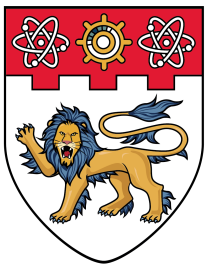

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

FULL STACK DEVELOPMENT & DEPLOYMENT OF ASR-GUI TEXT-TO-SPEECH APPLICATION

Tan Liang Meng

U2220261A

Supervisor: Prof Chng Eng Siong

A Final Year Report submitted to College Of Computing and Data Science, Nanyang Technological University in partial fulfilment of the requirements for the Degree of

BACHELOR OF ENGINEERING SCIENCE (COMPUTER SCIENCE)

2025/2026 Semester 2

Abstract

This project enhances ASR-GUI, an offline, full-stack web application that provides speech-to-text hot-word transcription using an Automatic Speech Recognition (ASR) model developed by NTU's Speech & Language Laboratory. The original system supported end-to-end transcription workflows but was limited to single-job processing and restricted audio format compatibility, reducing its practicality in real-world usage.

This work extends the system by introducing batch handling of recording transcription and deletion, allowing them to be processed concurrently rather than sequentially. In addition, the application is enhanced to support a wider range of audio formats, including WAV, MP3, MP4, M4A, and WebM, improving flexibility in audio input sources. Several usability and robustness improvements are also implemented, such as preventing deletion of recordings with active transcription jobs, optimizing recording upload latency, and providing real-time upload progress feedback. These enhancements collectively improve system efficiency, usability, and reliability,making ASR GUI a more efficient tool capable of handling heavy transcription demands.

Acknowledgement

First and foremost, I would like to express my sincere gratitude to Associate Professor Chng Eng Siong for giving me the opportunity to work on the ASR-GUI web application project. This experience significantly broadened my full-stack development skill set and provided valuable exposure to real-world system enhancement and integration.

I would also like to extend my heartfelt appreciation to my project mentor, Mr Kyaw Zin Tun, for his continuous guidance and support throughout the project. His approachable and responsive mentorship, along with the regular weekly meetings and constructive feedback, played a crucial role in helping me stay on track and successfully complete this project.

Finally, I would like to express my sincere appreciation to my friends and family for their constant support and encouragement throughout my Final Year Project. Their understanding and motivation played a crucial role in helping me complete this work.

Table of Contents

Abstract i

Acknowledgement ii

Lists of Figures vi

Lists of Tables vii

1 Chapter 1: Introduction 1

1.1 Background . . . 1

1.1.1 System Architecture . . . 1

1.1.2 Operational Workflow . . . 2

1.1.3 Current Limitations . . . 3

1.2 Objective . . . 5

1.3 Project Scope . . . 6

1.4 Contributions . . . 7

1.5 Structure of the Report . . . 8

2 Chapter 2: Literature Review 9

2.1 Related Works (Existing Speech-to-Text Services) . . . 9

2.1.1 Open-Source ASR Systems: Whisper AI by OpenAI . . . 9

2.1.2 Cloud-Based ASR Services: Google Speech-to-Text . . . 10

2.1.3 End-User ASR Application:Otter.ai . . . 11

2.1.4 Summary of Related Works . . . 12

3 Chapter 3: System Design and Architecture 14

3.1 System Overview 14
3.2 High-Level System Architecture 14
3.2.1 Software Requirements 15
3.3 Proposed Tech Stack 15
3.3.1 Next.JS Application 16
3.3.2 Node.JS & ExpressJS Server 16
3.3.3 MongoDB Database 17
3.3.4 Docker & Docker Compose 17
3.3.5 Other Technologies 18
3.4 Technical Design Considerations 18
3.4.1 Media Format Support and Audio Handling 18
3.4.2 Multi-recording Selection for Bulk Actions 19
3.4.3 Multi-config Input for Bulk Transcription 20

4 Chapter 4: System Implementation 21

4.1 Parallel Batch Processing Architecture 21
4.2 Efficient Buffer-Based Media Ingestion 22
4.3 Deletion Workflow and Data Integrity Measures 24
4.3.1 Pre-Deletion Validation 24
4.3.2 Sequential Deletion Protocol 25

5 Chapter 5: Conclusion and Future Work 27

5.1 Conclusion 27
5.2 Future Work 27
5.2.1 Real-Time Streaming Transcription 27
5.2.2 Secure Remote Accessibility 28
5.2.3 Headless API Integration 28

Lists of Figures

1-1 High-Level Technical Architecture of ASR-GUI 1

1-2 User Authentication Interface 2

1-3 Landing page of ASR-GUI (ASR Job Dashboard) 2

1-4 Recording Upload Interface 2

1-5 Transcribe Recording (Legacy System) 3

1-6 Transcript Viewing Interface (Legacy System) 3

1-7 Audio player fails to render for non-WAV formats, ASR model fails to generate transcript 4

1-8 Legacy interface restricting selection to a single recording 4

1-9 Hardcoded Feedback Timer 5

2-1 OpenAI Whisper 9

2-2 Operational workflow of OpenAI Whisper 10

2-3 Google Cloud Speech-to-Text 10

2-4 Operational workflow of Google Cloud Speech-to-Text 11

2-5 Operational workflow of Otter.ai 12

3-1 High-Level System Architecture of ASR-GUI 14

3-2 Next.JS 14 16

3-3 Express.js Server 16

3-4 MongoDB Database 17

3-5 Docker Containerization 17

3-6 Multi-Format Supports: MP3, MP4, M4A, WebM, WAV 18

3-7 Deletion Lock Mechanism to Prevent Data Inconsistencies 19

3-8 Dual Mode Configuration 20

3-9 Individual Mode Configuration 20

3-10 Configuration Interfaces of ASR-GUI 20

4-1 Implementation of Batch Processing with Concurrent HTTP Requests 21

4-2 Implementation of Recording Upload with In-Memory Buffering and Format Validation 23

4-3 Real-Time Upload Progress Visualization 24

4-4 Pre-Deletion Validation UI 25

4-5 Deletion Protocol 25

4-6 Multi-Location Cleanup 26

List of Tables

2-1 Simplified Comparison of ASR Solutions 13

3-1 ASR-GUI Software Requirements 15

Chapter 1: Introduction

1.1 Background

Automatic Speech Recognition (ASR) technology has emerged as a pivotal tool for digitizing spoken content, significantly enhancing productivity across various administrative and operational domains. To leverage this potential, the Speech & Language Laboratory at Nanyang Technological University (NTU) developed a high-accuracy, multilingual ASR model capable of handling domain-specific terminologies.

To operationalize this model, the **ASR-GUI** was developed as an offline, full-stack web application. It functions as an intuitive graphical interface for the underlying ASR engine, abstracting complex command-line interactions into a user-friendly experience.

1.1.1 System Architecture

ASR-GUI is built upon a modular client-server architecture designed for on-premise deployment. As illustrated in Figure 1-1, the system utilizes a containerized Docker environment to host the frontend, backend, and database services, ensuring consistent deployment and offline functionality without requiring continuous internet connectivity. The architecture incorporates a dedicated AI Service Layer powered by a local `llama.cpp` server to enable advanced transcript summarization and chat features.

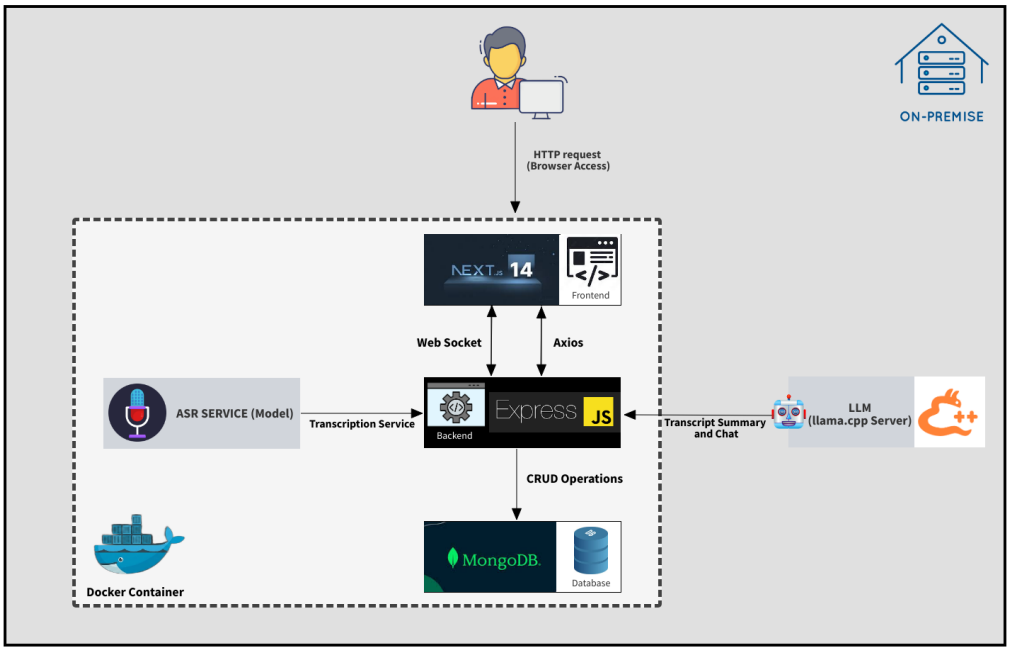

Figure 1-1: High-Level Technical Architecture of ASR-GUI

1.1.2 Operational Workflow

The existing system successfully established a proof-of-concept (PoC) with essential features such as user authentication, basic transcription services, and transcript editing capabilities. The core workflow of this legacy prototype is illustrated in Figures 1-2 to 1-6 below.

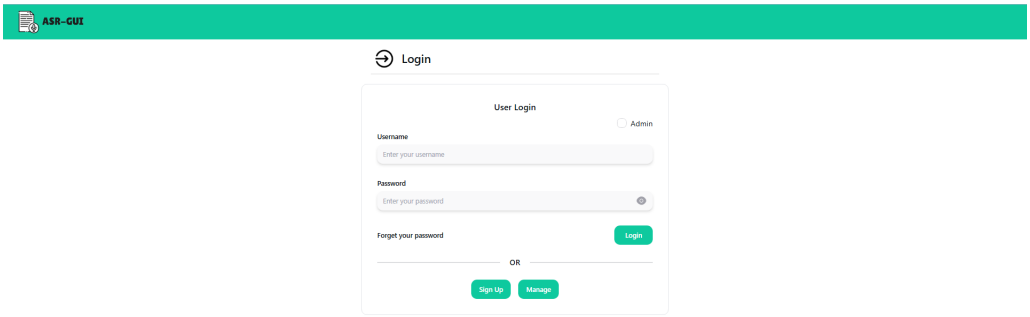

Figure 1-2: User Authentication Interface

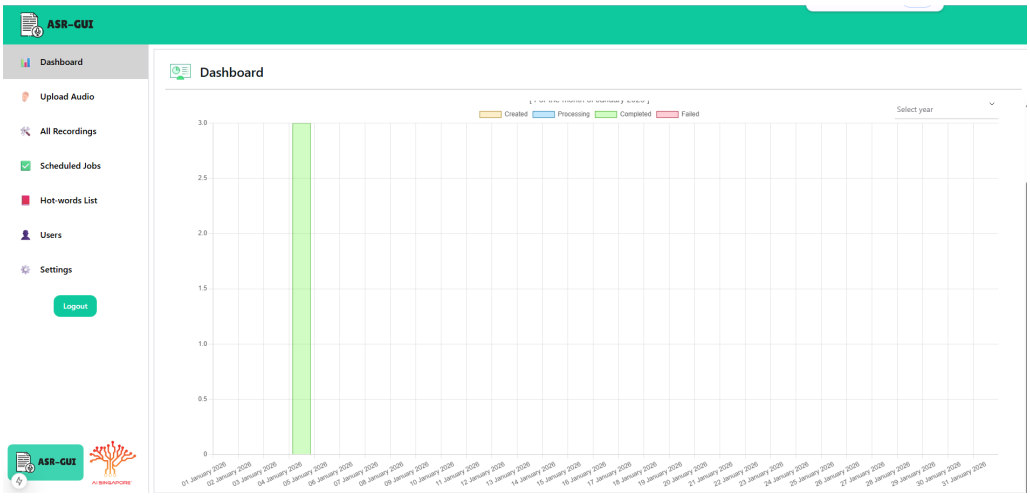

Figure 1-3: Landing page of ASR-GUI (ASR Job Dashboard)

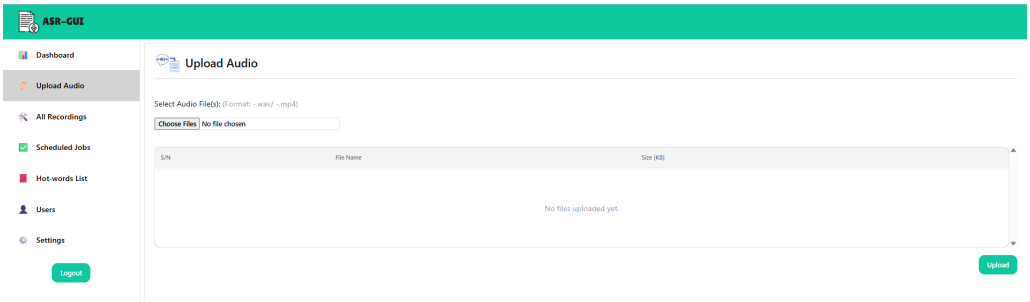

Figure 1-4: Recording Upload Interface

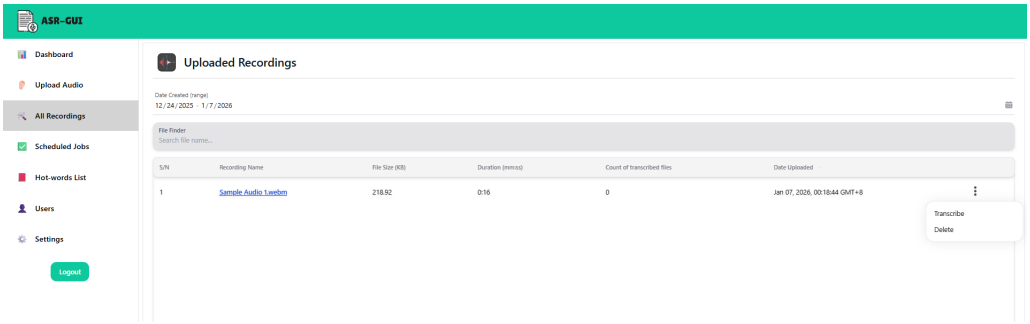

Figure 1-5: Transcribe Recording (Legacy System)

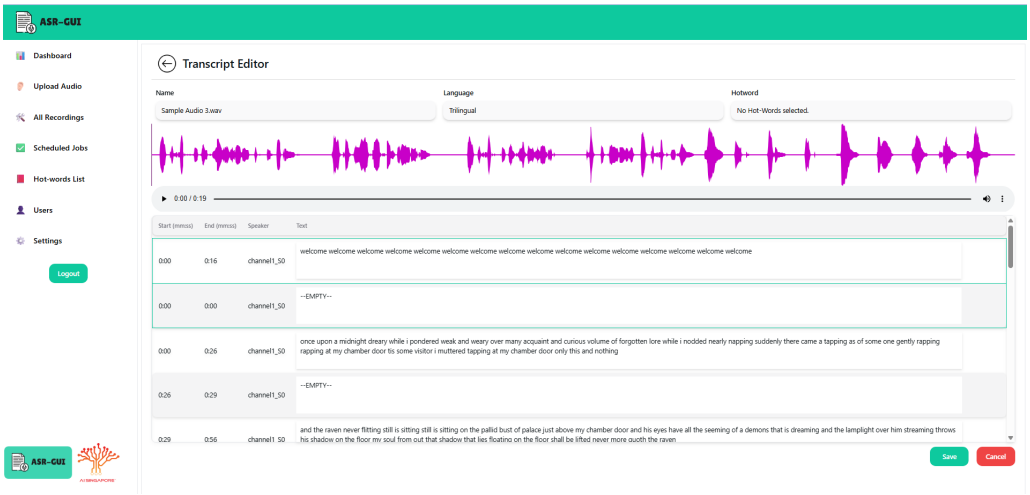

Figure 1-6: Transcript Viewing Interface (Legacy System)

1.1.3 Current Limitations

Despite establishing a functional baseline, the prototype currently operates with significant functional constraints that hinder its deployment as a production tool:

1. **Restricted Media Compatibility:** The system's audio transcription engine is strictly limited to the `.wav` format. Attempting to process other file types results in transcription failure, while the frontend audio player fails silently—refusing to render or playback the media without providing any error feedback to the user.

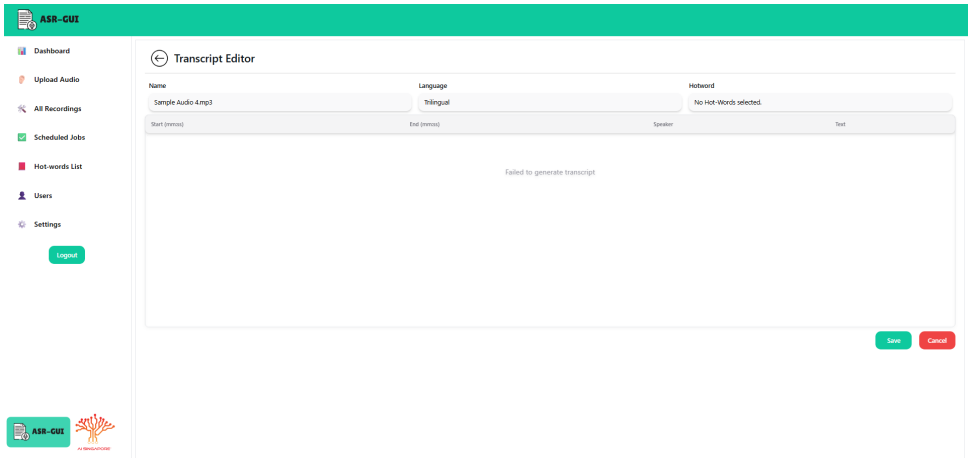

Figure 1-7: Audio player fails to render for non-WAV formats, ASR model fails to generate transcript

2. **Sequential Workflow:** Both the frontend workflow and backend processing logic are designed to handle tasks onde by one. Users are unable to batch-select recordings for transcription or deletion, creating a bottleneck for high-volume tasks.

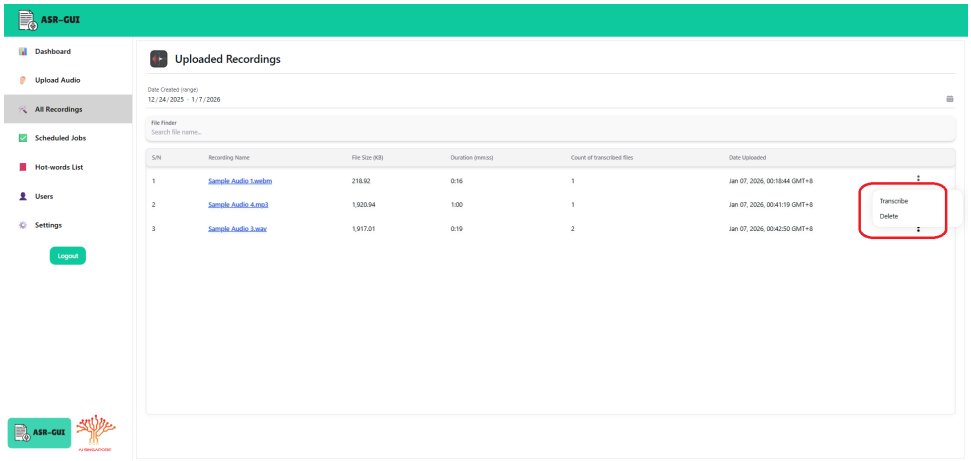

Figure 1-8: Legacy interface restricting selection to a single recording

3. **Misleading Feedback Mechanisms:** The interface relies on hardcoded heuristics rather than real-time data. For instance, file uploads trigger a fixed 5-second countdown timer regardless of the actual file size or network speed, making it misleading for the user.

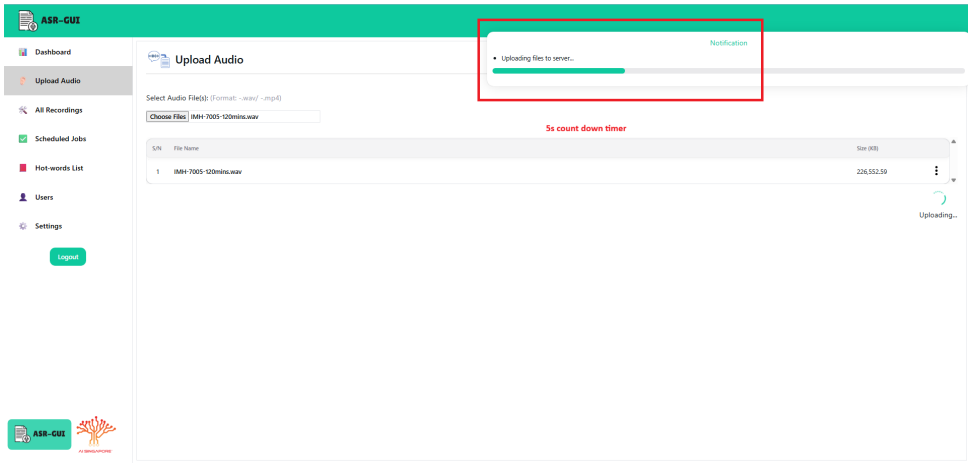

Figure 1-9: Hardcoded Feedback Timer

4. **System Instability:** The prototype suffers from several architectural defects and bugs that reduce overall reliability, including data persistence issues where deleted records occasionally reappear due to Docker volume misconfigurations.

To transition ASR-GUI from a functional prototype into a robust production tool, the system requires significant enhancements in data throughput, media compatibility, and system stability.

1.2 Objective

The primary motivation for this project stems from the operational bottlenecks identified in the previous iteration of the ASR-GUI. The existing workflow forces users to manually select, transcribe, and delete recordings one by one, which is inefficient for high-volume use cases.Furthermore, the restriction to .wav files limits usability, requiring users to perform manual file conversions before uploading. Beyond workflow limitations, the system faces stability challenges, including high memory usage during playback, unresponsive WebSocket updates, and data persistence issues where deleted records occasionally reappear due to Docker volume misconfigurations.

The primary objective of this project is to refine ASR-GUI into a stable, efficient, and user-friendly transcription tool. This involves optimizing the system for batch operations, expanding media support, and resolving critical architectural bugs to ensure reliability.

The specific goals of this project are defined as follows:

1. Enhancement of Usability and Workflow

- **Batch Processing:** To transition from a serial processing model to bulk operations, enabling users to delete and transcribe multiple recordings simultaneously.
- **Hotword Management:** To upgrade the Hotword UI, allowing users to key in aliases directly on the list and support the addition of multiple hotwords efficiently.

2. Expansion of Media Compatibility

To extend the system's capabilities beyond the legacy .wav format. The application will be updated to ingest and play back a variety of standard media formats, including .mp3, .mp4, .m4a, and .webm, accommodating diverse recording sources.

3. Performance Optimization

- **Resource Management:** To resolve high memory usage on the application tab and ensure responsive audio playback.
- **Real-time Updates:** To optimize WebSocket communication, reducing latency in file upload progress and job status updates.

4. System Reliability and Bug Resolution

- **Data Consistency:** To implement robust error handling and cleanup logic, ensuring that deleted recordings and jobs are permanently removed from both the database and Docker volumes to prevent the reappearance of old transcripts.
- **Stability Fixes:** To address critical bugs, such as the auto-logout issue upon creating a new user and data persistence errors when restarting Docker containers.

1.3 Project Scope

This section defines the specific technical deliverables and functional boundaries of the project. While the project objectives outline the strategic goals of improving usability and stability, the scope details the specific features, system optimizations, and defect resolutions implemented to achieve them.

a. Enable efficient batch processing workflows to allow the simultaneous transcription and deletion of multiple audio recordings
b. Expand the application's versatility and management capabilities:
    i. Support for diverse media formats (MP3, MP4, M4A, WebM) for file ingestion and playback, beyond the legacy WAV limitation.
    ii. Enhanced Hotword management allowing direct alias entry and bulk addition of keywords.
c. Optimize system performance to ensure responsiveness and resource efficiency:
    i. Resolution of high memory usage during prolonged browser sessions.
    ii. Optimization of WebSocket communication to minimize latency in status updates and progress bars

d. Implement comprehensive cleanup mechanisms to ensure deleted resources are permanently removed from the database and Docker volumes, preventing data persistence errors during restarts

1.4 Contributions

The contributions of this project encompass significant enhancements to the ASR-GUI's architectural robustness, operational efficiency, and user experience. By addressing critical bottlenecks in the legacy prototype, this work effectively bridges the gap between a proof-of-concept and a production-ready transcription tool. The key contributions are as follows:

a. **Upgrade for high-throughput workflows:**

   (a) Designed and implemented a batch processing logic in the backend to handle concurrent requests for transcription and deletion.

   (b) Refactored frontend state management to support bulk selection, significantly reducing the manual effort required for managing large datasets.

b. **Enhancement of system versatility:**

   (a) Integrated a versatile audio ingestion layer capable of processing lossy and compressed formats (MP3, MP4, M4A, WebM) alongside the original lossless WAV format.

   (b) Eliminated the dependency on external file conversion tools, streamlining the end-to-end user journey.

c. **Optimization of application performance:** Refined the WebSocket event loop to reduce latency, ensuring near real-time feedback for file uploads and job status updates.

d. **Reinforcement of system reliability and data integrity:** Improved system stability and data protection by fixing storage issues, ensuring that user files are permanently saved or deleted even after docker restarts

1.5 Structure of the Report

This report is structured into five chapters, each detailing specific aspects of the project:

Chapter 1 - Introduction: This chapter establishes the context of the project by presenting the background of the ASR-GUI system. It outlines the motivations driving the need for architectural enhancements, defines the specific project objectives and scope, and summarizes the key technical contributions delivered.

Chapter 2 - Literature Review: This chapter provides a comparative analysis of existing speech-to-text technologies, including OpenAI's Whisper, Google Speech-to-Text, and Otter.ai. It evaluates these solutions while justifying the necessity of the proposed ASR-GUI.

Chapter 3 - System Design and Architecture: This chapter details the high-level architectural framework of the enhanced system. It describes the adoption of a modular Component-Based Development (CBD) approach using the MERN stack and outlines the containerized infrastructure designed for secure, offline deployment.

Chapter 4 - Implementation: This chapter documents the specific engineering strategies employed to transform the architectural design into a production-ready system. It focuses on critical technical implementations, including buffer-based media ingestion, concurrent batch processing, real-time event feedback, and data integrity protocols.

Chapter 5 - Conclusion and Future Work: This chapter summarizes the project's achievements against its initial objectives and provides recommendations for future feature expansions and system optimizations.

Chapter 2: Literature Review

In this section, we will provide a comprehensive review and analysis of existing services that offer Automatic Speech Recognition (ASR) capabilities. We will evaluate their functionality and usability and compare them with those of ASR-GUI. The comparisons will highlight the strengths and limitations of ASR-GUI, as well as identify room for improvement. Overall, this section will offer a clear perspective on ASR-GUI's position within the current ASR ecosystem.

2.1 Related Works (Existing Speech-to-Text Services)

This section presents an analysis of existing speech-to-text transcription services, identifying their strengths and limitations motivates the proposed solution.

2.1.1 Open-Source ASR Systems: Whisper AI by OpenAI

Figure 2-1: OpenAI Whisper

Released in late 2022, Whisper AI represents the current state-of-the-art in open-source Automatic Speech Recognition (ASR). It is implemented as a Transformer-based encoder-decoder architecture and was trained on 680,000 hours of multilingual and multitask supervised data collected from the web [3].

Strengths: Whisper demonstrates exceptional robustness against background noise and varying accents, often outperforming fully supervised models in zero-shot transfer settings. Crucially for the medical domain, its open-weights nature allows for fully offline deployment. This enables institutions to run the model on local infrastructure, ensuring strict compliance with data residency and patient privacy regulations without transmitting audio to external cloud providers.

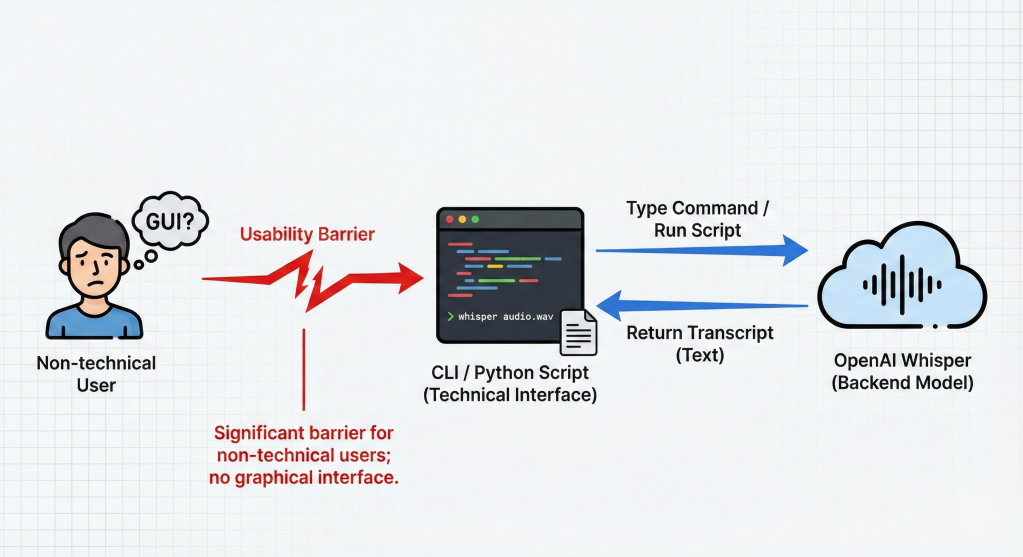

Figure 2-2: Operational workflow of OpenAI Whisper

Limitations: Despite its high accuracy, Whisper is distributed primarily as a backend Python library or Command Line Interface (CLI). This presents a significant usability barrier for non-technical end-users that require a graphical interface to interact with the system. Furthermore, the base Whisper model lacks a native, user-accessible mechanism for "hotword boosting"—the ability to bias the transcription towards specific terminology without expensive model fine-tuning. This functional gap necessitates the development of a wrapper application, such as the ASR-GUI proposed in this project, to make the underlying technology accessible and customizable for complex workflows.

2.1.2 Cloud-Based ASR Services: Google Speech-to-Text

Figure 2-3: Google Cloud Speech-to-Text

Google Cloud Speech-to-Text (STT) represents the industry standard for commercial, cloud-based transcription. It utilizes advanced deep learning neural network algorithms to convert audio to text via a REST API. Notably, Google offers domain-specific models, such as the "Medical Conversation" model, which has been fine-tuned on medical dictation and doctor-patient interactions [1].

Strengths: Google STT offers massive scalability and features that are difficult to replicate in open-source models, such as automatic punctuation, speaker diarization (distinguishing between multiple speakers), and support for over 125 languages. Its "Chirp" foundation models demonstrate high accuracy even with short utterances and technical terminology.

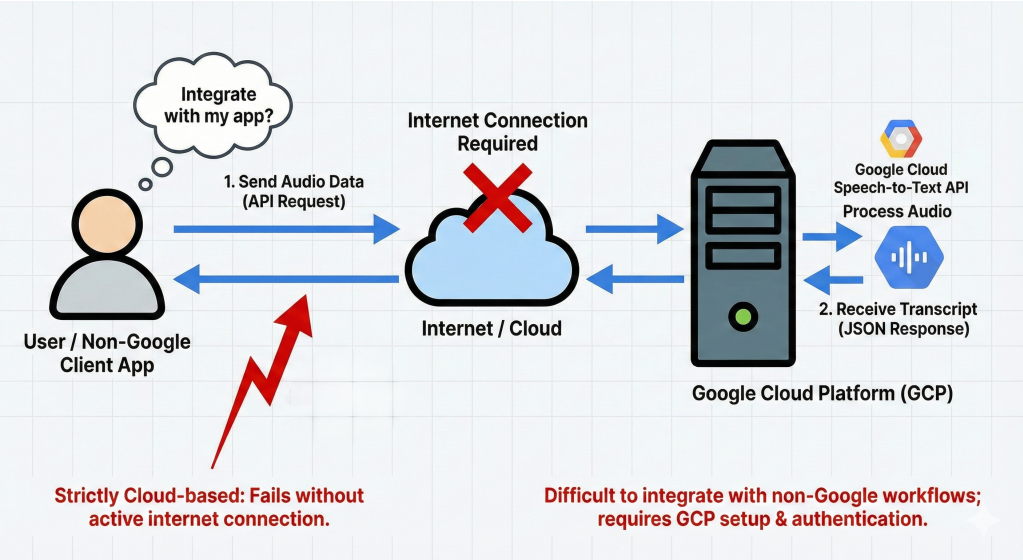

Figure 2-4: Operational workflow of Google Cloud Speech-to-Text

Limitations: As illustrated in Figure 2-4, Google Cloud SST is architecturally bound by its strict dependency on external cloud infrastructure. Unlike on-premise solutions, it requires a continuous and stable internet connection to function; audio data must be transmitted to Google's servers for processing, rendering the service inoperable in offline or air-gapped environments.

Furthermore, while the service offers seamless interoperability within the Google ecosystem (e.g., Firebase, Google Workspace), integrating it into non-Google workflows presents significant friction. Developers are often required to manage complex authentication protocols (OAuth 2.0, Service Accounts) and maintain specific Google Cloud Platform (GCP) project configurations, creating a steeper implementation curve for standalone or third-party applications.

2.1.3 End-User ASR Application:Otter.ai

Otter.ai is a leading commercial Software-as-a-Service (SaaS) application designed for meeting transcription and summarization. It offers a polished user interface that integrates with video

conferencing platforms and provides features such as speaker identification and keypoint summarization [4].

Strengths: Otter.ai excels in User Experience (UX), bridging the gap between raw ASR output and usable text notes. Key features include real-time synchronized playback (highlighting text as audio plays) and collaborative editing tools. Its proprietary diarization algorithms are highly effective at distinguishing multiple speakers in meeting environments, a feature that raw open-source models often struggle with out-of-the-box.

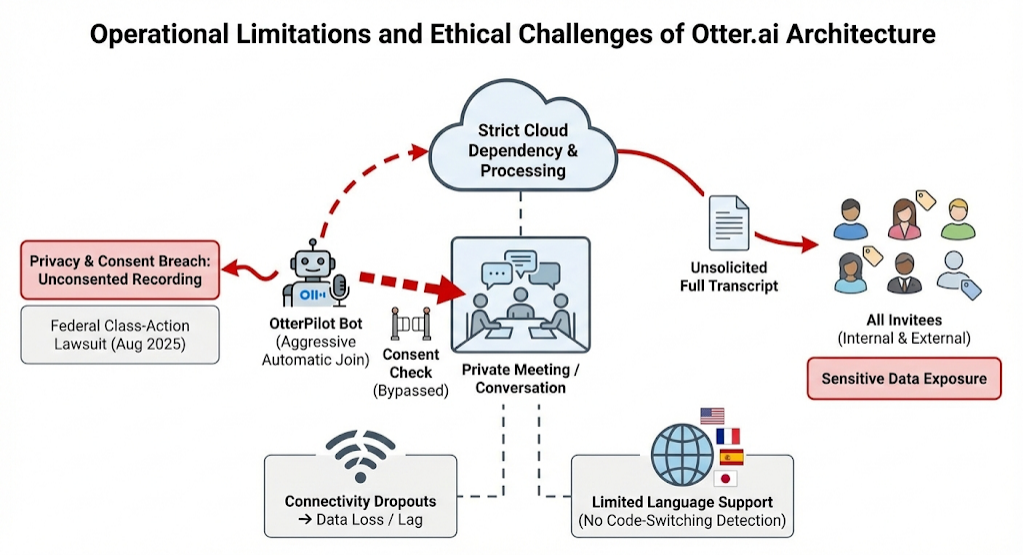

Figure 2-5: Operational workflow of Otter.ai

Limitations: While effective for general note-taking, Otter.ai's architecture presents significant ethical and technical challenges for professional deployment, as illustrated in Figure 2-5. The system's "aggressive" participation model—where the *OtterPilot* bot joins meetings automatically—has triggered severe privacy concerns and a federal class-action lawsuit (August 2025) regarding the recording of private conversations without explicit consent [2]. This privacy risk is compounded by the platform's default tendency to disseminate unsolicited full transcripts to all calendar invitees, potentially exposing sensitive data to external parties. On a technical level, the service is constrained by limited language support (English, French, Spanish, and Japanese) without real-time detection for code-switching, and its strict cloud dependency makes it vulnerable to data loss or transcription lag during connectivity dropouts.

2.1.4 Summary of Related Works

Table 2-1 summarizes the key differences between current market solutions and the proposed ASR-GUI.

The comparison highlights a clear gap in the market:

- **Privacy Issue:** Cloud-based applications like Otter.ai rely on internet connectivity, requiring audio files to be uploaded to third-party servers. This architecture poses a

privacy risk for any user handling sensitive or proprietary information, as the data must leave the secure local environment to be processed (Low Privacy).

- **Usability Issue:** Secure tools like Whisper AI are safer but require coding skills to operate (Hard to Use).

The ASR-GUI is designed to be the "best of both worlds", combining the **safety** of an offline tool with the **simplicity** of a modern web app.

Table 2-1: Simplified Comparison of ASR Solutions

Feature	**Whisper AI**	**Google Cloud**	**Otter.ai**	**ASR-GUI (Ours)**
Where it runs	Offline (PC)	Internet (Cloud)	Internet (Cloud)	**Offline (PC)**
Data Privacy	High (Safe)	Low (Shared)	Low (Shared)	**High (Safe)**
Ease of Use	Hard (Code)	Medium (API)	Easy (App)	**Easy (App)**
Cost	Free	Pay-per-use	Subscription	**Free**

Chapter 3: System Design and Architecture

3.1 System Overview

The ASR-GUI is a specialized, full-stack web application designed to operationalize the high-accuracy Automatic Speech Recognition (ASR) engine developed by the Speech & Language Laboratory at Nanyang Technological University (NTU). The primary function of the system is to provide a user-friendly Graphical User Interface (GUI) that abstracts the complexities of the underlying command-line based ASR inference engine, making advanced transcription capabilities accessible to non-technical users.

The application was designed with data privacy and deployment flexibility in mind, and operates on a containerized Client-Server architecture. This ensures that the system functions entirely offline, eliminating the need for continuous internet connectivity and preventing sensitive audio data from leaving the local environment.

3.2 High-Level System Architecture

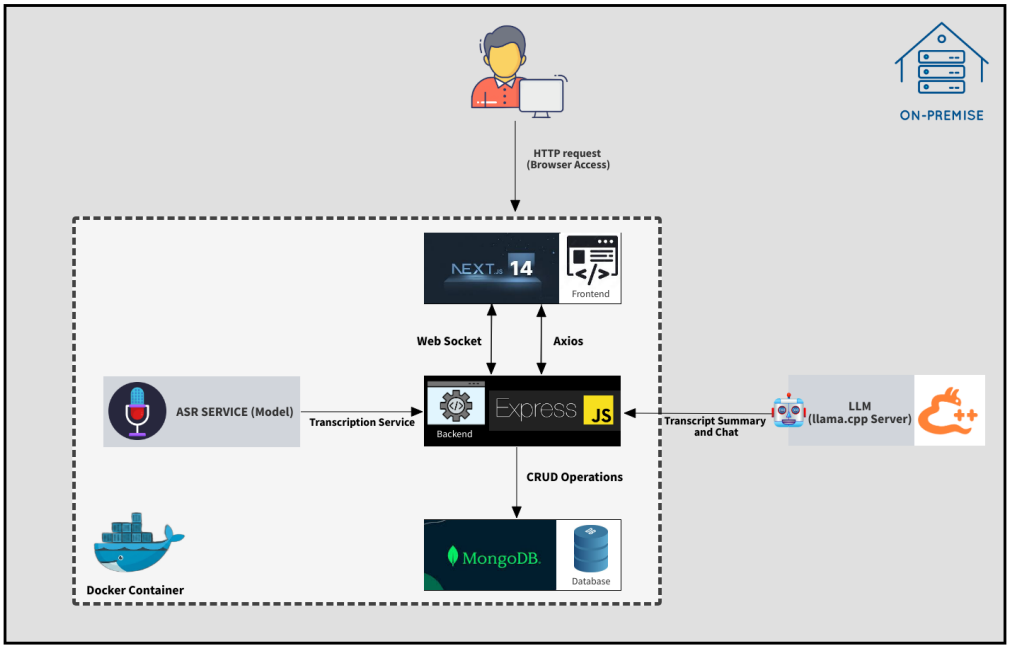

Figure 3-1: High-Level System Architecture of ASR-GUI

As shown in figure 3-1, the ASR-GUI system follows a client–server architecture, consisting of a frontend user interface, a backend processing service, and a persistent storage layer. These components communicate through well-defined application programming interfaces (APIs) and real-time communication channels. The entire system is containerized using Docker to ensure consistent deployment and offline operability.

To further augment the system's analytical capabilities, the architecture integrates a Generative AI service powered by the **Qwen3-A2B-32B-Instruct model**. This Large Language Model (LLM) is exposed as a decoupled microservice accessed via API endpoint, employing Server-Sent Events (SSE) to stream real-time transcript summaries and interactive chat context directly to the user interface.

3.2.1 Software Requirements

ASR-GUI consists of the following software requirements:

Frontend	**Backend**	**Miscellaneous**
Next.js (v15.1.2)	Node.js (v22.13.0)	Docker
NextUI (v2)	Express.js (v4.19.2)	Docker Compose
Wavesurfer.js (v7.8.8)	MongoDB	TypeScript (v5.5.3)
Axios (v1.7.2)	Mongoose (v8.4.4)	
Zod (v3.23.8)	WebSocket, ws (v8.18.0)	
	Multer (v1.4.5)	
	Music Metadata (v10.5.0)	

Table 3-1: ASR-GUI Software Requirements

3.3 Proposed Tech Stack

The development of ASR-GUI retains the core MERN stack (MongoDB, Express.js, React/Next.js, Node.js) established in the previous version. In addition to these core technologies, the system leverages TypeScript to ensure type safety across the stack, along with Mongoose for object modeling and Docker for containerized deployment.

Figure 3-2: Next.JS 14

3.3.1 Next.JS Application

Next.js is utilized as the primary frontend framework for building the user interface. It offers a hybrid rendering approach, leveraging Server-Side Rendering (SSR) to deliver fully rendered HTML to the client for improved initial load performance, and Client-Side Rendering (CSR) for dynamic interactivity. In this iteration, the application leverages the Next.js App Router for its efficient file-based routing system. Furthermore, the implementation of the React Context API facilitates global state management, which is critical for handling the complex batch-selection logic across different modules without the need for external state libraries.

Figure 3-3: Express.js Server

3.3.2 Node.JS & ExpressJS Server

The backend logic operates on the Node.js runtime. Its event-driven, non-blocking I/O model is particularly advantageous for this application's requirements, as it allows the server to handle multiple concurrent operations—such as batched file uploads and ASR job dispatching—without stalling the main execution thread. Express.js serves as the web application framework, providing a minimalist yet powerful structure for defining API endpoints and middleware. The utilization of a unified language (TypeScript/JavaScript) for both frontend and backend development significantly reduces the context-switching overhead and streamlines the implementation of shared logic.

Figure 3-4: MongoDB Database

3.3.3 MongoDB Database

MongoDB serves as the non-relational database for the application, designed to manage large volumes of unstructured data via collections. This schema flexibility accelerates the development cycle by eliminating the need for complex database migrations when features evolve. To interact with the database, the application uses Mongoose, an Object Data Modeling (ODM) library. Mongoose provides a rigorous schema-based solution on top of MongoDB, enforcing data validation and type casting to ensure data integrity before persistence.

Figure 3-5: Docker Containerization

3.3.4 Docker & Docker Compose

To address the requirement for offline functionality and ease of deployment, the application is fully containerized using Docker. By packaging the frontend, backend, and database with their specific dependencies into isolated containers, Docker ensures consistent behavior across different host environments. Docker Compose further simplifies the orchestration of this multi-service architecture through a single YAML configuration file. The entire application stack can be provisioned, networked, and managed with a single command, significantly enhancing the efficiency of development, testing and deployment across different machines.

3.3.5 Other Technologies

Several auxiliary libraries were integrated to enhance specific system functionalities:

- **WebSocket:** Establishes a real-time, bi-directional communication channel between the client and server, enabling instant feedback for long-running batch processes.
- **Axios:** A promise-based HTTP client used to execute asynchronous API requests from the frontend to the backend services.
- **Zod:** Employed for runtime schema declaration and validation, ensuring that data exchanged between the client and server adheres to strict type definitions.

3.4 Technical Design Considerations

This section discusses some key technical considerations that guided the design and implementation of ASR-GUI. These considerations address usability, performance, system reliability, and deployment constraints, ensuring that the system meets both functional and non-functional requirements.

3.4.1 Media Format Support and Audio Handling

The previous version of the system had a major limitation: it only worked with WAV audio files. This meant users had to manually convert their recordings before uploading them, which was slow and inconvenient. To fix this, we improved the system to automatically handle different file types while ensuring playback and transcription is smooth.

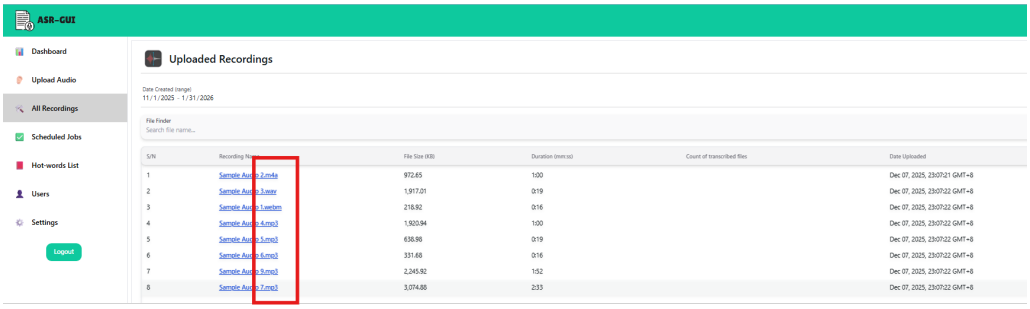

Figure 3-6: Multi-Format Supports: MP3, MP4, M4A, WebM, WAV

Diverse Media Acceptance with validation: The system's upload handler analyzes the file's binary structure using music-metadata.parseBuffer() to validate audio container formats and extract technical metadata (duration, codec). This content inspection occurs without any file extension or MIME type checking, allowing the system to accept diverse media containers (MP3, MP4, M4A, WAV, WebM) as long as they contain valid audio data. Files with corrupted

headers or invalid formats are automatically rejected when buffer parsing fails, providing resilience against malformed inputs.

3.4.2 Multi-recording Selection for Bulk Actions

Enabling batch operations necessitated a fundamental shift in frontend state management to handle multi-item tracking efficiently and safely. Enabling users to process multiple recordings simultaneously was a primary usability requirement to address the inefficiencies of the legacy workflow.

Workflow Optimization: In the previous system, administrative tasks were repetitive and time-consuming, requiring users to transcribe or delete files one by one. This version implements a bulk action workflow that allows users to select multiple recordings and trigger a single unified operation. This significantly reduces the number of clicks required for routine maintenance, allowing operators to clear old data or queue new jobs in seconds rather than minutes.

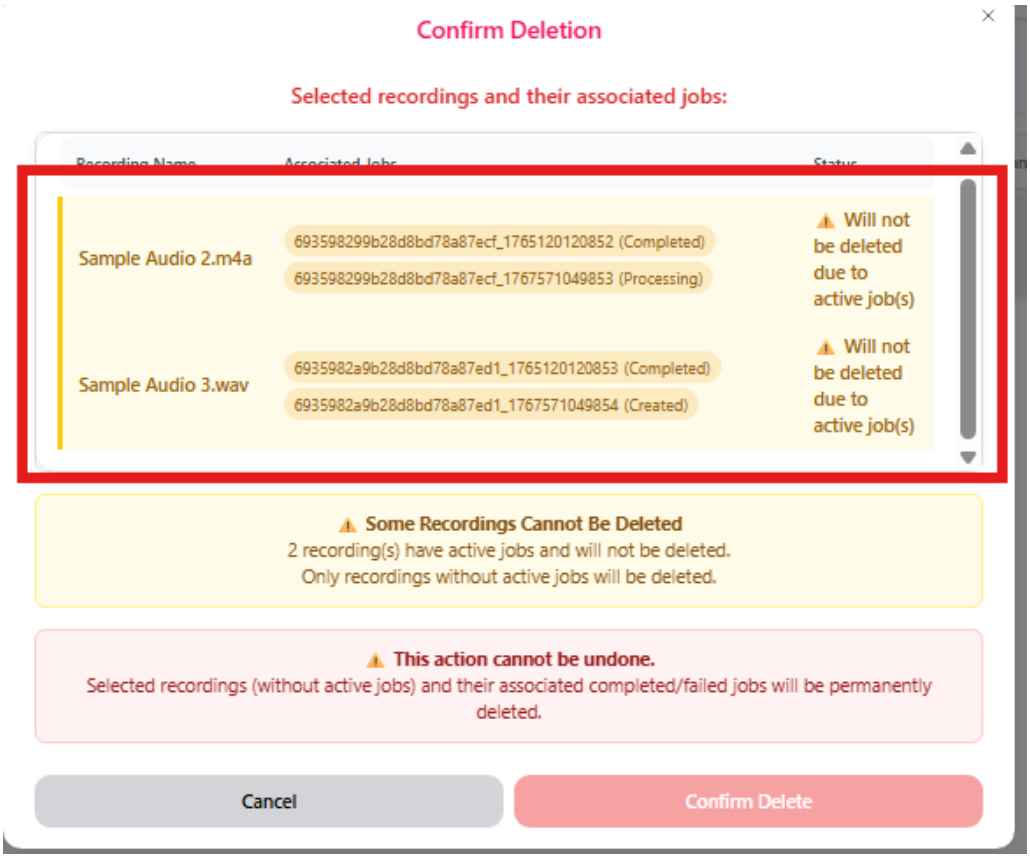

Figure 3-7: Deletion Lock Mechanism to Prevent Data Inconsistencies

Conditional Logic for Data Integrity (The Deletion Lock): To prevent database inconsistencies, the bulk deletion logic incorporates a strict validation check. The system enforces a deletion lock on any recording currently marked as Processing. Since the ASR model cannot be interrupted once inference begins, deleting a source file during this state would result

in orphan transcripts, transcripts generated for a file that no longer exists. Therefore, if a user attempts to bulk-delete a mix of Completed and Processing files, the system will only remove the completed ones and reject the active jobs to preserve data integrity.

3.4.3 Multi-config Input for Bulk Transcription

A critical challenge in batch processing is handling job configuration such as Language, Audio Type, Output format etc for multiple files that may require distinct processing parameters.

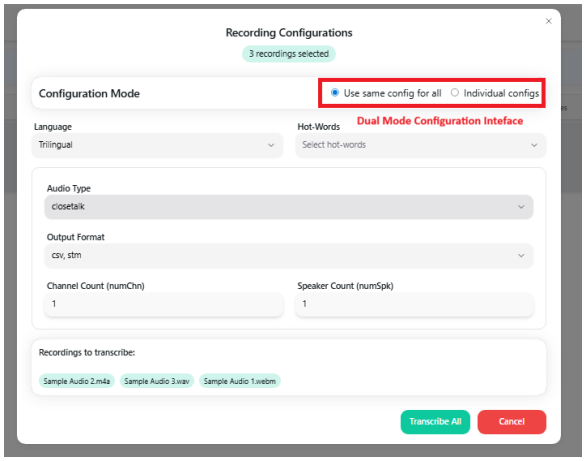

Figure 3-8: Dual Mode Configuration

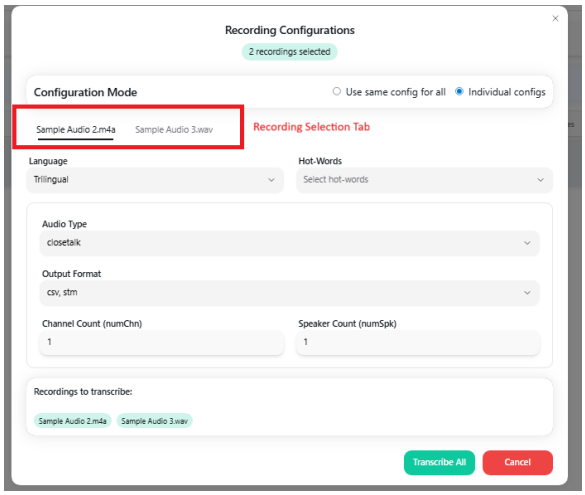

Figure 3-9: Individual Mode Configuration

Figure 3-10: Configuration Interfaces of ASR-GUI

Dual-Mode Configuration Interface: To balance flexibility with efficiency, the system implements a Dual-Mode Configuration Strategy. Users can toggle between a **Global Mode**, which applies a master template to all selected files, and an **Individual Mode**, which creates a tabbed interface for distinct configurations. This allows users to rapidly process homogenous batches while retaining the capability to handle mixed-config batches in a single workflow.

Concurrent Batch Processing: To prevent server timeouts caused by processing large batches of files in a single request, the system utilizes a Concurrent Batch Processing pattern. Instead of sending one monolithic payload, the frontend iterates through the user's selection and triggers concurrent asynchronous requests to the server for each selected item. This approach maximizes network throughput by processing multiple items simultaneously and ensures fault tolerance. If a specific file fails, the error is isolated to that single request, allowing the remaining files in the batch to be processed successfully without aborting the entire operation.

Chapter 4: System Implementation

This chapter details the engineering implementation of the ASR-GUI, focusing on the transformation of the legacy prototype into a stable, high-performance application. The technical strategies detailed below were selected specifically to address the operational objectives defined in **Section 1.2**.

4.1 Parallel Batch Processing Architecture

The legacy system's sequential workflow created a significant bottleneck, requiring users to manually trigger operations one by one. To address **Objective 1 (Enhancement of Usability and Workflow)**, specifically the requirement for high-throughput operations, the frontend workflow was fundamentally re-engineered to support parallel execution.

```
const handleConfirmDelete = async () => {
  try {
    // Filter out recordings with active jobs
    const recordingsToDelete = recordingIds.filter(id => !recordingsWithActiveJobs.includes(id));

    if (recordingsToDelete.length === 0) {
      setAlert('No recordings can be deleted due to active jobs', true);
      setIsDeleteModalOpen(false);
      return;
    }

    // Delete only recordings without active jobs
    const deletePromises = recordingsToDelete.map(id => deleteUploadedRecordings(id));
    await Promise.all(deletePromises);

    const deletedCount = recordingsToDelete.length;
    const skippedCount = recordingsWithActiveJobs.length;

    let message = `Successfully deleted ${deletedCount} recording(s)`;
    if (skippedCount > 0) {
      message += `. ${skippedCount} recording(s) skipped due to active jobs`;
    }

    setAlert(message);
    setRecordingIds([]);
    setIsDeleteModalOpen(false);
  } catch (error) {
    setAlert('Error deleting selected recordings');
  }
};
```

Figure 4-1: Implementation of Batch Processing with Concurrent HTTP Requests

- **Concurrent Job Dispatching:** The frontend implements a parallel HTTP request pattern for batch operations. The system uses **Promise.all()** to dispatch the API requests concurrently rather than sequentially. Figure 4-1 shows an example of how bulk deletion maps each validated recording ID to an HTTP DELETE request and executes them simultaneously, allowing the browser to send multiple requests in parallel. The

Node.js backend handles these concurrent HTTP requests naturally through its event loop, processing each recording operation independently as requests arrive.

- **Operational Efficiency:** This parallel request architecture reduces the wall-clock time for administrative tasks from a user perspective. For example, deleting or transcribing a batch of ten recordings now requires a single user action (one button click) and completes faster due to overlapping network I/O, compared to the previous approach that required manual triggering of each operation individually. While each recording is still processed separately on the backend, the concurrent HTTP requests eliminate the sequential bottleneck at the user interaction level.

4.2 Efficient Buffer-Based Media Ingestion

To fulfill **Objective 2 (Expansion of Media Compatibility)**, the system required a robust mechanism to ingest diverse file formats without the performance penalties associated with the legacy WAV-only system. Furthermore, to satisfy **Objective 3 (Performance Optimization)**, the upload process needed to minimize processing overhead and provide immediate user feedback.

To achieve these goals, the system employs a buffer-based ingestion strategy with asynchronous metadata extraction:

- **In-Memory Buffering:** The backend utilizes Multer middleware configured for memory storage (`multer()` with no storage options), buffering uploaded files entirely in RAM during transmission. This approach eliminates intermediate disk writes during the active upload phase. Once the complete file buffer is received, the backend persists it to disk using Node.js's asynchronous file system operations (`fs.promises.writeFile()`) as shown in figure 4-2 line 137. This balances memory efficiency with I/O performance.
- **Post-Upload Format Validation:** After successful upload and disk persistence, the system employs the `music-metadata` library's `parseBuffer()` method to extract technical metadata from the buffered file data (as illustrated in figure 4-2 line 141 to 150). This library analyzes binary headers, container structures, and codec information to identify format types, supporting MP3, MP4, M4A, WAV, WebM, and other multimedia containers. By validating formats through programmatic buffer analysis rather than filename extensions, the system ensures accurate format detection and prevents malformed or mislabeled files from entering the transcription pipeline. Critical metadata such as duration and file size are stored in the database alongside the file reference.

```
1 reference
public uploadRecording = async (
  req: Request,
  res: Response,
  next: NextFunction,
) => {
  try {
    console.log('Uploading recording to server...');
    const userId = req.body['userId'] as string;
    const files = req.files as Express.Multer.File[];
    const userDirectoryPath: string = path.join(
      recordingDirectoryPath,
      userId,
    );

    console.log('Validating server directories...');
    checkOrCreateRecordingDirectory(recordingDirectoryPath);
    checkOrCreateUserDirectory(userDirectoryPath);

    await Promise.all(
      files.map(async (file: Express.Multer.File) => {
        console.log('Storing recording file to backend server...');
        const filePath = path.join(userDirectoryPath, file.originalname);
        await fs.promises.writeFile(filePath, file.buffer);

        console.log('Uploading recording metadata to backend server...');
        const fileStats: fs.Stats = await fs.promises.stat(filePath);
        const { parseBuffer } = await loadMusicMetadata();
        const audioMetadata = await parseBuffer(file.buffer);
        const newRecording = new RecordingModel({
          userId: userId,
          fileName: file.originalname,
          fileSize: fileStats.size,
          duration: audioMetadata.format.duration || 0,
          pathDir: filePath,
        });

        await newRecording.save();
      })
    );
    console.log(`Saved ${files.length} recording files to server.`);

    return res.status(200).json({
      message: 'Successfully uploaded recording to server!',
    });
  } catch (error) {
    next(error);
```

Figure 4-2: Implementation of Recording Upload with In-Memory Buffering and Format Validation

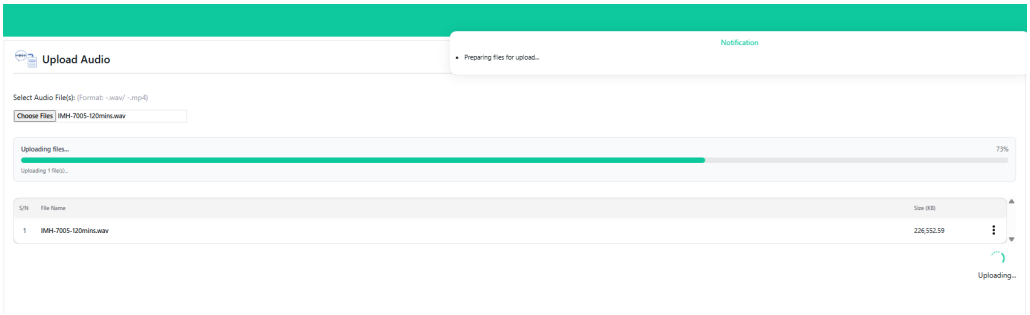

Figure 4-3: Real-Time Upload Progress Visualization

- **Real-Time Upload Progress Tracking:** The frontend implements byte-accurate progress monitoring using Axios's `onUploadProgress` callback, which tracks HTTP multipart/form-data transmission in real time. As shown in figure 4-3, the progress indicator calculates percentage completion based on bytes transmitted (`progressEvent.loaded / progressEvent.total`), providing users with precise visual feedback during file uploads. This replaces the unresponsive interface of the legacy system and ensured users have clear visibility into long-running upload operations.

4.3 Deletion Workflow and Data Integrity Measures

To address **Objective 4 (System Reliability and Data Consistency)**, the system implements several safeguards in the deletion workflow. The approach combines pre-deletion validation, ordered operations, and error handling to minimize inconsistent states.

4.3.1 Pre-Deletion Validation

Before deletion operations execute, the frontend performs validation checks to prevent problematic deletions:

- **Active Job Detection:** The system queries all associated ASR jobs for selected recordings and identifies those with `CREATED` or `PROCESSING` status. Recordings with active transcription jobs are automatically filtered out from bulk deletion operations, preventing the removal of files currently being processed by the ASR engine.

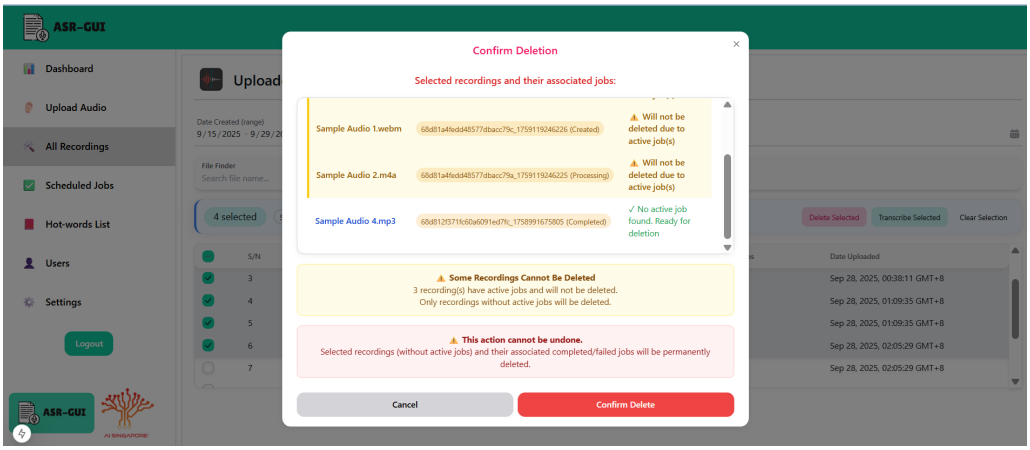

Figure 4-4: Pre-Deletion Validation UI

- **User Notification:** Users receive clear feedback about which recordings cannot be deleted and why, with separate counts for successful deletions and skipped items due to active jobs.

4.3.2 Sequential Deletion Protocol

The backend implements an ordered deletion sequence to minimize certain classes of inconsistency:

```
255     // Delete the recording
256     const deletedRecording = await RecordingModel.findOneAndDelete({
257       _id: recordingId,
258     });
259
260     console.log('Successfully deleted recording metadata from database.');
261
262     if (!deletedRecording) {
263       return res
264         .status(404)
265         .json({ message: 'Recording not found in database.' });
266     }
267
268     if (!deletedRecording.pathDir) {
269       return res
270         .status(404)
271         .json({ message: 'Recording file is not uploaded to server.' });
272     }
273
274     // Delete the associated audio file from the filesystem
275     fs.unlinkSync(deletedRecording.pathDir);
276     console.log('Successfully deleted recording file from server.');
277
```

Figure 4-5: Deletion Protocol

- **Database-First Ordering:** As shown in figure 4-5 line 256, the system first removes recording metadata and associated configuration records from MongoDB using `findOneAndDelete()` operations. Only after successful database deletion does the

system attempt filesystem cleanup using Node.js's synchronous `fs.unlinkSync()` function (line 275).

```
274     // Delete the associated audio file from the filesystem
275     fs.unlinkSync(deletedRecording.pathDir);
276     console.log('Successfully deleted recording file from server.');
277
278     // Clean up corresponding files from root input directory
279     try {
280       const rootInputPath = '/node-service/asr/input';
281
282       // Find all jobs associated with this recording to get their jobIds
283       const relatedJobs = await ASRJobModel.find({ recordingId: recordingId });
284
285       for (const job of relatedJobs) {
286         const jobId = job.jobId;
287
288         // Check and delete audio files and JSON configs from root input folder
289         const possibleFiles = [
290           path.join(rootInputPath, `${jobId}.mp3`),
291           path.join(rootInputPath, `${jobId}.wav`),
292           path.join(rootInputPath, `${jobId}.m4a`),
293           path.join(rootInputPath, `${jobId}.webm`),
294           path.join(rootInputPath, `${jobId}.json`),
295         ];
296
297         for (const filePath of possibleFiles) {
298           if (fs.existsSync(filePath)) {
299             fs.unlinkSync(filePath);
300             console.log(`Successfully deleted ${filePath} from input directory.`);
301           }
302         }
303       }
304     } catch (inputCleanupError) {
305       console.warn('Warning: Could not clean up files from input directory:', inputCleanupError);
306     }
```

Figure 4-6: Multi-Location Cleanup

- **Multi-Location Cleanup:** Following primary file deletion, the system attempts best-effort cleanup of related files in the ASR Docker volume's input directory (audio files and JSON configurations), using a try-catch wrapper to prevent cleanup failures from blocking the main deletion flow as illustrated in figure 4-6.

Chapter 5: Conclusion and Future Work

5.1 Conclusion

The development of the ASR-GUI has successfully bridged the gap between high-performance research algorithms and practical end-user accessibility. By operationalizing the NTU Speech & Language Laboratory's ASR engine into a full-stack web application, this project addresses the critical usability and workflow limitations of the legacy command-line interface.

The system achieves its primary objectives through three key architectural decisions. First, the expansion of media compatibility eliminates the rigid dependency on WAV files, allowing users to seamlessly process diverse inputs including MP3, MP4, and WebM formats. Second, the **Concurrent Batch Processing:** strategy significantly optimizes the transcription workflow, enabling batch operations that maximize throughput without compromising system stability. Finally, the **Containerized Architecture** ensures that the application remains deployment-agnostic and privacy-compliant, capable of running entirely offline on local hardware to protect sensitive audio data.

Beyond standard transcription, the integration of the Qwen Large Language Model transforms the system from a passive transcription tool into an active analytical assistant. This feature allows users to not only generate text but to immediately synthesize summaries and extract insights, substantially reducing the time required for post-transcription analysis. In summary, ASR-GUI delivers a robust, scalable, and user-centric solution that operationalizes advanced speech recognition technology for real-world deployment.

5.2 Future Work

While the current iteration of ASR-GUI meets all core functional requirements, several avenues for further enhancement have been identified to expand its scope and scalability.

5.2.1 Real-Time Streaming Transcription

The current system operates on a batch-processing model (File Upload $\rightarrow$ Process). A significant enhancement would be the implementation of **Real-Time Streaming ASR** via WebSocket technology. This would allow users to transcribe live audio feeds (e.g., from a microphone or live meeting stream) with low latency, displaying the text as it is spoken. This requires refactoring the backend ingestion layer to handle continuous audio streams rather than discrete file payloads.

5.2.2 Secure Remote Accessibility

The current system architecture is strictly containerized for **On-Premise** deployment to ensure data sovereignty. While this guarantees privacy, it limits operational flexibility by restricting access to the local network infrastructure. Future iterations could implement a secure reverse-proxy or VPN-based gateway. This would enable authorized personnel to review and manage transcripts remotely via mobile devices or off-site workstations without compromising the strict air-gapped security model inherent to the design2.

5.2.3 Headless API Integration

The system currently relies exclusively on a **Graphical User Interface (GUI)** for all interactions. To evolve the application into a central infrastructure component, future development should focus on exposing the backend functionality via a documented **RESTful API**. This would facilitate machine-to-machine workflows (External System $\rightarrow$ API $\rightarrow$ Transcript), allowing third-party applications to programmatically dispatch audio jobs and retrieve results without manual user intervention.

References

[1] Google. *Cloud Speech-to-Text: Medical Models Documentation*. https://cloud.google.com/speech-to-text/docs/medical-models. Accessed: 2024-03-21. 2024.

[2] Steve Larson. *Otter.ai Faces Class-Action Lawsuit Over Secret Meeting Recordings*. Accessed: 2026-01-08. Stoll Berne. Aug. 2025. URL: https://stollberne.com/class-actions-blog/otter-ai-faces-class-action-lawsuit-over-secret-meeting-recordings/.

[3] OpenAI. *Introducing Whisper*. Accessed: 2026-01-26. Sept. 2022. URL: https://openai.com/index/whisper/.

[4] Otter.ai. *What Is Conversational AI? How It Works and Use Cases*. https://otter.ai/blog/conversational-ai. Accessed: 2025-08-05. 2025.