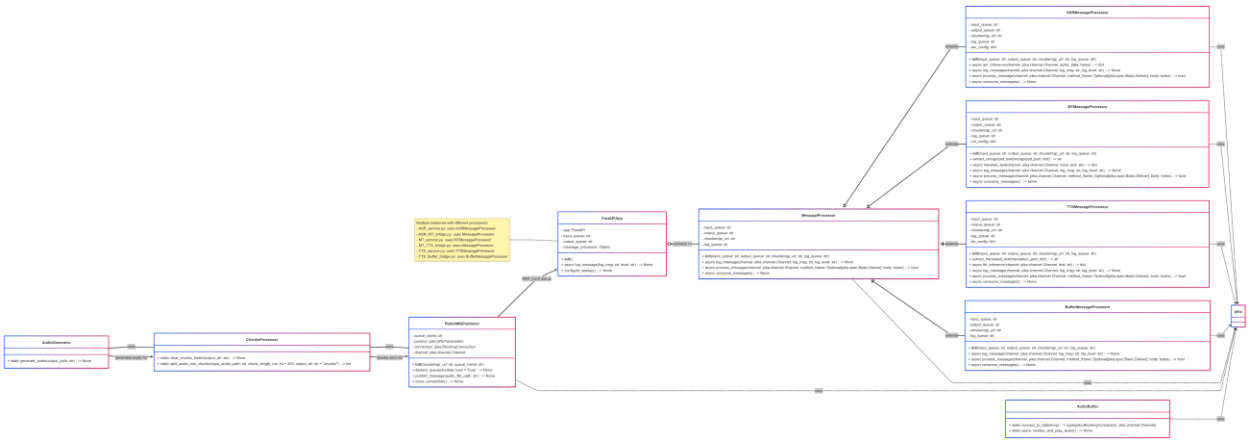
Product Design

Team – 44 – SPEECH TO SPEECH BUFFER

Team Members: Arya, Varun, Akash, Karthik, Prashant.

Design Model

Image Link



Class 1: ASRMessageProcessor

Class State	• input_queue: Name of the input queue for audio files • output_queue: Name of the output queue for ASR results • cloudamqp_url: URL for RabbitMQ connection • log_queue: Name of the log queue • asr_config: Configuration for ASR API based on input language
Class Behavior:	• __init__(): Initializes the processor with queue configuration and ASR settings • asr_inference(): Performs speech recognition on audio data • log_message(): Logs messages to the specified queue • process_message(): Processes individual messages with error handling • consume_messages(): Main message consumption loop with retry logic

Purpose	Handles the Automatic Speech Recognition (ASR) process, receiving audio files from an input queue, sending them to an ASR API, and publishing the results to an output queue. Includes robust error handling and retry mechanisms.

Class 2: MTMessageProcessor

Class State	• input_queue: Name of the input queue for ASR results • output_queue: Name of the output queue for MT results • cloudamqp_url: URL for RabbitMQ connection • log_queue: Name of the log queue • mt_config: Configuration for MT API based on language pair
Class Behaviour	• __init__(): Initializes the processor with queue configuration and MT settings • extract_recognized_text(): Extracts text from ASR JSON response • translate_text(): Performs machine translation on recognized text • log_message(): Logs messages to the specified queue • process_message(): Processes individual messages with error handling • consume_messages(): Main message consumption loop with retry logic

Purpose	Handles the Machine Translation (MT) process, taking recognized text from ASR output, sending it to a translation API, and publishing the translated results to the next queue. Includes error handling and retry mechanisms.

Class 3: TTSMessageProcessor

Class State	• input_queue: Name of the input queue for MT results • output_queue: Name of the output queue for TTS results • cloudamqp_url: URL for RabbitMQ connection • log_queue: Name of the log queue • tts_config: Configuration for TTS API based on output language
Class Behaviour	• __init__(): Initializes the processor with queue configuration and TTS settings • extract_translated_text(): Extracts translated text from MT JSON response • tts_inference(): Performs text-to-speech conversion on translated text • log_message(): Logs messages to the specified queue • process_message(): Processes individual messages with error handling • consume_messages(): Main message consumption loop with retry logic
Purpose	Handles the Text-to-Speech (TTS) process, converting translated text to audio files and publishing the results to the next queue. Includes error handling and retry mechanisms.

Class 4: BufferMessageProcessor

Class State	• input_queue: Name of the input queue for TTS results • output_queue: Name of the output queue for final audio buffer • cloudamqp_url: URL for RabbitMQ connection • log_queue: Name of the log queue
Class Behavior	• __init__(): Initializes the processor with queue configuration • log_message(): Logs messages to the specified queue • process_message(): Processes TTS results, downloads audio files, potentially compresses them, and publishes to the buffer queue • consume_messages(): Main message consumption loop with retry logic
Purpose	Manages the final stage of the pipeline, downloading audio files from S3 URLs in the TTS response, potentially compressing them, and forwarding them to a buffer queue for playback.

Class 5: MessageProcessor

Class State	• input_queue: Name of the input queue • output_queue: Name of the output queue • cloudamqp_url: URL for RabbitMQ connection • log_queue: Name of the log queue
Class Behavior:	• __init__(): Initializes the processor with queue configuration • log_message(): Logs messages to the specified queue • process_message(): Simple pass-through of JSON messages between queues • consume_messages(): Main message consumption loop with retry logic
Purpose	Acts as a generic message bridge between different service components, simply forwarding messages from one queue to another without any transformation.

Class 6: FastAPIApp

Class State	• app: FastAPI application instance • input_queue: Name of the input queue • output_queue: Name of the output queue • message_processor: Instance of specific MessageProcessor class
Class Behavior	• __init__(): Initializes the FastAPI application and MessageProcessor • log_message(): Delegates logging to the message processor • configure_startup(): Configures application startup event

	• start_consumer(): Starts async task for message consumption
Purpose	Acts as an API service wrapper around each message processor, creating an HTTP server that initializes and manages the message consumption process. Multiple instances exist for different parts of the pipeline (ASR, MT-bridge, MT, TTS-bridge, TTS, Buffer-bridge).

Class 7: RabbitMQPublisher

Class State	• queue_name: Name of the target queue • params: Connection parameters for RabbitMQ • connection: The RabbitMQ connection • channel: The RabbitMQ channel
Class Behavior	• __init__(): Initializes connection to RabbitMQ • declare_queue(): Declares the target queue • publish_message(): Publishes audio file to the queue • close_connection(): Closes the RabbitMQ connection
Purpose	Handles the initial publication of audio chunks to the ASR input queue, serving as the entry point for the entire pipeline.

Class 8: AudioGenerator

Class State	Static methods only
Class Behavior	generate_audio(): Generates a sample audio file with specific duration and properties
Purpose	Creates test audio files for demonstration purposes, generating a 30-second speech sample to be processed by the pipeline.

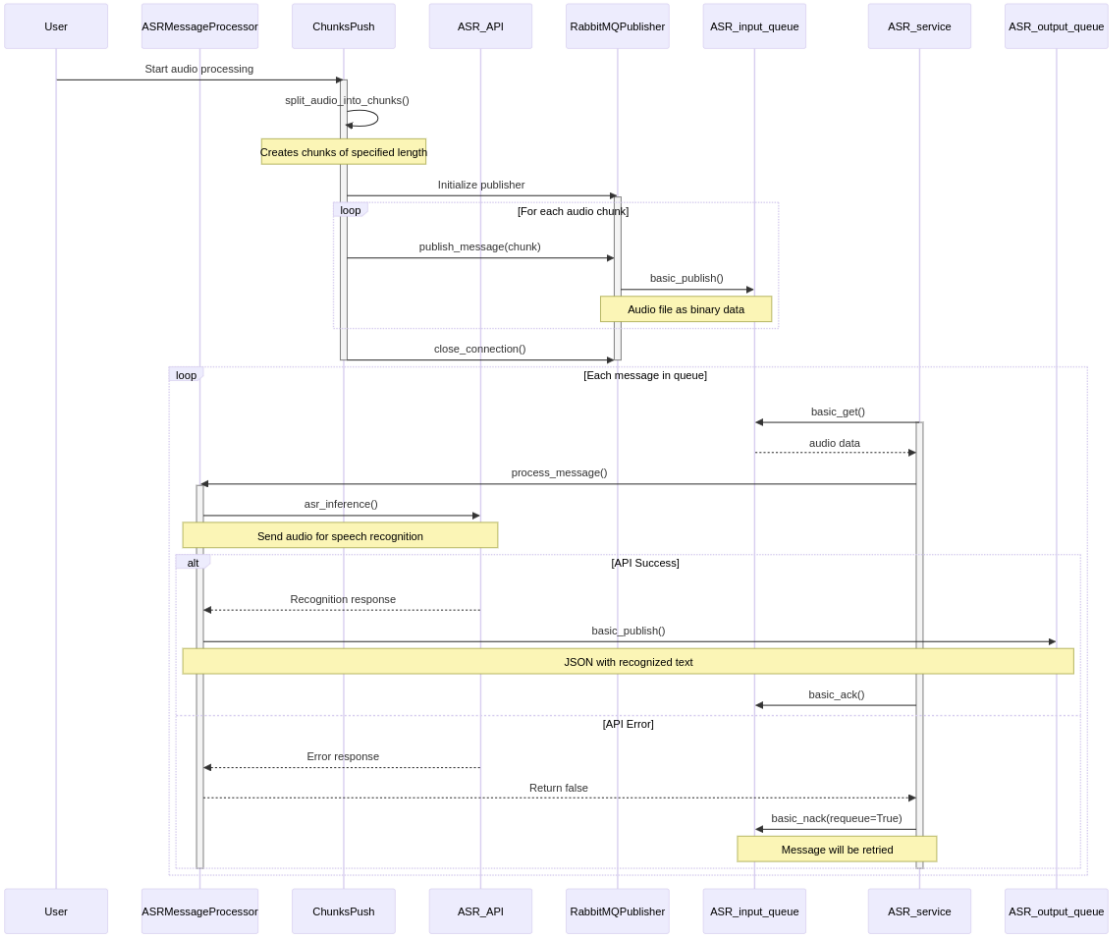
Class 9: JSONFixer

Class State	Static methods only
Class Behavior	fix_json_string(): Attempts to fix malformed JSON strings
Purpose	Utility class for repairing malformed JSON data that might be encountered in the message processing pipeline.

Sequence Diagram

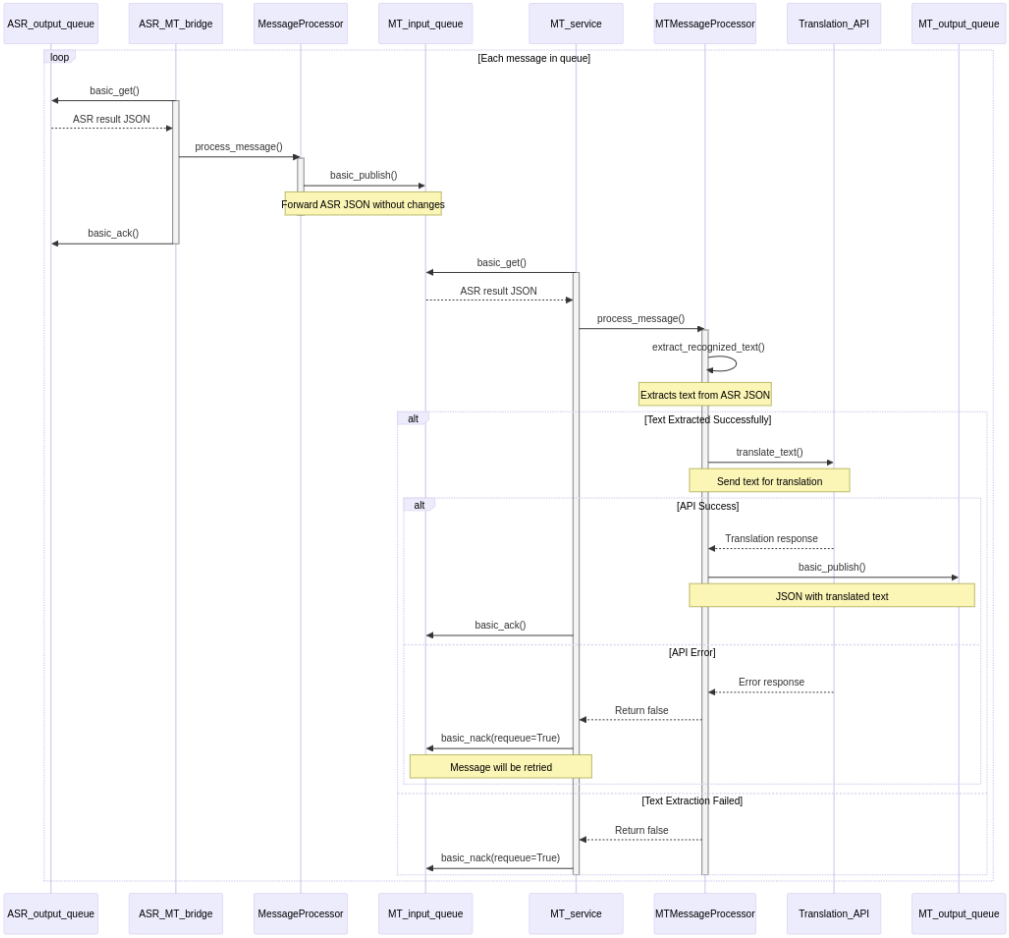
Use Case 1: Audio Upload and ASR Processing

image link



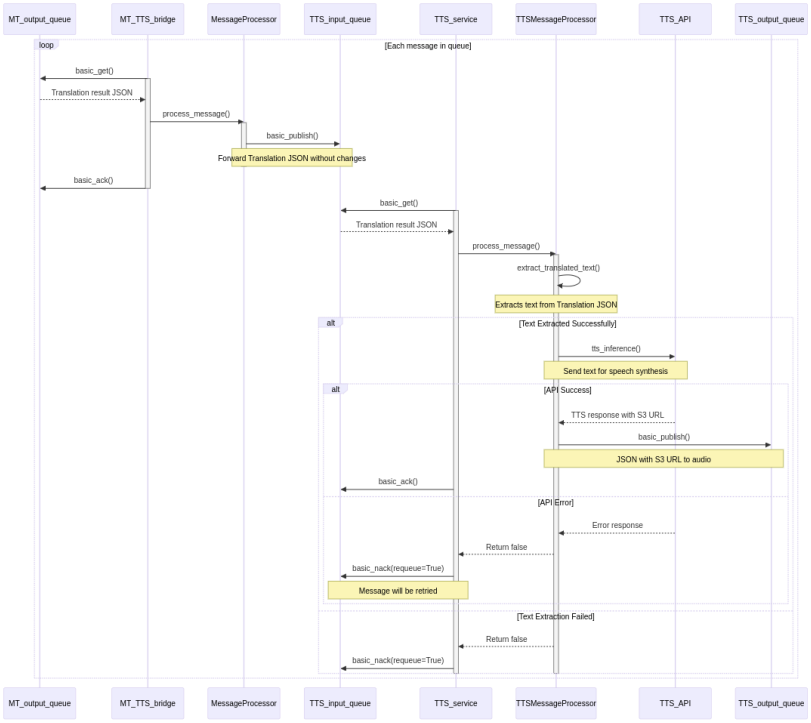
Use Case 2: Translation Process

Image Link



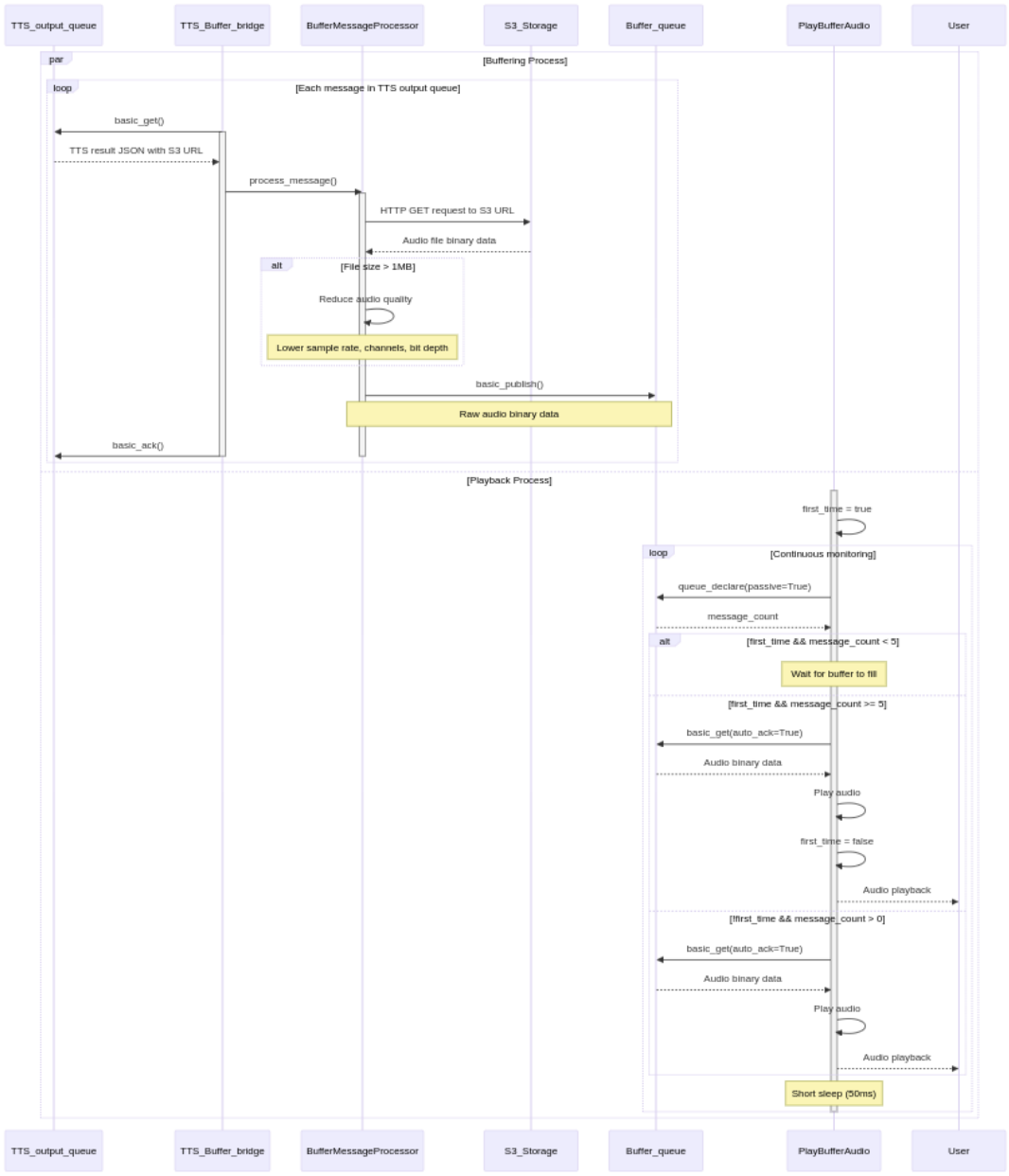
Use Case 3: Text-to-Speech Conversion

Image Link



Use Case 4: Audio Buffer and Playback

[Image Link]



Design Rationale

Rejected Proposals

1. Durable RabbitMQ Queues with Backup Logic

Alternative Considered:

We were tasked with enhancing the durability of our RabbitMQ queues. To address this, we proposed and implemented a backup logic that dynamically stores a copy of the queue on the local server. This backup synchronizes with the queue as changes occur.

Pros:

- Increased Reliability: The backup provides a safety net against cases where the queue might be deleted from the admin side.
- Local Data Visibility: Server-side copies enable direct access to queue data, facilitating troubleshooting and inspection.
- Proactive Monitoring: Enables ongoing monitoring of queue health, allowing quick resolution of issues.

Cons:

- Latency Impact: The backup process adds additional latency to queue operations.
- Interference with External Deletion: Deletions performed by external code may be blocked unless settings are adjusted.
- Effort vs. Benefit: The increased complexity and resource usage may not be justified given that it addresses a relatively rare scenario.

Final Decision:

We opted to keep the current codebase intact. The backup logic will be an optional feature—activated based on the specific use case of the backend (not solely for the speech-to-speech buffer scenario). This approach maintains general-purpose flexibility while addressing durability as needed.

2. Handling Malformed Messages

Alternative Considered:

To further improve queue reliability, we explored the idea of using an intermediate code module to automatically correct malformed JSON messages. Since most messages are in JSON, this solution aimed to automatically fix errors and reduce the buildup of error messages.

Pros:

Automatic Correction: Capable of fixing JSON formatting issues without needing third-party intervention.

Error Prevention: Reduces the accumulation of error messages by correcting common formatting mistakes.

Improved Stability: With fewer errors propagating downstream, overall system stability could be enhanced.

Cons:

Limited to JSON: The solution does not handle non-JSON formatted messages, leaving some cases unaddressed.

Risk of Incorrect Correction: The automatic fixes might sometimes be inaccurate, potentially introducing further issues.

Final Decision:

After discussing with the client, we decided to log all malformed messages to an error queue without attempting automatic correction. This business decision was made to avoid the risk of incorrect corrections causing more significant problems.

3. Logging Strategy

Alternative Considered:

Initially, RabbitMQ messages were logged locally on the server, which was convenient for the server manager. We later evaluated shifting to logging via a separate queue on the RabbitMQ server.

Pros of Local Logging:

- Ease of Access: Logs stored locally are straightforward for the server manager to review and manage.

Pros of RabbitMQ Queue Logging:

- Centralized Management: A separate logging queue on RabbitMQ makes it easier for anyone to access and manage logs across the system.

Final Decision:

We adopted the RabbitMQ site logging as our final approach. Although it may be less familiar to the concerned server manager, it aligns better with our business goals by providing a scalable and centrally managed logging solution.

4. Text-Based Chunking Approach

Alternative Considered:

Initially, we implemented a system that processed audio by first converting it entirely to text and then chunking the text for further processing. This approach was intended to simplify the pipeline by working with uniform text data.

Pros:

- Processing uniformity with consistent text data format
- Potentially simpler implementation for downstream processing
- Reduced complexity in handling audio format variations

Cons:

- Increased latency waiting for complete audio-to-text conversion
- Loss of important prosodic features (tone, emphasis, pauses) that exist in audio

Final Decision:

We shifted to chunking the audio files (.wav) directly at 300ms intervals before ASR processing. This change significantly improved speech recognition accuracy by preserving acoustic context and allowing for better processing of speech boundaries. The direct audio chunking approach also proved more efficient for real-time applications, as each chunk could be processed immediately without waiting for complete transcription. This decision aligned with our goal of optimizing both accuracy and response time in the speech processing pipeline.

Accepted Proposals

1. Server Crash Robustness

To ensure our project's robustness, we proactively tested the system's behavior during server crashes by running the code on a local server and simulating crash conditions. This testing confirmed that our system could recover gracefully from unexpected shutdowns, and our client was very satisfied with the robustness this approach provided.

2. Centralizing Our System

Managing queues across different locations prompted us to explore cloud solutions for centralization. After evaluating several options, we chose Cloud AMQP as the best free option available. This decision allowed us to manage queues on the cloud, simplifying operations and providing a single control point. The client accepted this proposal, and we successfully transitioned to a centralized cloud-based system.3. Handling Internet Issues

3. Handling Internet Issues

Recognizing the potential impact of bad internet connectivity on system operations, we implemented measures to maintain system robustness during network outages. Our approach is simple: we continuously retry until the connection is restored.

4. Audio Chunking Logic

We conducted extensive testing to determine the optimal size for audio chunks in our processing pipeline. Our findings led us to implement a 300ms chunking strategy, which proved to deliver the best results:

- Balanced accuracy and processing speed
- Reduced latency in the speech recognition pipeline
- Improved handling of continuous speech
- Optimal memory usage on server-side processing

This chunking strategy was particularly effective when paired with our Streamline ASR implementation, creating a robust solution for speech processing that meets our performance requirements.

5. Buffer Playback Optimization

To enhance the system's responsiveness and audio quality, we implemented a buffer playback mechanism that activates only after accumulating at least 5 messages. This design decision offers several benefits:

- Reduced stuttering and audio gaps during playback
- More natural speech flow in the output
- Decreased likelihood of interruptions during processing fluctuations
- Improved overall user experience with smoother audio delivery

Testing confirmed that this threshold provides an optimal balance between responsiveness and continuous playback quality, effectively addressing the client's requirements for seamless audio output.

In summary, our design process involved:

- **Looking at Different Options:**

We explored several ways to make our system robust. This included finding better methods for keeping queues safe, fixing broken messages, and managing logs. We also checked how to handle server crashes, use cloud solutions for central control, and keep things running during internet outages.

- **Pros and Cons**

  For each option, we looked at the good points and the challenges. We considered things like extra delay, possible errors, added complexity, and relying on outside services. This helped us understand the trade-offs in every part of the system.

- **Making Appropriate Decisions:**

  After our analysis and discussions with the client, we chose the solutions that best balanced reliability, scalability, and flexibility. Our final decisions included using durable backup logic, logging errors from malformed messages, testing for server crashes, using Cloud AMQP for centralized queue management, and ensuring the system keeps trying to reconnect during internet outages until the connection returns.

AMQP and designing for offline resilience ensure that our backend can be effectively used in varied scenarios rather than being limited to a single use case.