to select.

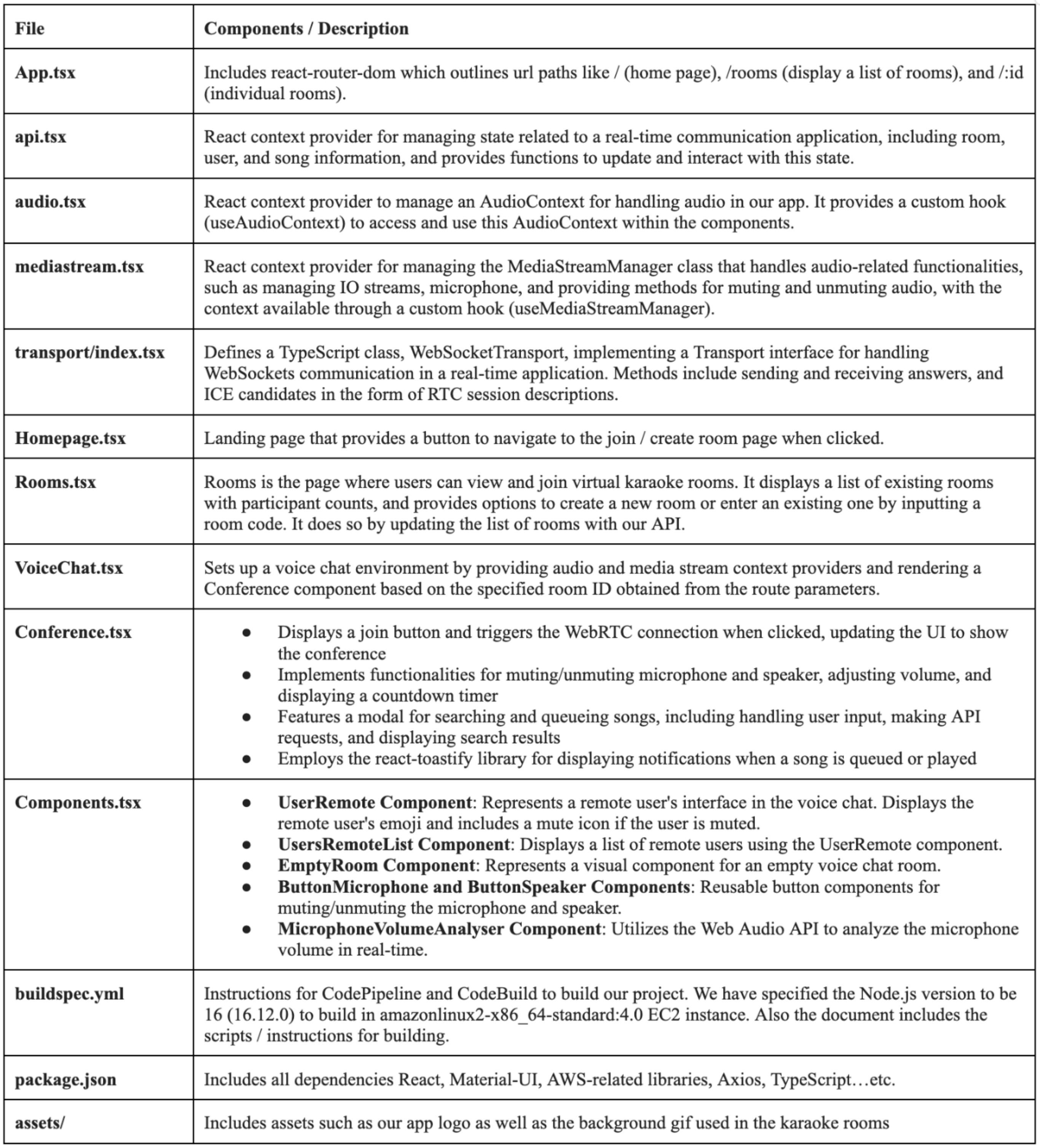
Requesting a Song

1) Once a song is selected, the frontend sends a request to the API Gateway to queue the song.
2) The API Gateway triggers the enqueue-music Lambda function.
3) $enqueue-music$ sends a message to RabbitMQ to add the song to the queue for the user's room.
4) The Room Manager monitors RabbitMQ and manages which songs are played in each room.

Playing a Song

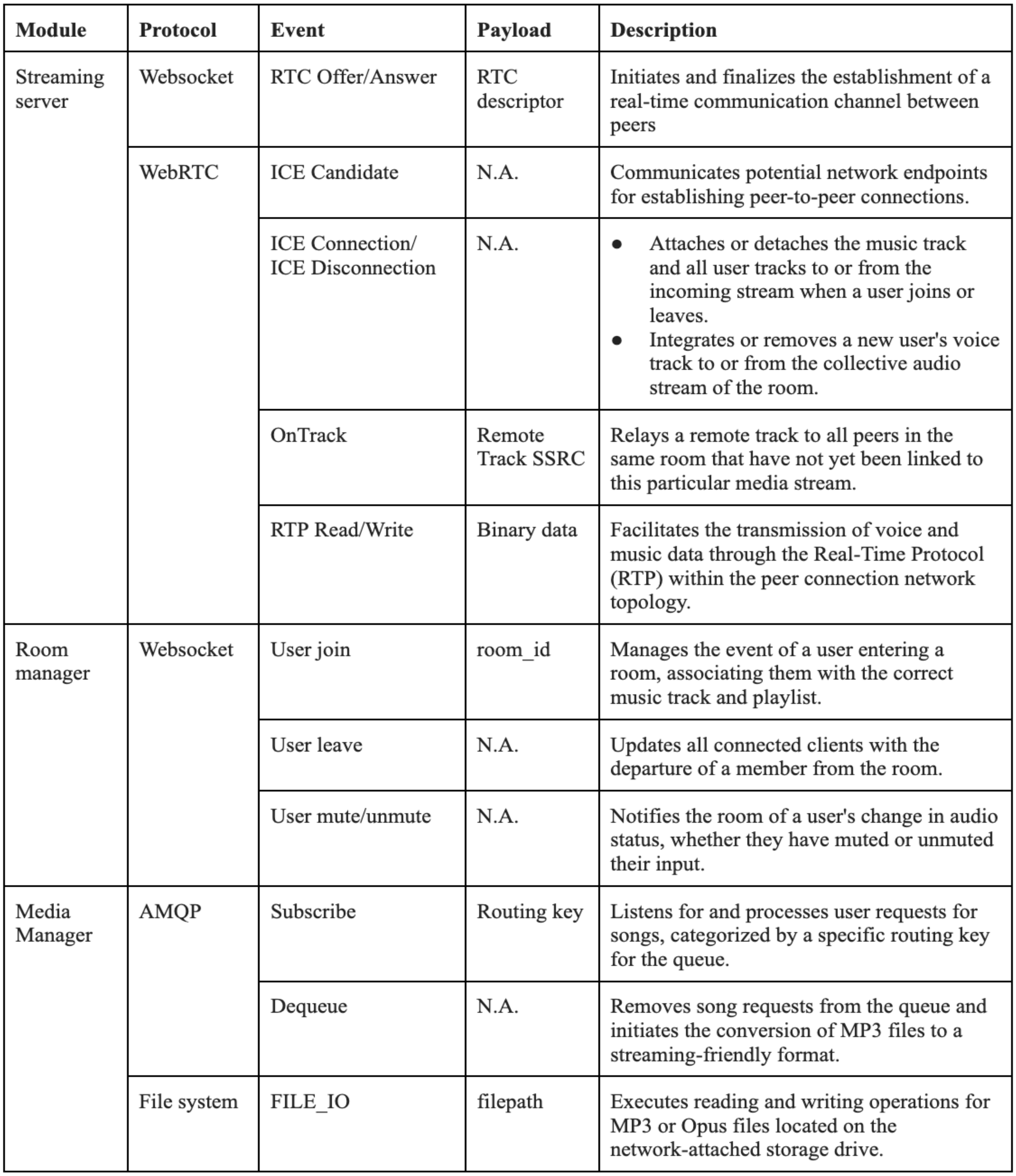
1) When a song is next in the queue, the Media Manager retrieves the MP3 file from the S3 Music Bucket.
2) The Media Manager decodes the MP3 to opus, a streaming format, and sends it to the Streaming Server.
3) The Streaming Server streams the audio to clients in the room via WebRTC connections.

Design Philosophy The design philosophy of SingSphere is rooted in creating a seamless and interactive online karaoke ex-



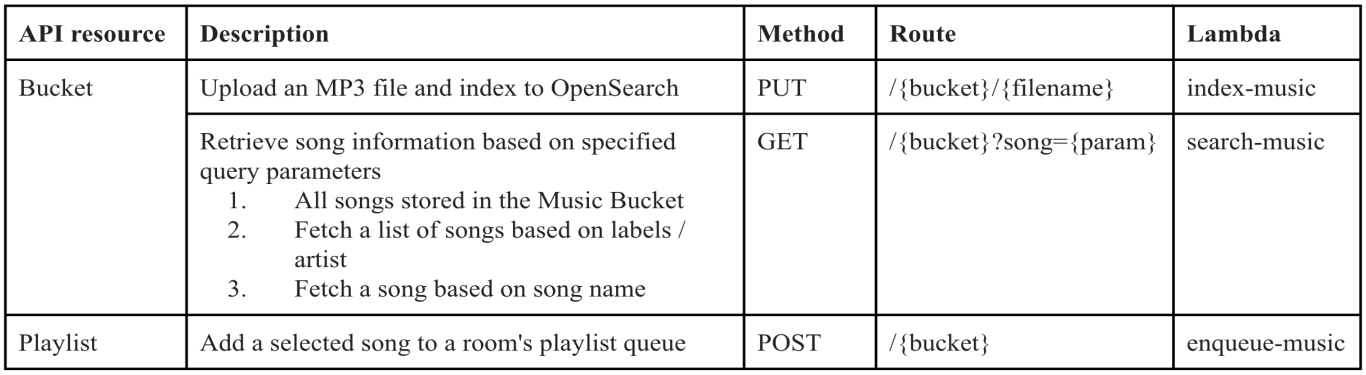
File	**Components / Description**
App.tsx	Includes react-router-dom which outlines url paths like / (home page), /rooms (display a list of rooms), and /:id (individual rooms).
api.tsx	React context provider for managing state related to a real-time communication application, including room, user, and song information, and provides functions to update and interact with this state.
audio.tsx	React context provider to manage an AudioContext for handling audio in our app. It provides a custom hook (useAudioContext) to access and use this AudioContext within the components.
mediastream.tsx	React context provider for managing the MediaStreamManager class that handles audio-related functionalities, such as managing IO streams, microphone, and providing methods for muting and unmuting audio, with the context available through a custom hook (useMediaStreamManager).
transport/index.tsx	Defines a TypeScript class, WebSocketTransport, implementing a Transport interface for handling WebSockets communication in a real-time application. Methods include sending and receiving answers, and ICE candidates in the form of RTC session descriptions.
Homepage.tsx	Landing page that provides a button to navigate to the join / create room page when clicked.
Rooms.tsx	Rooms is the page where users can view and join virtual karaoke rooms. It displays a list of existing rooms with participant counts, and provides options to create a new room or enter an existing one by inputting a room code. It does so by updating the list of rooms with our API.
VoiceChat.tsx	Sets up a voice chat environment by providing audio and media stream context providers and rendering a Conference component based on the specified room ID obtained from the route parameters.
Conference.tsx	● Displays a join button and triggers the WebRTC connection when clicked, updating the UI to show the conference ● Implements functionalities for muting/unmuting microphone and speaker, adjusting volume, and displaying a countdown timer ● Features a modal for searching and queueing songs, including handling user input, making API requests, and displaying search results ● Employs the react-toastify library for displaying notifications when a song is queued or played
Components.tsx	● **UserRemote Component**: Represents a remote user's interface in the voice chat. Displays the remote user's emoji and includes a mute icon if the user is muted. ● **UsersRemoteList Component**: Displays a list of remote users using the UserRemote component. ● **EmptyRoom Component**: Represents a visual component for an empty voice chat room. ● **ButtonMicrophone and ButtonSpeaker Components**: Reusable button components for muting/unmuting the microphone and speaker. ● **MicrophoneVolumeAnalyser Component**: Utilizes the Web Audio API to analyze the microphone volume in real-time.
buildspec.yml	Instructions for CodePipeline and CodeBuild to build our project. We have specified the Node.js version to be 16 (16.12.0) to build in amazonlinux2-x86_64-standard:4.0 EC2 instance. Also the document includes the scripts / instructions for building.
package.json	Includes all dependencies React, Material-UI, AWS-related libraries, Axios, TypeScript…etc.
assets/	Includes assets such as our app logo as well as the background gif used in the karaoke rooms

Fig. 2. Frontend Structure



Module	Protocol	Event	Payload	Description
Streaming server	Websocket	RTC Offer/Answer	RTC descriptor	Initiates and finalizes the establishment of a real-time communication channel between peers
	WebRTC	ICE Candidate	N.A.	Communicates potential network endpoints for establishing peer-to-peer connections.
		ICE Connection/ ICE Disconnection	N.A.	● Attaches or detaches the music track and all user tracks to or from the incoming stream when a user joins or leaves. ● Integrates or removes a new user's voice track to or from the collective audio stream of the room.
		OnTrack	Remote Track SSRC	Relays a remote track to all peers in the same room that have not yet been linked to this particular media stream.
		RTP Read/Write	Binary data	Facilitates the transmission of voice and music data through the Real-Time Protocol (RTP) within the peer connection network topology.
Room manager	Websocket	User join	room_id	Manages the event of a user entering a room, associating them with the correct music track and playlist.
		User leave	N.A.	Updates all connected clients with the departure of a member from the room.
		User mute/unmute	N.A.	Notifies the room of a user's change in audio status, whether they have muted or unmuted their input.
Media Manager	AMQP	Subscribe	Routing key	Listens for and processes user requests for songs, categorized by a specific routing key for the queue.
		Dequeue	N.A.	Removes song requests from the queue and initiates the conversion of MP3 files to a streaming-friendly format.
	File system	FILE_IO	filepath	Executes reading and writing operations for MP3 or Opus files located on the network-attached storage drive.

Fig. 3. Backend Voice Server APIs



API resource	Description	Method	Route	Lambda
Bucket	Upload an MP3 file and index to OpenSearch	PUT	/{bucket}/{filename}	index-music
	Retrieve song information based on specified query parameters 1. All songs stored in the Music Bucket 2. Fetch a list of songs based on labels / artist 3. Fetch a song based on song name	GET	/{bucket}?song={param}	search-music
Playlist	Add a selected song to a room's playlist queue	POST	/{bucket}	enqueue-music

Fig. 4. Backend REST APIs

perience that closely mimics the social dynamics of in-person karaoke sessions. To achieve this, we emphasized real-time data transmission, low-latency media streaming, and robust user interaction capabilities. The architecture is designed to be scalable, to accommodate a growing user base and song library, and modular, to facilitate maintenance and the potential integration of new features. Scalability is ensured through stateless design patterns and the use of message queues, while modularity is achieved through microservices architecture, where each service is responsible for a discrete function of the application.

Technology Choices. The selection of AWS services, React.js, Go, and Python was made to leverage the strengths of each to fulfill our design philosophy. AWS provides a comprehensive suite of scalable cloud services that support the deployment and runtime of the application. React.js was chosen for the frontend due to its efficient update and rendering capabilities, enabling dynamic user interfaces that respond quickly to user interactions. Go's performance in handling concurrent operations makes it ideal for the Room Manager, which requires high throughput for real-time communication. Python's rich set of libraries and its simplicity make it perfect for writing AWS Lambda functions for backend processes like searching and indexing songs.

User Experience. User interface and interaction are central to SingSphere, where the goal is to create an engaging and intuitive user experience. The interface design is clean and straightforward, minimizing cognitive load and allowing users to focus on the core functionality of singing and social interaction. Accessibility considerations are integrated, such as clear labels and voice navigation, to ensure that the platform is inclusive. Interaction design follows the principle of immediate feedback, ensuring that any action taken by a user is met with a swift and clear response from the system, whether it's searching for a song or interacting with other users in a karaoke room.

Security and Privacy. Data security and user privacy are paramount in SingSphere's design. All communications are secured using HTTPS, ensuring that data in transit is encrypted. For data at rest, we use AWS's robust security protocols for S3 buckets and databases, including encryption and access control measures. User data is handled in compliance with GDPR and other relevant privacy regulations, ensuring that personal information is processed transparently and with user consent. The application implements best practices such as regular security audits, minimal data retention policies, and secure credential storage to protect user information and maintain privacy.

Please refer to Figure 2, 3, and 4 for further details about code details.

III. Results

In order to assess the performance and reliability of our streaming system, we conducted a controlled experiment with the objective of evaluating system robustness, identifying latency bottlenecks, and analyzing the impact of different variables on streaming quality.

The experiment was conducted in a series of steps designed to measure the latency from the speaker's output to the listener's perception under various conditions. Here's how we proceeded:

1) Initiation: Testers began a timer at the moment of speaking into the system, marking this as the start time (T1).
2) Observation: The tester listened to the audio streaming back from the peer device.
3) Closure: Upon hearing the playback of their own voice, testers recorded the time and stopped the timer, noting this as the stop time (T2).
4) Latency Calculation: The latency was calculated by the difference between T2 and T1.

To account for manual reaction delays in stopping the timer, we adjusted the recorded T2 by a standard reaction time of 0.2 seconds, providing a more accurate representation of actual latencies.

The test was executed under four distinct scenarios to simulate different usage situations and load conditions:

1) Single Speaker: A scenario with one person speaking.
2) Dual Speakers: A scenario simulating two people speaking simultaneously.

Voice Test	Average Latency (second)	Max latency (second)	Min Latency (second)
One speaker	0.585	0.81	0.75
Two speakers	1.0375	1.53	0.86
One speaker with music	1	1.42	1.03
Two speakers with music	1.2525	1.65	1.23

Fig. 5. Experiment Results

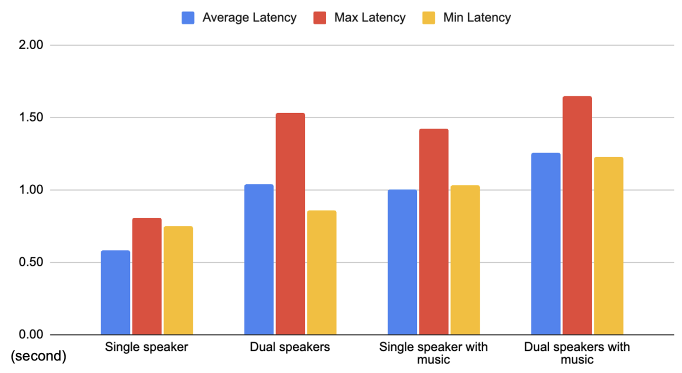


Fig. 6. Latency Profile by Scenario

3) Single Speaker with Music: A scenario with one person speaking while music played in the background.
4) Dual Speakers with Music: A scenario with two people speaking over the background music.

The collected data indicates that the introduction of additional speakers and background music increases the average, maximum, and minimum latency experienced in the streaming system. This suggests that while the system maintains robust performance under light loads, the complexity of audio processing and network handling scales with the number of simultaneous input sources. The additional latency in scenarios with music suggests that data throughput and processing time are potential areas for optimization.

We have identified a critical synchronization issue within our application, specifically pertaining to the alignment of voice and music streams during playback. The most prominent manifestation of this problem is that the voice consistently lags

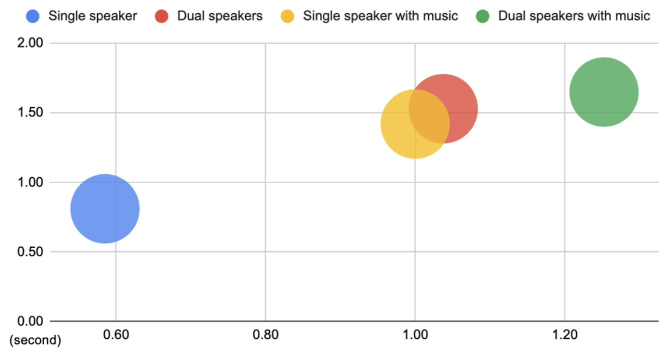


Fig. 7. Latency Distribution by Scenarios

behind the music. This delay not only disrupts the intended harmony between the vocal and musical elements but also detracts from the user experience, which is crucial for a karaoke application.

The root causes of this desynchronization can be attributed to three main factors:

- **Music Propagation Delay.** There is an inherent delay in the propagation of the music stream. This latency, likely caused by the initial processing and buffering of the music data, results in the music starting ahead of the voice. Since both streams should ideally commence in unison, this issue indicates a need for a more synchronized buffering strategy.
- **Voice Congestion.** We have observed instances of voice congestion, where the vocal data experiences bottlenecks, possibly due to inadequate handling of high-traffic conditions or suboptimal audio data compression. This congestion exacerbates the misalignment, as the voice stream is further delayed while the music stream continues to play.
- **Network Connection Quality.**Variations in network quality are also contributing to this problem. Fluctuations in bandwidth and latency can affect the timing with which both streams are transmitted and received. While the music stream may tolerate a degree of delay without noticeable effect, the real-time nature of the voice stream is much less forgiving, making any lag immediately apparent

In response to the identified synchronization challenges within our karaoke application, we are proposing a multi-faceted approach aimed at scaling and optimizing our system's infrastructure and services.

- **Addressing Voice Congestion.** To mitigate the issue of voice congestion, we are planning to implement an advanced load balancing strategy across multiple EC2 instances. The introduction of Kubernetes and Elastic Load Balancing (ELB) will distribute the traffic more efficiently, ensuring that no single server becomes a bottleneck and that voice data is handled with minimal delay.
- **System Scaling.** We believe vertical scaling of our EC2 instances, OpenSearch service, and RabbitMQ could provide immediate relief to our current challenges. By enhancing the computing resources available to these services, we can significantly reduce the latency experienced during peak usage. This will involve allocating additional CPU, memory, and network capacity to handle the increased load more effectively.
- **Exploring Synchronization Solutions** The pursuit of flawless online music and voice synchronization continues to be a complex issue within the industry. Recognizing the human ear's acute sensitivity to rhythmic discrepancies, we are exploring innovative synchronization techniques currently employed by platforms like Zoom, Google Meet, and Discord, which have demonstrated competency in managing real-time audio feeds.

- **Musical Audio Sensitivity.** Our research acknowledges that users are particularly sensitive to timing issues in musical contexts. As such, our solution will not only address the technical aspects of data transmission but also the perceptual components that influence user experience. By studying the synchronization strategies of successful applications, we aim to develop an approach that is acutely attuned to the nuances of musical rhythm and the user's auditory expectations.
- **System Scaling.** We believe vertical scaling of our EC2 instances, OpenSearch service, and RabbitMQ could provide immediate relief to our current challenges. By enhancing the computing resources available to these services, we can significantly reduce the latency experienced during peak usage. This will involve allocating additional CPU, memory, and network capacity to handle the increased load more effectively.
- **Exploring Synchronization Solutions.**The pursuit of flawless online music and voice synchronization continues to be a complex issue within the industry. Recognizing the human ear's acute sensitivity to rhythmic discrepancies, we are exploring innovative synchronization techniques currently employed by platforms like Zoom, Google Meet, and Discord, which have demonstrated competency in managing real-time audio feeds.
- **Musical Audio Sensitivity.** Our research acknowledges that users are particularly sensitive to timing issues in musical contexts. As such, our solution will not only address the technical aspects of data transmission but also the perceptual components that influence user experience. By studying the synchronization strategies of successful applications, we aim to develop an approach that is acutely attuned to the nuances of musical rhythm and the user's auditory expectations.

Through these strategies, we aim to enhance the robustness and responsiveness of our platform, delivering a synchronized karaoke experience that meets the high standards of our users. This endeavor will be an ongoing process, reflecting our commitment to continuous improvement and innovation.

IV. Results

In conclusion, SingSphere has demonstrated the capacity to provide an engaging and socially connective online karaoke experience while highlighting the complexities of real-time audio streaming. Our latency performance tests indicate that while our system upholds robust streaming under various scenarios, there is room for improvement in voice and music synchronization. Efforts to scale up the system through vertical scaling and load balancing have been initiated, presenting a promising avenue for enhancing user experience. Furthermore, our exploration into synchronization strategies used by prominent communication platforms suggests potential adaptive solutions that could be implemented in future iterations. SingSphere's development reflects a step forward in the evolution of online social entertainment, and ongoing research will focus on optimizing real-time audio processing to achieve near-perfect synchronization.