School of Computing and Mathematical Sciences

CO7201 Individual Project

Final Report

Actor's Line Learning tool

Sudharsan Velraja

sv223@student.le.ac.uk

239042988

Project Supervisor: Dr Karim Mualla
Principal Marker: Dr Anand Sengodan

Word Count: 18284
16th May 2025

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Sudharsan Velraja
Date:16th May 2025

Contents

List of Figures

1. Introduction

Rehearsal plays a crucial role in the creative journey of an actor, enabling them to enhance their performance, delve into character motivations, and assimilate scripts with greater efficacy [1]. While rehearsals typically involve teamwork with directors, co-actors, and acting mentors, actors often find themselves rehearsing solo, which presents obstacles like insufficient feedback, challenges in sustaining focus, and potential inaccuracies in line delivery [2].
Recent developments in artificial intelligence (AI) have catalysed remarkable progress in the creative arts, encompassing areas such as acting, scriptwriting, and performance coaching [3]. AI technologies now offer resources for self-directed learning and practice, thereby improving individuals' capacity to hone their abilities independently of conventional collaborative settings [4]. Furthermore, advancements in speech recognition and natural language processing (NLP) have proven especially beneficial in automating functions like dialogue simulation and providing feedback on pronunciation [7].
Mobile applications have become significant tools for education and training, providing flexibility, accessibility, and intuitive interfaces [5]. In the realm of acting, mobile rehearsal applications enable users to rehearse scripts anytime and anywhere, thereby enhancing efficiency and minimizing reliance on in-person collaborators. Additionally, the ability to function offline is essential for users with inconsistent internet access, guaranteeing that training resources remain accessible even in areas with limited connectivity [6].
Apropos of these challenges and opportunities, the AI Script Rehearsal App was created. This application allows actors to practice their lines interactively, providing real-time feedback through advanced AI technologies such as speech recognition, semantic similarity evaluation, and text-to-speech functionalities. By integrating a user-focused design with state-of-the-art AI innovations, the app meets the demand for rehearsal solutions that are accessible, self-sufficient, and efficient.

1.2 Challenges Faced by Actors

Actors face multiple obstacles during the rehearsal process, including:

- Securing Scene Partners: It can be quite challenging for students, freelancers, or individuals preparing for auditions on short notice to find someone who is available to rehearse at a suitable time. [8].

- Lack of Constructive Critique: Individual practice sessions often fail to provide valuable insights into performance quality, complicating the process for actors to recognize errors or opportunities for enhancement. [9].

- Motivation and Engagement: Conventional rehearsal techniques often lack variety and do not effectively replicate the vibrant energy of an authentic performance, which can diminish an actor's motivation and hinder their learning outcomes.[10].

- Resource Limitations: Financial constraints may hinder the ability of novice and independent artists to hire professional coaches or invest in specialized rehearsal software. [11].

These challenges underscore the necessity for a rehearsal solution that is more accessible, adaptable, and intelligent.

1.3 Role of AI and Mobile Applications in Creative Arts

Artificial intelligence technologies, including speech recognition, machine learning, and algorithms for assessing sentence similarity, are gaining significant traction in creative fields [12]. Mobile applications serve as robust platforms that broaden access to training resources for a diverse user base [13]. In conjunction with script rehearsal, artificial intelligence can:

- Identify spoken dialogue in real-time, enabling performers to rehearse independently of a human counterpart [14].
- Utilise natural language processing to assess the actor's delivery against the original script, delivering immediate feedback on precision [15].
- Monitor development over time, assisting users in evaluating enhancements and identifying aspects that require focus [16].
- Provide a tailored experience by modifying speech tempo, tone, and vocal characteristics to align with the demands of the character [17].

Mobile devices offer the benefit of portability, allowing users to practice in various locations and at any time, without the requirement for specialised equipment [18].

1.4 Aim and Objectives of the Project

The overarching aim of this project is to design, develop, and evaluate an **AI-powered mobile application** that assists users in **script rehearsal** through **speech recognition**, **real-time feedback**, and **performance tracking**.
The specific objectives are as follows:

- **Develop a user-friendly mobile app** using React Native (Expo framework) to allow platform-independent deployment (Android and iOS).
- **Integrate speech recognition** capabilities to capture the user's spoken lines during rehearsal.
- **Implement a natural language processing (NLP) system** to evaluate the similarity between the spoken lines and the original script, using a lightweight FastAPI server.
- **Design an intuitive, chat-style user interface**, allowing scripts to be displayed like a conversation for ease of rehearsal.
- **Provide visual feedback** after each rehearsal session, including highlighting correct and incorrect lines and visualizing progress over time through charts.
- **Create adjustable settings** such as speech pitch, rate, and gender to customise the rehearsal experience for different user needs.

Through achieving these objectives, the application aims to empower actors and performers by making rehearsal more accessible, engaging, and effective.

1.5 Structure of the Report

This report is organized into several chapters:

- **Literature Review:** Provides a review of existing literature and technologies related to AI-based rehearsal applications, speech recognition, NLP, and mobile development.
- **System Analysis:** Details the system analysis, including essential functional and desirable requirements and optional functional.
- **System Design:** Outlines the system design and architecture, including UI/UX considerations.

- **Implementation:** Describes the implementation process, challenges faced, and technical decisions made during development.
- **Testing and Evaluation:** Covers the testing methodologies applied and the evaluation of the system's performance and usability.
- **Results and Discussion:** Presents the results, discusses their implications, and compares the developed application with existing methods.
- **Future Work:** Explores potential future work and improvements.
- **Conclusion:** concludes the report by summarizing the achievements and personal reflections on the project.

2. Literature Review

2.1 Introduction

This chapter examines current technologies, applications, and research pertinent to the creation of a mobile application powered by artificial intelligence for script rehearsal. It addresses available rehearsal tools, progress in speech recognition, natural language processing methods for assessing text similarity, offline-first mobile frameworks, and optimal practices in chat-based user interfaces. The analysis identifies deficiencies in existing solutions and lays the technological groundwork for this project.

2.2 Existing Rehearsal Applications

A variety of mobile applications have been created to aid performers in the process of script learning and rehearsal. Noteworthy examples include:

- Script Rehearser is a mobile application tailored for actors to practice their scripts autonomously. It enables users to record their lines, listen to playback, and rehearse scenes with adjustable pacing and character options [19]. While Script Rehearser provides versatile functionalities such as voice recording and automatic cueing, its main emphasis lies on recording and playback capabilities, in contrast to the AI Script Rehearsal App, which offers real-time feedback on speech accuracy and semantic similarity.

- Rehearsal Pro is a premium application that enables actors to emphasize scripts, insert annotations, and record their performances [20]. It is popular among professional actors due to its user-friendly interface and effective annotation features. Nevertheless, it does not incorporate AI-based feedback systems for assessing the accuracy of spoken lines, semantic validity, or automated rehearsal tracking. Additionally, it does not offer real-time speech recognition or feedback capabilities [20].

- LineLearner is designed to enable users to record their lines and subsequently play them back for rehearsal purposes [22]. While it provides a straightforward and user-friendly interface, it is deficient in advanced functionalities such as speech recognition, real-time feedback, and semantic analysis, which constrains its effectiveness for actors desiring a more comprehensive rehearsal experience.

- Scene Partner: An application dedicated to voice recording and playback, which predominantly utilises pre-recorded audio and lacks AI-driven feedback mechanisms [23].

- ColdRead is a mobile teleprompter application designed to assist actors in delivering their lines during self-taped auditions, although it provides only basic rehearsal tracking functionalities [24].

Although these applications present valuable functionalities like script reading, text highlighting, and audio playback, they lack capabilities such as real-time line recognition, sentence similarity assessment, and an interactive conversational interface. This initiative seeks to fill these deficiencies by integrating real-time speech recognition, AI-driven feedback, and a chat-oriented user interface.

2.3 Advances in Speech Recognition Technologies

Over the last ten years, speech recognition technology has made remarkable progress, primarily driven by advancements in machine learning, deep learning, and the capabilities of mobile hardware. Notable developments encompass:

- The **Google Speech-to-Text** API is a cloud-based service that provides highly accurate transcription capabilities and supports a variety of languages [7].
- **Expo Speech Recognition Module**: A React Native interface that encapsulates native speech recognition APIs, providing a lightweight and offline-capable solution ideal for mobile applications [26].
- **On-device Speech Models**: Contemporary mobile operating systems, including iOS and Android, are progressively enhancing their capabilities for offline speech recognition, thereby augmenting both efficiency and user privacy [27].

The Speech Recognition module from Expo was selected for this project due to its minimal resource requirements, capability for offline operation, and facilitation of seamless cross-platform implementation.

2.4 Natural Language Processing for Sentence Similarity

Evaluating the accuracy of a user's spoken lines requires comparing the spoken input to the original script. In natural language processing, several methods exist to measure text similarity:

- **Cosine Similarity**: Computes the cosine of the angle formed by two vectors representing the sentences. Suitable for measuring the similarity of bag-of-words representations [28].
- **Levenshtein Distance**: Calculates the least number of modification (insertions, deletions, substitutions) required to convert one string into another [29].
- **Semantic Embeddings**: Using models such as Word2Vec, GloVe, or Sentence-BERT to capture semantic meaning beyond exact word matching [30].

Given the need for lightweight and fast evaluation in a mobile context, simpler techniques like Levenshtein Distance or lightweight embedding-based methods are appropriate. This project uses a FastAPI backend to process and score sentence similarity efficiently without overloading the client device.

2.5 Chat-Based User Interfaces for Learning and Engagement

Research indicates that chat-based user interfaces (UIs) can enhance engagement, particularly in educational and training contexts. Benefits include:

- **Familiarity**: Most users are comfortable with messaging apps, lowering the learning curve [31].
- **Turn-Taking Simulation**: Chat UIs mimic real conversations, improving user immersion during dialogue practice [32].
- **Focus on Segmentation**: Presenting script lines as individual chat bubbles helps users focus on one idea at a time, supporting cognitive load theory [33].

By designing the rehearsal app to display scripts in a chat-bubble format, users are guided naturally through scenes, making the experience more dynamic and relatable compared to traditional script readers.

2.6 Offline-First Mobile Application Architecture

An important design requirement for this project is ensuring functionality even when no internet connection is available. Offline-first mobile app design principles suggest:

- **Data Caching and Local Storage**: Storing critical data locally using tools like AsyncStorage or SQLite [34].
- **Graceful Degradation**: Allowing features that depend on network connectivity (e.g., advanced feedback) to degrade gracefully without causing app crashes [35].
- **Sync Mechanisms**: Optionally syncing data when the device is back online [36].

This project leverages AsyncStorage for local script and rehearsal data storage, ensuring seamless rehearsal experiences without requiring constant internet access.

2.7 Identified Gaps and Research Direction

While existing rehearsal applications offer important functionality, they often fall short in the following areas:

- Lack of real-time feedback on spoken lines.
- No integration of sentence similarity evaluation.
- Limited or no offline support.
- Static user interfaces that do not simulate conversation dynamics.

This project aims to fill these gaps by developing an AI-powered, offline-capable mobile application offering real-time rehearsal assistance in an engaging and accessible format.

2.8 Proposed Solution

To address the identified problems and meet the outlined requirements, the proposed solution is the development of a **mobile AI Script Rehearsal Application** built using **React Native with Expo** for the frontend and **FastAPI** for the backend services (optional for online features). The application will integrate speech recognition and AI-based sentence similarity evaluation to provide real-time rehearsal support.

The core of the solution revolves around a **chat-based rehearsal interface** where users interact with script lines as if messaging a partner. Speech recognition will capture user responses, while a lightweight local database (AsyncStorage) will track rehearsal progress. A simple offline-first design ensures that the application remains functional without relying heavily on internet connectivity.

Key aspects of the proposed solution include:

- **Chat-style Rehearsal Interface**: Scripts are presented in familiar WhatsApp-style bubbles for intuitive interaction.
- **Speech Recognition and Matching**: Spoken lines are processed and evaluated against the original text using semantic similarity scoring.
- **Progress Tracking**: Past attempts are stored and visualized using bar charts and statistics.
- **Offline Capability**: Core rehearsal functions, including script loading, speech capturing, and progress storage, are available offline.
- **Customisable Settings**: Users can adjust speech playback pitch, rate, and gender to tailor their rehearsal experience.

This solution emphasizes **usability**, **accuracy**, and **accessibility**, ensuring that users can rehearse scripts anytime, anywhere, with minimal setup.

3. System Analysis

3.1 Introduction

This chapter offers a comprehensive examination of the system requirements, architecture, and operational workflows pertaining to the AI Script Rehearsal Application. It elaborates on the defined problem statement, as well as both functional and non-functional requirements, accompanied by visual aids such as use case diagrams and system flowcharts. A meticulous system analysis guarantees that the final product meets user expectations and project objectives.

3.2 Problem Definition

Performers, learners, and orators frequently encounter difficulties when practicing scripts in the absence of collaborators. Current applications mainly focus on script reading or line recording, yet they fall short in providing real-time feedback, assessing sentences, and offering engaging, interactive interfaces. Additionally, numerous solutions depend on constant internet access, which restricts their functionality in offline settings.

Consequently, there exists a necessity for a mobile application powered by artificial intelligence that enables users to:

- Engage in independent practice scripts.
- Receive prompt feedback on online precision.
- Function smoothly without internet access.
- Immerse yourself in an interactive, chat-oriented rehearsal setting.

3.3 Requirements

3.3.1 Essential Requirements:

- **Mobile Application**: Develop an intuitive app that acts as the primary interface for users.
- **AI-Powered Text-to-Speech (TTS):** Read scripts aloud with distinct character voices to differentiate roles effectively.
- **Speech Recognition & Error Detection: Analyse** an actor's spoken lines, identify discrepancies, and provide real-time feedback.
- **Semantic Understanding:** Recognize when a spoken line differs from the original text but retains the same meaning, ensuring minor paraphrasing isn't marked as a mistake.
- **Customisable Voice Settings:** Adjust voice gender, pitch, and speed to match user preferences and enhance immersion.
- **Script Management System**: Enable users to upload, store, and organize scripts within the app.
- **Real-Time Corrections:** Offer immediate feedback when an actor makes an error while rehearsing.
- **Deadline & Notifications:** Allow users to set deadlines and receive push notifications to encourage consistent script practice.
- **Video Recording:** Provide an option to record the user's performance for later review.

3.3.2 Recommended Requirements:

- **Progress Tracker:** Track rehearsal progress over time, offering insights into improvement and consistency.
- **Intuitive User Interface**: Enable users to easily upload, edit, and interact with scripts through a clean, well-organized UI.
- **Shared Scripts**: Allow users to mark scripts as public or private. Public scripts can be accessed and rehearsed by all users, promoting collaboration and community learning.

- **Leaderboard**: Display rankings based on user performance for public scripts, encouraging friendly competition and motivation.

3.3.3 Optional Requirements:

- **Custom Background Colours**: Allow users to personalize script display backgrounds (e.g., yellow or green) to improve readability and reduce eye strain.
- **Full-Script Playback**: Enable users to listen to the entire script with AI-generated voices before practicing.

3.4 Use case Diagram and Workflow

To gain a deeper insight into the interactions between users and the AI Script Rehearsal Application, a use case diagram has been created. This diagram serves to visually illustrate the various methods through which users can engage with the system, such as loading scripts, choosing characters, modifying rehearsal parameters, practicing dialogue with speech recognition, and assessing performance feedback. The use case diagram (Figure 1) offers an extensive examination of the essential functionalities of the application.

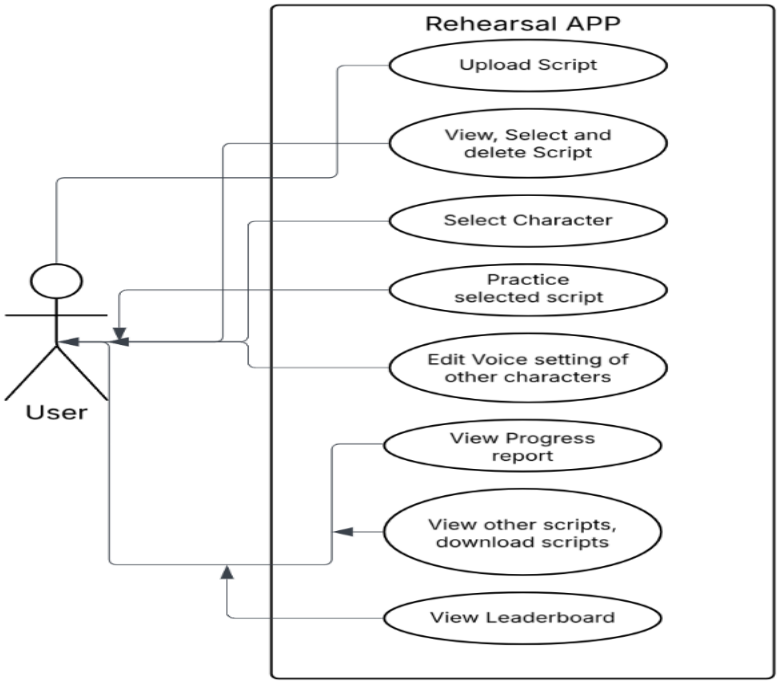


Figure 1: **Use case Diagram**

3.4.1 System Workflow

The high-level system workflow during rehearsal is as follows:

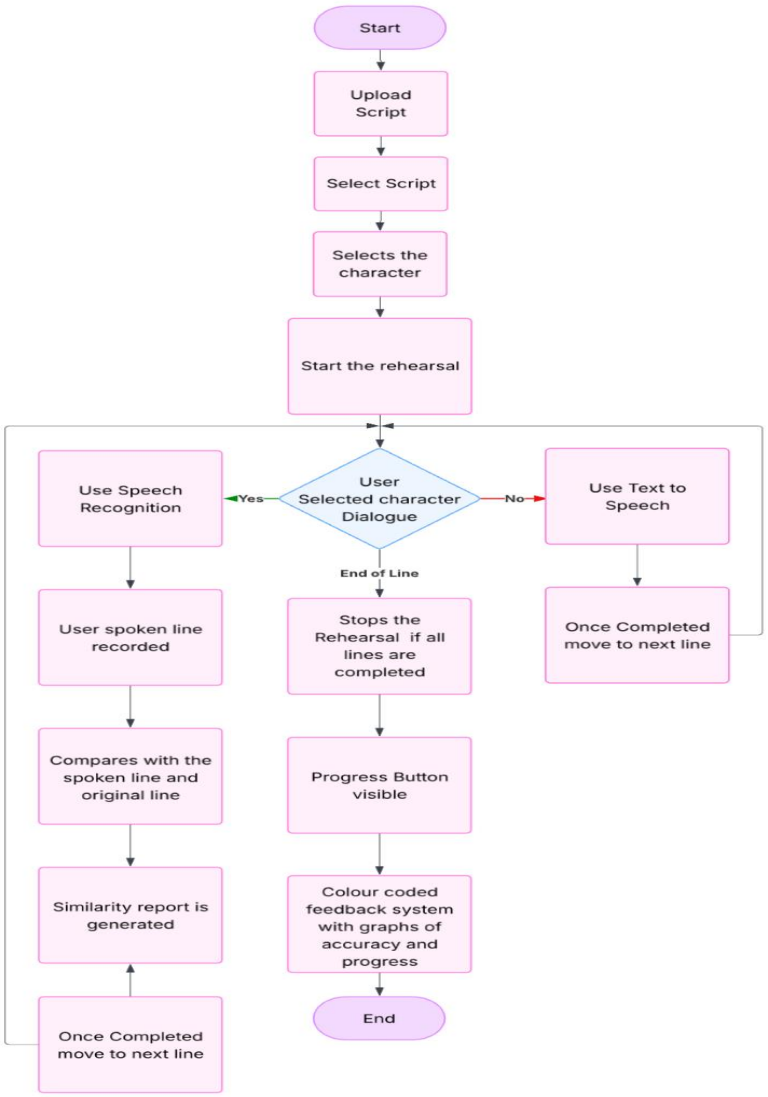


Figure 2: High level System Workflow Diagram

- User upload script documents
- User selects a script and character.
- App displays the script line-by-line in a chat bubble format.
- User taps on a line and speaks.
- User taps the play button to start the rehearsal.
- Speech recognition records the spoken input.
- Spoken input is compared to the original line.
- Sentence similarity score is calculated and stored.
- Colour coded Feedback is shown once the user clicks the progress button.
- After completing all lines, user views a summary with a bar chart showing overall performance.

4. System Design

4.1 Introduction

The process of system design converts the specified requirements and analyses into a comprehensive blueprint for the development of the application. This chapter delineates the architecture, technology stack, module designs, and data flow pertinent to the AI Script Rehearsal Application. Through the use of visual diagrams and thorough explanations of components, it is ensured that the system remains scalable, maintainable, and efficient.

4.2 System Architecture

The application is structured according to a client-server architecture, featuring a frontend developed with React Native Expo and a backend powered by FastAPI. The React Native component manages user interactions, local data storage, speech recognition, and script rehearsal functionalities, whereas the FastAPI backend delivers critical services include text extraction from uploaded PDF documents, character name identification, and sentence similarity assessment utilising sophisticated Natural Language Processing model.

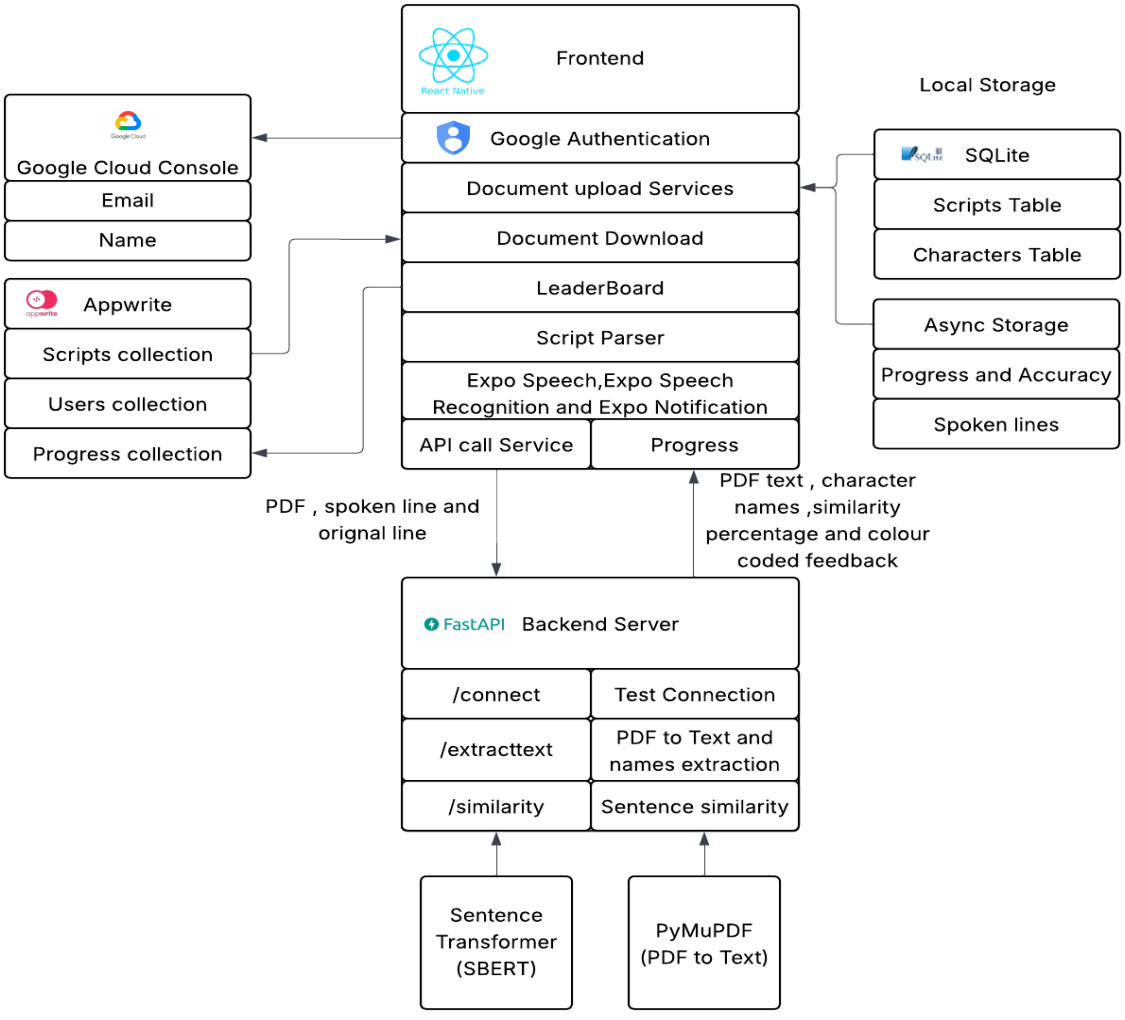


Figure 3: Overall Architecture Diagram of the System

The system follows a **mobile-first architecture** designed for offline-first operation, with optional cloud support for advanced features.

1.Client Side (React Native Expo App)
The client-side architecture involves the **React Native Expo** app, which acts as the interface between the user and the backend server. Key components include:

- **User Interface (UI)**
  - Handles user interactions such as script selection, character settings, practicing lines, and receiving feedback.
  - Displays scripts and character dialogue in a WhatsApp-style chat bubble format.
  - Highlights the current line and provides controls for playback, speech recognition, and settings.
- **Google Sign-In Authentication**
  - Uses Google Sign-In to authenticate users securely and simplify account management.
  - On successful sign-in, user data is synced with Appwrite for personalized experience, progress tracking, and secure access to private scripts.

- **Speech Recognition**
  - Converts the user's speech into text using **Expo Speech Recognition**.
  - Tracks real-time progress and compares spoken lines with the original script.
  - Actively listens during rehearsal, detects silence to auto-advance lines, and compares spoken text to the original using similarity scoring.

- **Playback and Feedback Display**
  - Provides real-time feedback on the user's spoken lines, such as sentence similarity scoring.
  - Displays results like "correct" or "incorrect," and automatically progresses to the next line once criteria are met (e.g., similarity threshold or silence detection).

- **Local Storage/Progress Tracker**
  - Stores local data like Progress and accuracy using **AsyncStorage**.
  - Uses SQLite for local script management and character voice settings.

- **Online Storage using Appwrite**
  - User Management: Handles authenticated user-specific data like name, email, scripts and rehearsal history.
  - Database Storage: Stores scripts, characters, and rehearsal progress in structured Appwrite collections.

- **Communication with Server**
  - Makes HTTP requests (using **Axios**) to the FastAPI backend for operations such as fetching scripts, extracting character names and processing feedback.

2. Server Side (FastAPI Backend)
The server side is built with python FastAPI, which handles requests from the client-side app. The backend architecture includes the following components:

- FastAPI Application
  - Handles HTTP requests, routing, and integrates with other backend components. CORS middleware is enabled on the server to allow requests from any origin.
  - Provides endpoints to handle script retrieval, line-by-line feedback, and storing user progress.
  - Integrates AI models to compare spoken lines with the script and provide feedback on accuracy.
  - Uses Sentence Transformers (bert-base-nli-mean-tokens) for semantic similarity

- API Endpoints
    - Provides APIs for interacting with the client-side app, including:
    - **POST /similarity**: Submit feedback for a given line based on speech recognition.
    - **POST /extracttext**: Extract text from PDF and character names and send them as response.

3. Communication between Client and Server

- HTTP Requests: The client communicates with the server via HTTP requests, typically using RESTful APIs (with JSON payloads). Common interactions include:
- Fetching scripts and rehearsal data.
- Submitting speech input for real-time processing and feedback.

4. Data Flow

- **Client to Server**:
    - Request sent to the server for sentence similarity scoring.
    - Request sent to the server for extracting the Text and character from the uploaded script.

- **Server to Client**:
    - Server sends script text data and character list to the client.
    - Similarity percentage along with the color coded feedback is sent back.

- **Appwrite Data Sync:**
    - The client syncs user data (e.g., rehearsal progress, scripts, user details) with Appwrite's database.
    - Rehearsal metrics are stored or updated in the appropriate Appwrite collections.

4.3 Technology Stack

In the development of the AI-driven script rehearsal application, a diverse array of technologies was chosen to guarantee scalability, efficiency, and a smooth user experience. The technologies employed encompass the frontend, backend, and database management, with each fulfilling a distinct function within the comprehensive system architecture. The chosen technologies for this initiative comprise React Native Expo, FastAPI, SQLite, Speech Recognition, and Sentence Similarity Scoring.

Layer	**Technology**	**Purpose**
Frontend	React Native (Expo)	Mobile App Development, develop services like Script parser to extract dialogue, actions and description from scripts
Speech API	Expo Speech & Expo Speech Recognition	Text-to-Speech (TTS) and Speech-to-Text (STT) functionalities
Local Storage	AsyncStorage and SQLite	Store user progress, Script data, character list.
Online Storage	Appwrite	Store user progress, Script data, character list and user details.
Authentication	Google cloud Console	For user authentication using Google sign in.
Backend	FastAPI (Python)	Sentence similarity computation using pre-trained SBERT models, Extract text from PDF, Extract character names from scripts

Charts	React native charts	Display bar charts for visualizing user progress
State Management	React Native Hooks/Context	Manage app state across components

Table 1: Technology stack list

1. **React Native Expo (Frontend)**

- React Native, an open-source framework developed by Facebook, enables developers to create mobile application that function across multiple platform using JavaScript and React. This framework facilitates the creation of applications that function smoothly on both iOS and Android platforms, eliminating the necessity for distinct codebases for each. Expo comprises a collection of tools designed to enhance React Native, offering a user-friendly framework for the development and deployment of React Native applications [37].

- Role in the App:
  - User Interface (UI): React Native Expo allows for the creation of a responsive, mobile-friendly UI that displays scripts, tracks progress and manages user interactions such as character selection and playback of lines.
  - Speech Recognition Integration: Expo's integration with libraries like Expo Speech Recognition [26] allows the app to capture user speech, convert it into text, and compare it to the script.

2. FastAPI (Backend)

- FastAPI is an advanced and high-efficiency web framework developed for the purpose of constructing APIs using Python 3.7 and later, leveraging standard Python type hints. It is recognized for its rapid execution and user-friendly interface, facilitating automatic data validation, documentation, and serialisation. FastAPI is constructed upon Starlette for web functionalities and Pydantic for data validation.

- Role in the App:
  - Backend API: FastAPI handles all the API requests from the client-side application. It processes requests related to extracting text from PDF files, sentence similarity checking, character extraction from script text.

3. SQLite (Database)

- SQLite is a relational database engine that operates without a server, is self-sufficient, and requires no configuration [40]. It is particularly well-suited for embedded systems or applications that require the storage and querying of small to moderate volumes of data.

- Role in the App:
  - Data Storage: SQLite stores the user progress, rehearsal data including script file, text, character name list and their voice settings ensuring that data can be easily accessed and updated by the app during rehearsals.
  - Offline Access: Since SQLite is a lightweight database, it allows the app to work offline.

4. Speech Recognition

- Speech recognition technology facilitates the transformation of verbal language into written text [41], [7]. In this application, speech recognition is employed to record the actor's dialogue during practice sessions and juxtapose it with the original script.

- Role in the App:
  - Voice Input: The app listens to the user's speech input, converting it into text using speech recognition libraries like Expo Speech Recognition [41].

- Feedback Generation: This technology is essential for providing real-time feedback on pronunciation accuracy, line timing, and sentence similarity.

5. Sentence Similarity Scoring

- Sentence Similarity Scoring is a method employed to assess the degree of similarity between two textual segments [43], [44]. In this application, it is utilised to juxtapose the transcribed lines spoken by the actor against the original script, offering insights into the extent of their correspondence.

- Role in the App:
  - Feedback Accuracy: Sentence similarity scoring helps the system assess how well the user has delivered the lines, accounting for minor differences in phrasing.

6. Appwrite

- Appwrite is an open-source backend server that provides essential backend services such as database management, file storage, and serverless functions for web and mobile applications [42].
- Role in the App:
  - Database: Appwrite is used to store structured data such as scripts, rehearsal progress, user details and character list in secure collections.

4.4 Database Design

The application uses a hybrid storage architecture that combines AsyncStorage, SQLite, and Appwrite to manage user data across offline and online contexts. This ensures smooth user experience, offline rehearsal capabilities, and centralized data management when connected to the internet.

4.4.1 Local data storage using SQLite

The AI Script Rehearsal App uses a local SQLite database named localdb to manage and store rehearsal data efficiently. The database consists of two primary tables:

- scripts: Stores information about uploaded script files and user-selected characters.
- characters: Stores associated characters for each script, including optional voice settings.

The structure has been designed to ensure fast retrieval, update operations, and extensibility for future features such as advanced voice customisation and rehearsal tracking.
The Figure 4 below shows entity relationship diagram of both the primary tables.

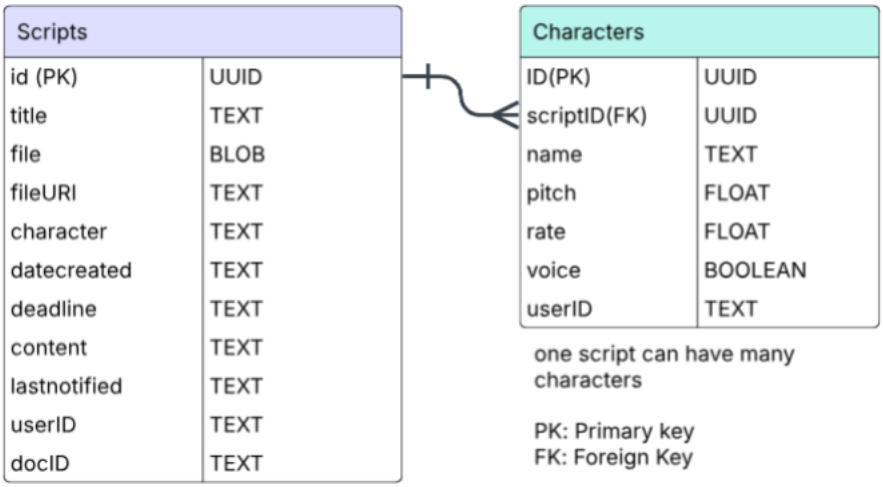


Figure 4: SQLite Database ER Diagram

Column Name	Type	Description
id	UUID (PK)	Unique identifier for each script.
title	TEXT	Title of the script.
file	BLOB	Binary storage for the script file.
fileURI	TEXT	Local URI to the file location.
content	TEXT	Parsed text content from the file.
character	TEXT	Name of the user-selected character.
dateCreated	TEXT	Timestamp when script was created.
deadline	TEXT	User-defined deadline for rehearsal.
lastnotified	TEXT	To track the last notification sent date
userID	TEXT	Document ID of the newly created user
docID	TEXT	Document ID of the newly created script

Table 2: Field Description of scripts Table

Column Name	Type	Description
id	UUID (PK)	Unique identifier for each character.
scriptID	UUID (FK)	References the script to which it belongs.
name	TEXT	Character name.
pitch	FLOAT	Custom pitch setting for the voice.
rate	FLOAT	Custom rate setting for the voice.
voice	TEXT	Stores the character voice type
userID	TEXT	Document ID of the newly created user

Table 3: Field Description of Characters Table

4.4.2 Local data storage using AyncStorage

AsyncStorage is utilised to handle simple local data persistence. AsyncStorage is a key-value storage system available in React Native that allows the app to store small amounts of data on the device, even after the app is closed or restarted [45]. This solution was chosen over heavier options like SQLite because the amount of data is small, the structure is simple.

- In this project, AsyncStorage is used primarily to:
- Store the user's rehearsal progress, including Spoken lines and original lines.
- Progress percentages over the last five rehearsal attempts.
- Enable quick access to recent rehearsal results without requiring an internet connection.

By leveraging AsyncStorage, the application ensures that users can track their performance history smoothly and efficiently without relying on a backend server for every session. This provides an offline-first experience, which is critical for usability in various environments where network connectivity may be inconsistent.

4.4.3 Online data storage using Appwrite

Provides cloud-based persistence and synchronisation for scripts, progress tracking, and user data.

- **Scripts:** Uploaded scripts are stored in Appwrite's database for cloud backup and public sharing.
- **Leaderboard Metrics:** Only the most recent rehearsal progress and sentence accuracy are uploaded to Appwrite when online. This data is used to rank users for a specific script on the public leaderboard.

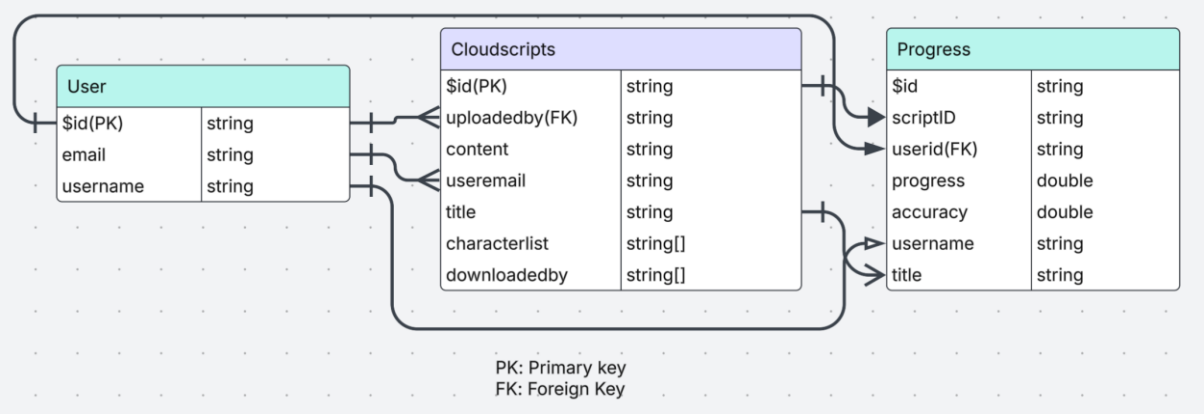


Figure 5: Appwrite collection relationship Diagram

The database is composed of three core collections:

- Users
- CloudScripts
- Progress

Each collection stores structured documents. Below is a detailed explanation of each collection's schema and their relationships.

4.4.3.1 User Collection

Field	Type	Description
$id	string	Primary key- Unique user ID.
email	string	User's email address.
username	string	Display name or alias of the user.

Table 4: collection schema of user collection

- User collection fields are referenced in the cloudscripts.uploadedby, cloudscripts.useremail, and progress.userid fields.

4.4.3.2 cloudscripts Collection

Field	Type	Description
$id	string	**Primary key**. Unique script document ID.
uploadedby	string	**Foreign key**. Refers to user.$id of the script uploader.
useremail	string	Redundant info: email of uploader (used to simplify lookups and avoid joins).

title	string	Title of the script.
content	string	Full script text content.
characterlist	string[]	List of character names involved.
downloadedby	String[]	Foreign key array**.** List of user.$ids who have downloaded the script.

Table 5: collection schema of cloudscripts collection

4.4.3.3 User Collection

Field	**Type**	**Description**
$id	string	**Primary key**. Unique progress document ID.
scriptid	string	**Foreign key**. Links to cloudscripts.$id.
userid	string	**Foreign key**. Links to user.$id of the person rehearsing.
username	string	Cached username of the person rehearsing.
title	string	Title of the script being rehearsed (redundant for quick lookup).
progress	double	Rehearsal progress (e.g., percent completion).
accuracy	double	Accuracy metric for how well the user performed.

Table 6: collection schema of progress collection

- **Redundancy for Performance**: Fields like useremail and username are stored redundantly in cloudscripts and progress to reduce the need for joins when querying, improving performance on mobile devices.

- **Use of Arrays**: downloadedby is stored as an array to quickly identify which users have downloaded a script.

4.5 Backend Design

The backend architecture for this initiative was constructed utilising FastAPI, a contemporary web framework in Python recognized for its rapid performance, development simplicity, and automatic generation of documentation. This backend is tasked with managing client requests, processing uploaded PDF documents, extracting pertinent information, and delivering structured responses for integration within the mobile application.

4.5.1 Technology Stack

Framework: FastAPI
Programming Language: Python 3.11

- Libraries:
  - PyMuPDF (fitz): For reading and extracting text from PDF documents.
  - Pydantic: For data validation and serialization.
  - re (Regular Expressions): For text processing and pattern matching.

    - Sentence Transformers (BERT-based): for comparing two sentences and to find the similarity.
- Middleware:
    - CORS Middleware: Configured to allow cross-origin requests, enabling seamless communication between the frontend (React Native app) and the backend server.

4.5.2 API Endpoints

The backend exposes two primary RESTful API endpoints:

1. **POST /connect**
    - Purpose: Establishes a basic connection to verify the server is reachable.
    - Request Body: Accepts a JSON object containing a name field.
    - Response: Returns a simple message confirming the connection with the provided name.

2.**POST /extracttext**

- Purpose: Accepts a PDF file upload, extracts the text content, and identifies potential character names from the script.
- Request: Multipart/form-data containing a PDF file.
- Processing Steps:
- Reads the uploaded PDF file into memory.
- Extracts textual content from the PDF using PyMuPDF.
- Parses the extracted text to detect possible character names based on scriptwriting conventions (e.g., uppercase names before dialogues, names ending with a colon).
- Response: Returns the extracted text along with a list of detected character names.

4.5.3 Text and Character Extraction Logic

The backend implements custom logic for parsing scripts, designed specifically for typical screenplay formatting:

- **Text Extraction:**
    - Utilises PyMuPDF to iterate through all pages in the PDF and aggregate their text content.
- **Character Name Detection:**
    - Names are identified based on formatting rules:
    - Lines entirely in uppercase.
    - Lines ending with a colon (:) that are in uppercase.
    - Filtering includes:
    - Ignoring lines longer than 40 characters.
    - Excluding common scene headings (e.g., "INT", "EXT", "DAY", "NIGHT").
    - Allowing only one or two-word names.
    - Removing artifacts like "(CONT'D)" that commonly appear in screenplays.

  This lightweight heuristic method offers high accuracy for typical English-language scripts, enabling reliable automatic extraction of dialogue characters for rehearsal purposes.

4.3.4 Sentence Similarity API

The backend includes a dedicated /similarity endpoint designed to compute the semantic similarity between two sentences. This functionality supports real-time performance feedback during script rehearsals.

When a POST request is made with two sentences, the server follows these steps:

- **Sentence Embedding:**
  The input sentences are encoded into high-dimensional vectors using a pre-trained **Sentence-BERT (SBERT)** model [4]. SBERT generates dense sentence representations that capture semantic meaning beyond simple keyword matching.

- **Cosine Similarity Computation:**
  Cosine similarity measures the angle between the two sentence vectors, calculating their semantic closeness. The formula is:
- **Similarity Percent Scaling:**
  The cosine similarity value (ranging from 0 to 1) is scaled into a percentage to improve user interpretation.

- **Colour-Based Feedback:**
  Depending on the similarity percentage, a color code is returned:
  - Green: similarity ≥ 90%
  - Orange: 70% ≤ similarity < 90%
  - Red: similarity < 70%

This colour coding enables immediate visual feedback for users during rehearsals.

- **Error Handling:**
  - Robust exception handling ensures the API returns informative error messages in case of failure during encoding or computation.
  - Thus, the Sentence Similarity API forms a critical part of the backend system for providing automated, real-time rehearsal evaluation.

4.5.4 Middleware and Security

- **CORS Configuration:**
  - Configured to allow all origins ("*"), all methods, and all headers.
  - This ensures that the mobile application can interact with the backend without encountering cross-origin request errors, simplifying development and deployment.
- **File Handling:**
  - The backend directly processes the uploaded files in memory without persisting them to disk, reducing storage overhead and enhancing security.

4.6 Frontend Design

The wireframes were created using Figma to visualize the app's structure, navigation flow, and user interaction with key features such as script upload, script playback, progress tracking, and settings management.

4.6.1 Wireframe Overview

The Figure 6 below shows the different wireframe screens that’s been designed in Figma.

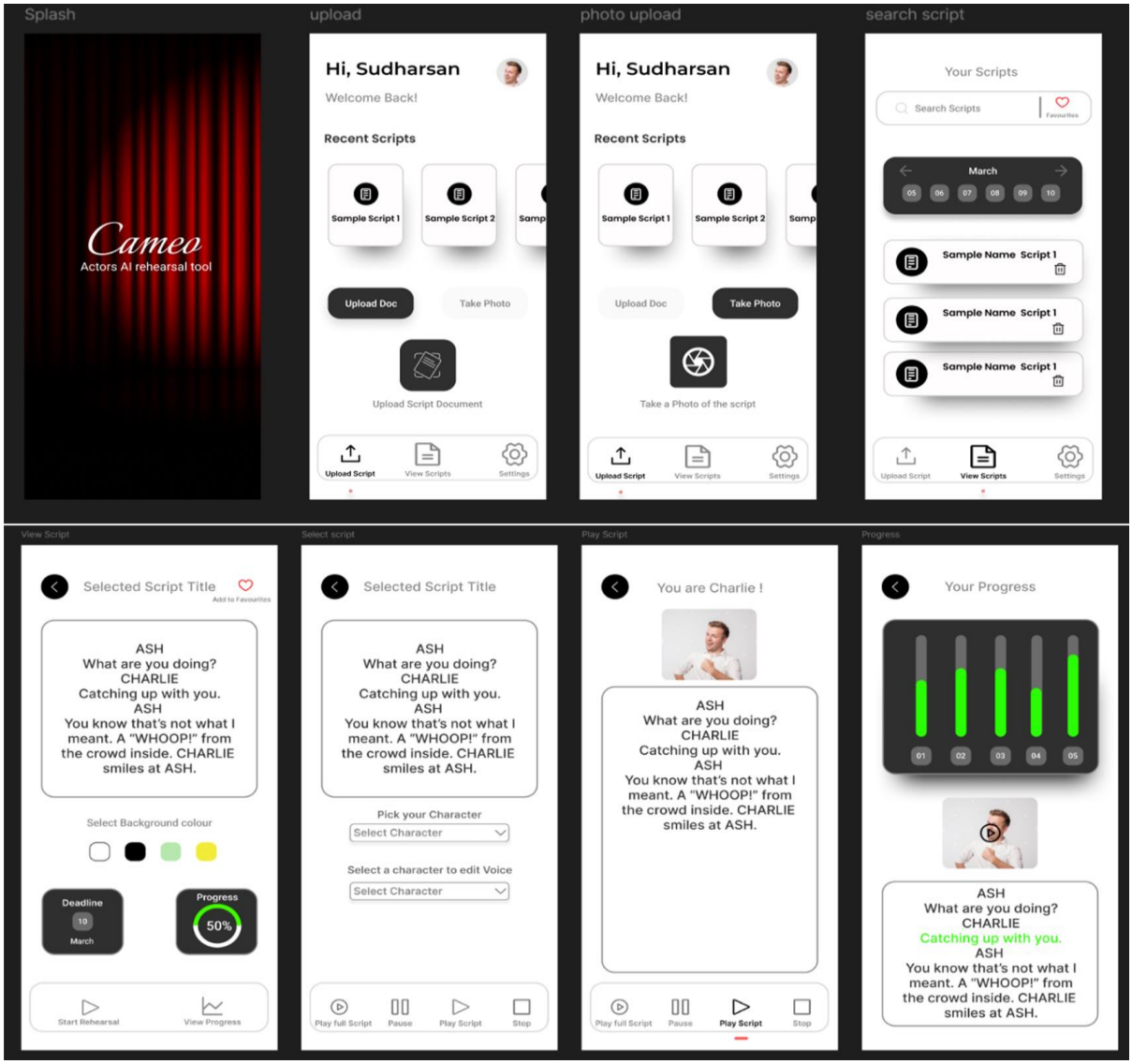


Figure 6 : Wireframe Screen design

The wireframe consists of the following screens:

- Splash Screen
  - Displays the app name ("Cameo") and tagline ("Actors AI rehearsal tool") with a visual background of theatre curtains, setting the theme of acting and performance.
- Upload Screen
  - Greets the user personally ("Hi, Sudharsan").
  - Shows recent scripts in a horizontally scrollable format.
  - Options to upload a document or take a photo of a physical script.
  - Navigation bar with three options: Upload Script, View Scripts, Settings.
- Photo Upload Screen
  - Similar layout to Upload Screen but focuses on script upload by taking a photo.
- Search Scripts Screen
  - Allows users to search for existing scripts by name.
  - View favorite scripts and browse scripts by date (calendar layout).
  - Lists available scripts with easy access for rehearsal.

- View Script Screen
    - Displays the selected script with character dialogues.
    - Option to set background color for better readability.
    - Button to start script rehearsal and a progress tracker showing completion percentage.
- Select Script Screen
    - Enables users to pick a character to rehearse.
    - Users can also select a character to edit voice parameters (e.g., pitch, rate).
- Play Script Screen
    - Displays the script during rehearsal.
    - Buttons to Play Full Script, Pause, Play Script, and Stop rehearsal session.
- Progress Screen
    - Shows a bar chart of user's past 5 rehearsal attempts
    - Displays user's spoken dialogues compared to original script, highlighting spoken and missed words.

5. Design Implementation

5.1 Backend Design Implementation

The application's backend is developed with FastAPI, a contemporary web framework designed for API creation in Python. It is responsible for processing PDF scripts, extracting character names, and assessing the similarity between spoken lines and their original counterparts through Natural Language Processing (NLP). This backend operates in a stateless manner, refraining from data persistence as it does not utilise a database, making it particularly suitable for server less or ephemeral function environments.

5.1.1 Server Setup

The FastAPI server is set up with the following configurations:

```
5   app = FastAPI()
6
7   app.add_middleware(
8       CORSMiddleware,
9       allow_origins=["*"],  # Allow all origins
10      allow_credentials=True,
11      allow_methods=["*"],
12      allow_headers=["*"],
13  )
```

Listing 1: Fast API setup.

CORS Middleware:
Configured to allow cross-origin requests, enabling frontend-backend communication regardless of domain.

5.1.2 API endpoints

1. Simple connection test - /connect:
    - Verifies client-server connection and quick test of backend availability.
    - Takes a JSON body with name field as input and outputs a confirmation message back to the client.

2. **PDF Upload, Text Extraction, Character Detection - /extracttext:**

```
@app.post("/extracttext")
async def extract_text(file: UploadFile = File(...)):
    """API endpoint to extract text from a Base64 PDF string."""
    try:

        pdf_bytes = await file.read()
        extracted_text = extract_text_from_pdf(pdf_bytes)
        character_names = extract_character_names(extracted_text)

        return {"text": extracted_text, "characters":
character_names}
```

Listing 2: Extract text endpoint

- extract_text_from_pdf(pdf_bytes) : This function uses fitz from PyMuPDF library to extract the text from the input PDF data which is a BASE64 string.
    - Extract the text by iterating through the pages and then joins them line by line.
    - The function returns the extracted text which is then passed on to the next function and added to the response body.
- extract_character_names(extracted_text) : The purpose of this function is to seperate the characters name that are involved in the script and who has dialogue to speak.
    - Takes the extracted text as input and outputs an sorted list of character names.
    - Since scripts have various formatting conventions, this function intelligently uses regular expressions, structural patterns, and filtering rules to identify names.
    - A set variable is declared to store the characters names in order to avoid duplicate names thus prevents from adding names multiple times.
    - As part of pre-processing the entire script(raw text) is split into lines because character names usually appear line-by-line in a script. An iterator is used to process line by line. The whitespaces are removed from each line. The line is checked for long string length (which are likely descriptions or dialogues, not names).and empty line in that case it's ignored.
    - Character name detection: we have two formats of character naming styles in a script

      1.**Colon style followed by dialogue**: This is frequently utilised in casual script drafts, transcripts, or scenes that are rich in dialogue.
        - Example : "Rob: how are you?"

      2. **All caps and the next line is a dialogue:** Character names are frequently presented in uppercase letters and are usually succeeded by the dialogue of the character on the subsequent line. [41].
        - Example : "Rob
          How are you?"

1. **Colon Style:**
    - The code detects this pattern using a regular expression.
    - This pattern matches lines that begin with uppercase letters, possibly containing spaces, hyphens, or apostrophes (e.g. Rob-JR), and ending with a colon. If such a line is matched, the part before the colon is extracted as the character name using split function.

```
        #COLON style example NAME:
        if re.match(r"^[A-Z][A-Z\s\-']{1,30}:", line_stripped):
            name = line_stripped.split(":")[0].strip()
```

Listing 3 : Regex for Colon style

2. ALL CAPS Style:

The following condition should be used to extract the character names for all caps:

- Checks if the current line is in uppercase using the .isupper() function, Thus the below regex was used which satisfy the above rules. which is a strong indicator that it may be a character name. To further validate this assumption, the code then checks if there is a following line. It continues by confirming that the next line is not empty and, importantly, that the next line is not also in uppercase. since uppercase lines typically indicate scene headings or other metadata. If all these conditions are true, the current line is likely to be a character name, and it is stored in the set() variable.

```
- line_stripped.isupper()
-             and i + 1 < len(lines)
-             and lines[i + 1].strip()
-             and not lines[i + 1].strip().isupper()
```

Listing 4: All Caps character names extraction logic

- **Normalizing Character names:** After a potential character name is identified, it undergoes normalization to remove extra noise or formatting artifacts. The first regular expression removes common screenplay annotations like (CONT'D) or (CONT), which indicate that the character is continuing their dialogue from an earlier line. Again with the use of a regular expression it strips out any non-alphanumeric characters—except spaces, hyphens, and apostrophes—to ensure the name is clean and human-readable(eg. ROB (CONT'D)). This process is essential for accurate storage and later comparisons, especially when names are matched or clustered.
-

```
• name = re.sb(r"\s*\(?(CONT'?D|CONT)\)?", "", name,
  flags=re.IGNORECASE)
•             name = re.sub(r"[^\w\s'-]", "", name).strip()
```

Listing 5: Normalizing character name regex

- **Filtering scene direction and description words:** This segment of the logic is designed to filter out false positives, ensuring that scene directions or script headers are not mistakenly identified as character names. Initially, the code restricts names to one or two words, as genuine character names like JOHN, MR. BOND, or DR. SMITH typically conform to this length. Names exceeding this limit are likely to be misidentified directions, sentences, or script titles. Furthermore, the code discards names that include recognized scene formatting keywords such as INT, EXT, ACT, etc., which are integral to screenplay structure for denoting scene locations and times. These terms do not represent character names but are essential metadata in screenwriting and must be excluded. The complete code snippet is in appendix A.1.

3. Spoken Line Similarity Check- /similarity: The /similarity endpoint represents a FastAPI route intended to assess the semantic similarity between two user-provided sentences. When a POST request is received with a JSON payload that includes sentence1 and sentence2, the endpoint employs the Sentence Transformer model (bert-base-nli-mean-tokens) to transform both sentences into dense vector embeddings.The Sentence Transformer is preloaded at start-up to avoid repeated initialization and speed up performance. The embeddings encapsulate the contextual and semantic significance of each sentence. The similarity between the two generated vectors is subsequently assessed through cosine similarity, a metric that evaluates the cosine of the angle formed between two vectors in a multidimensional space, yielding a value ranging from -1 to 1. This score is then converted into a percentage and rounded to two decimal places for

clarity. Furthermore, the endpoint classifies the similarity into three categories utilising a colour-coding system.

- Green: Similarity ≥ 90% (High similarity)
- Orange: 70% ≤ Similarity < 90% (Moderate similarity)
- Red: Similarity < 70% (Low similarity)

This color-coded feedback offers a clear comprehension of the degree of similarity between the two sentences. The Appendix A.2 has full code snippet for the feedback logic.

```
vectors = similarity_model.encode([data.sentence1, data.sentence2])
        similarity = cosine_similarity([vectors[0]], [vectors[1]])[0][0]
        similarity_percent = float(round(similarity * 100, 2))
```

Listing 6: Similarity check endpoint

5.2 Database Design Implementation

The system uses a hybrid database approach combining local SQLite and Appwrite (a cloud backend) to manage script data efficiently.

5.2.1 SQLite Local Database Implementation

A lightweight client-side database is implemented using **SQLite**, specifically tailored for mobile or local-first applications using **Expo SQLite**. The database stores two main entities: scripts (uploaded documents) and characters (associated with each script)

The below table consists of different Async functions that's been used.

Method	**Purpose**
execAsync()	Executes raw SQL statements, often for setup/configuration (e.g., schema creation).
runAsync()	Executes SQL commands that **modify data** (INSERT, UPDATE, DELETE). Returns metadata like lastInsertRowId.
getAllAsync()	Fetches multiple rows of result (used for SELECT queries returning multiple rows).
getFirstAsync()	Fetches the first row of a SELECT result.

Table 7: Different Async functions and purpose

Different Methods are implemented to perform create, update, delete and read operation on Scripts and characters table.

1. createDatabase():
This function initializes the local SQLite database named "localdb" and sets up two primary tables: scripts and characters. It ensures that the schema is created only if it doesn't already exist using CREATE TABLE IF NOT EXISTS. The scripts table holds content of the uploaded scripts, including title, file, content, and deadline. The characters table links characters to specific scripts and stores their voice attributes (pitch, rate, voice). It uses PRAGMA journal_mode = WAL to enhance concurrency and performance. Errors during setup are logged using console.error. The Appendix B.1 can be referred for full code snippet for database and table creation.

```
await db.execAsync(
            `
            PRAGMA journal_mode = WAL;
            CREATE TABLE IF NOT EXISTS scripts (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            userID TEXT,docID TEXT,createdby TEXT,title TEXT,file BLOB,
            fileURI TEXT,content TEXT,character TEXT,
            lastnotified TEXT DEFAULT CURRENT_TIMESTAMP,
            dateCreated TEXT DEFAULT CURRENT_TIMESTAMP,
            deadline TEXT DEFAULT CURRENT_TIMESTAMP);`

        );
```

Listing 7: Script table creation

2. insertScript():

This function inserts a new script entry into the scripts table. It accepts metadata such as title, file blob, URI and content. It also receives docID and createdby fields which are optional as it is set to NULL by default. These values are overridden when a script is downloaded from the public database. The userID is fetched asynchronously to associate the script with a user so that when retrieving only the scripts uploaded by the current user are displayed. After insertion, it retrieves and logs all script rows for debugging, and returns the newly inserted script's ID.

```
const insertedScript=  await db.runAsync(
        `INSERT INTO scripts (userID, title, file, fileURI, content, docID,
createdby)
       VALUES (?, ?, ?, ?, ?, ?, ?)`,[userID, title, fileBlob, fileURI,
content, docID, createdby] );
```

Listing 8: Insertscript function SQL command

3. insertCharacters():

This function adds character entries to the characters table for a given script. It iterates over a list of character names, normalizes pitch and rate as floats, and ensures entries are not duplicated using INSERT OR IGNORE. It links each character with a specific scriptID and userID, making it easy to retrieve and manage them per script.

```
await db.runAsync(
 'INSERT OR IGNORE INTO characters ( userID, name, scriptID, pitch, rate )
VALUES (?,?,?,?,?)',userID, name, scriptID, parsedPitch, parsedRate);
```

Listing 9: Insertcharacters function SQL command

4. updatecharactername()

This function updates the character field in the scripts table for a given script ID. It's useful for associating a primary character or set of characters with a script post-processing.

```
const result = await db.runAsync('UPDATE scripts SET character = ? WHERE id =
?', character, scriptID);
```

Listing 10: updatecharactername function SQL command

5. updateCharacterVoice()

This function modifies the voice parameters—pitch, rate, and selected voice—for a specific character in a script. It locates the character by both scriptID and name, then updates the corresponding row. This is crucial for personalized text-to-speech or voice synthesis workflows tied to specific characters.

```
await db.runAsync(
'UPDATE characters SET pitch = ?, rate = ?, voice = ? WHERE scriptID = ? AND
name = ?', pitch, rate, voice, scriptID, name );
```

Listing 11: updatecharactervoice function SQL command

6. getAllCharacters()

This function fetches all characters associated with a given scriptID. It queries the characters table and returns an array of matching character objects. If the query fails, it catches the error and returns an empty array, maintaining system stability.

```
const results = await db.getAllAsync(
            'SELECT * FROM characters WHERE scriptID =?',scriptID );
```

Listing 12: getAllCharacters function SQL command

7. getAllScripts()

This function retrieves all script entries for the currently authenticated user. It uses the userID to filter results, ensuring that users can only access their own scripts. Retrieved rows are logged for debugging.

```
const result = await db.getAllAsync("SELECT * FROM scripts WHERE userID = ?",
userID);
```

Listing 13: getallscripts function SQL command

8. deleteScript()

This function deletes a specific script from the database based on its id. After deletion, it retrieves and logs all remaining script entries to confirm the deletion took place**.**

```
await db.runAsync("DELETE FROM scripts WHERE id = ?;",id);
```

Listing 14: deleteScript function SQL command

9. getScriptById()

This function fetches a single script row based on a given ID. It is typically used for viewing a specific script.

```
const result= await db.getFirstAsync("SELECT * FROM scripts WHERE id =?;",id);
```

Listing 15: getScriptById() function SQL command

10. updateScriptDeadline()

This function updates the deadline field for a specific script. It's useful reminders and scheduling notification purposes within the app. It uses a parameterized query for security and returns the result of the update operation.

```
return await db.runAsync(
      'UPDATE scripts SET deadline = ? WHERE id = ?',deadline, id);
```

Listing 16: updateScriptDeadline() function SQL command

11. updateLastNotified()

The updateLastNotified function is responsible for updating the lastnotified timestamp of a specific script entry in the scripts table. This is useful for tracking the notification and processing event related to a script. The function retrieves the current timestamp using JavaScript's Date().toISOString() method to ensure a standardized format compatible with SQLite. It then performs an asynchronous UPDATE operation using runAsync. (see Appendix B.2)

```
const currentTimestamp = new Date().toISOString();
    await db.runAsync(
      `UPDATE scripts SET lastnotified = ? WHERE id = ?`,
      [currentTimestamp, scriptId]    );
```

Listing 17: updateLastNotified() function SQL command

Each of these functions is designed with asynchronous operations, error handling, and logging in mind—ensuring reliability, traceability, and user-focused data flow in your local SQLite implementation. The entire code for the all functions can be viewed in Appendix B.3

5.2.2 Appwrite online Database Implementation

The integration of the Appwrite online database functions as the foundational element for cloud-based storage, synchronisation, and collaborative efforts within the application. It allows users to securely store, manage, and access scripts and associated data across various devices. The system facilitates critical operations through a series of well-defined functions, including the uploading and downloading of scripts, management of user accounts, monitoring of script usage, and tracking of user progress. By connecting local storage to a centralised cloud backend, this implementation guarantees data consistency, accessibility for multiple users, and real-time interaction, thereby offering scalability and offline-first capabilities for an enhanced user experience.

The following functions are used to perform the essential operations:

1**. getcloudscripts()**
This function retrieves all documents from the cloudscripts collection in the Appwrite database. It uses database.listDocuments to fetch scripts stored on the cloud, enabling users to view publicly shared or previously uploaded scripts

```
const {documents, total} = await
database.listDocuments(config.db,config.col.cloudscripts)
```

Listing 18: getcloudscripts() function to retrieve Cloud Scripts

2. adduser()
The adduser function checks whether a user with a given email already exists in the Appwrite user collection. If the user exists, it returns their document ID. Otherwise, it creates a new user document with the provided email and username. For performance and offline capabilities, the user ID is cached locally using AsyncStorage. The complete code for adding a user is attached in Appendix B.4.

```
const existing = await database.listDocuments(
      config.db,config.col.user,[Query.equal("email", email)] );
```

Listing 19: Query to check if the user already exists

```
onst response = await database.createDocument(
      config.db,config.col.user,ID.unique(),{email,username,});
```

Listing 20: Query to add a new user

3. addscript()
This function adds a new script document to the cloudscripts collection. Before uploading, it checks for duplicates based on useremail, title, and characterlist to prevent redundant entries(see Appendix B.5). If the script is unique, it is saved in the database with metadata such as content, uploader, and characters. The function returns detailed upload status, including the document ID.

```
const response = await database.createDocument(
      config.db,
      config.col.cloudscripts,
      ID.unique(),{ uploadedby,useremail,content,characterlist,title});
```

Listing 21: Query to add a new script if it doesn't exist

4. deletescript()

The deletescript function ensures secure deletion by verifying if the current user is the uploader of the script. If authorized, it deletes the script from the cloudscripts collection and also removes any associated progress data using the deleteprogress function.

```
if (script.uploadedby !== userId) {
      console.log('You are not authorized to delete this script');
      return { status: 'unauthorized', message: 'User does not have permission
to delete this script' };
    }
    const response = await database.deleteDocument(config.db,
config.col.cloudscripts, scriptId);
    const deleteResponse = await deleteprogress(scriptId);
```

Listing 22: Query to delete a script if the user is the uploader

5. updateDownloadedBy()

This function updates the downloadedby array of a script to track users who have downloaded it. It first ensures the user isn't trying to download their own script and prevents redundant entries by checking if the user already exists in the array. If valid, the user ID is appended to the list and saved back to the database. The complete code for updating downloaded by attribute is in Appendix B.6.

```
const updatedScript = await database.updateDocument(config.db,
config.col.cloudscripts, scriptId, { downloadedby: updatedDownloadedBy});
```

Listing 23: Query to add a new user to downloadedby attribute

6. getprogress()

getprogress function fetches progress-tracking documents associated with a specific script from the progress collection. The scriptId is passed as parameter.

```
const { documents, total } = await database.listDocuments(config.db,
config.col.progress, [query]);
```

Listing 24: Query to get the progress of a script

7. updateProgress()

This function either updates an existing progress document or creates a new one based on whether the user has already interacted with the given script. It stores progress percentage and accuracy metrics, which can be used to monitor engagement or performance in real-time. The complete code for updating progress collection is in Appendix B.7.

8. deleteprogress()

deleteprogress removes all progress records linked to a specific script. It first queries the database for documents matching the script ID and deletes each one. This guarantees the complete elimination of all associated tracking data upon the removal of a script from the system.

9. downloadScriptFromAppwriteToLocal()

This function facilitates downloading a script from Appwrite to the local SQLite database. It retrieves the script by ID, inserts it locally using insertScript() function and insertCharacters() function, and updates the cloud record to reflect the new downloader. It integrates both local and cloud databases to support seamless sync and offline access. The complete process for downloading a script from the cloudscrips collection is in Appendix B.8.

5.3 Frontend Design Implementation

The frontend design of the application is centered around a clean, user-friendly interface built with React Native and integrates essential features such as authentication, navigation, and interaction with a cloud-based backend. This section outlines each key screen implemented in the application, explaining its purpose, core functionalities, and any notable libraries or APIs used. For folder structure please see Appendix C.5.

5.3.1 Login Screen

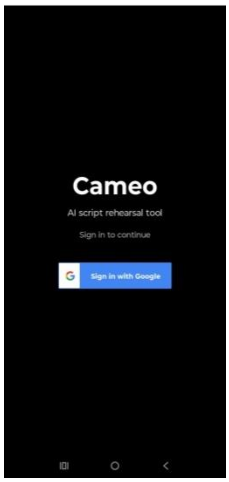


Figure 7: Login screen

The login screen serves as the entry point for user authentication. It leverages the @react-native-google-signin/google-signin library to enable seamless Google-based login. Upon launch, the app checks if a user is already authenticated using GoogleSignin.getCurrentUser() — if a session exists, the user is automatically redirected to the main tab screen (/(tabs) route), providing a smooth and uninterrupted experience. New users are registered in the Appwrite database using the adduser function, and existing users are retrieved using their email.

The Google Sign-In button is rendered using GoogleSigninButton, styled for visibility and consistency with Google's branding. The handleGooglesignin function manages the entire login flow, including service availability checks, response handling, and error messages via Alert. Parameters such as the user's name, email, and photo are passed to the next screen using Expo Router's navigation system, allowing for a personalized experience across the app.

5.3.2 Upload Screen

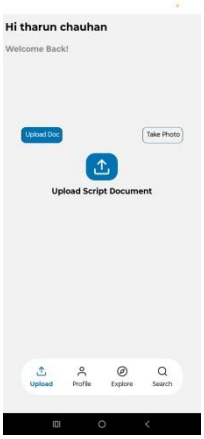


Figure 8: Upload screen to upload script document in pdf format

The Upload screen plays a central role in the app's core functionality by enabling users to import new script content either through uploading documents or capturing photos. It provides a dual-mode interface — users can choose between "Upload Doc" and "Take Photo" options — allowing flexibility in how scripts are input into the system. This is particularly useful for actors, writers, or directors who may have access to scripts in different formats.

Upon first launch, the screen:

- Initializes the local SQLite database using createDatabase()
- Preloads a sample script from the assets folder (PDF format) into the local database using the storeAssetFileAsBlob() method
- Schedules notifications for script reminders using initializeNotifications() and scheduleNotificationsForScripts()

The upload mechanism leverages the expo-file-system and expo-asset libraries to handle reading and writing file data. The script content is extracted and stored along with its characters using insertScript() and insertCharacters() respectively. The pickDocument() function triggers the device's native file picker, and success/failure is communicated via Alert.

5.3.3 Local user uploaded script search Screen

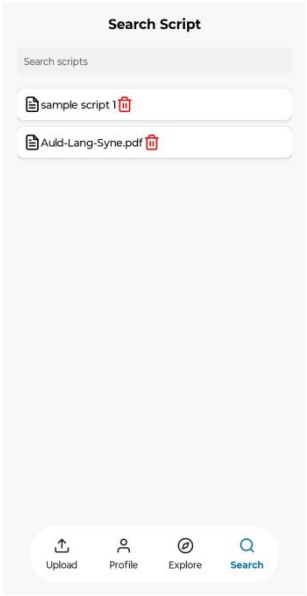


Figure 9: Local user uploaded script search Screen

The Search screen acts as a dynamic script browser and management hub for users, allowing them to easily search, view, and manage their uploaded or sample scripts.Upon entering the screen, the app fetches all stored scripts from the local SQLite database using getAllScripts() and displays them in a scrollable list via FlatList.

The list includes the script title and options to:

- Navigate to the detailed view of the script via Link
- Delete a script using a trash icon, which triggers a confirmation alert

The user can use the search bar at the top of the screen to filter scripts by title. The filtering is case-insensitive and updates in real time using a useEffect hook that listens for changes in the search input. To ensure that the script list is always up-to-date, the screen uses useFocusEffect from @react-navigation/native to re-fetch the script list whenever the screen gains focus (e.g., after a new upload or a return from another tab).

5.3.4 online user uploaded script search Screen

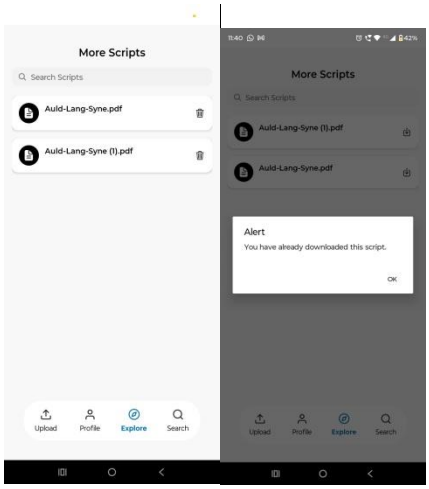


Figure 10: Online user uploaded script search Screen

The Explore screen functions as a discovery hub, showcasing scripts uploaded by other users to the cloud (via Appwrite). This screen allows users to explore, preview, and optionally download scripts to their local storage for offline access and customisation.

It implements the following key features:

- A search bar at the top enables real-time filtering of the fetched scripts based on their titles. This makes it easy for users to find relevant content quickly.
- The screen pulls all available scripts from the cloud using the getcloudscripts() service and stores them in local state. It ensures Only un-downloaded scripts are shown (via filtering with downloadedby.includes(currentUserId)).The currently logged-in user is identified using getCurrentUserID().
- When a user taps Download, the script and its associated character list are saved locally using insertScript() to store the script and insertCharacters() to store the character data.This approach supports offline usage and later editing, enabling a hybrid cloud-local model.
- Scripts uploaded by the current user display a delete button, which calls deletescript() to remove the script from the cloud, enhancing user control over their shared content.

5.3.5 User Profile Screen

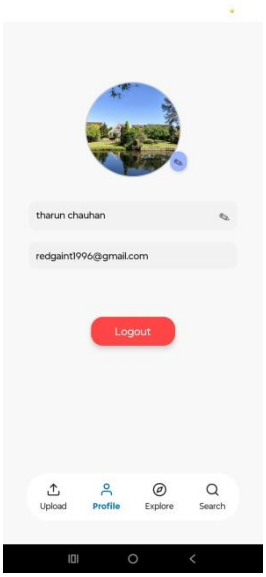


Figure 11: User Profile Screen

The Profile screen is a user-centric view allowing users to view, edit, and manage their personal profile within the app. It combines local storage, third-party authentication (Google), and native features to offer a rich yet intuitive profile experience. Uses @react-native-google-signin/google-signin to retrieve user info on load and handle logout and redirect to the login screen via router.replace('../login'). The user can edit the username and upload a new profile picture if needed.

5.3.6 Script Detail & Character Selection Screen

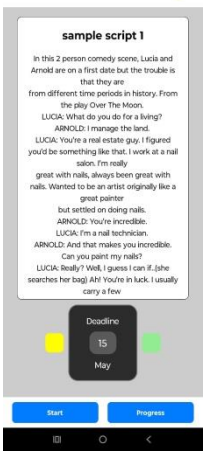


Figure 12: Script Detail & Character Selection Screen

This screen offers users a detailed view of a specific script, along with tools to select a character and set a rehearsal deadline**.**

Renders the selected script title and content inside a scrollable, styled card. The background colour of the scroll view can be changed. The Modal is triggered by the "Start" button opens a picker for character selection the user want to practice. Pulls character names from the database using getAllCharacters(). Selected character is confirmed and passed via router.push to the rehearsal screen. Users can select or change the script's deadline via a date picker. Updates are saved immediately in the database using updateScriptDeadline().

5.3.7 Rehearsal Screen

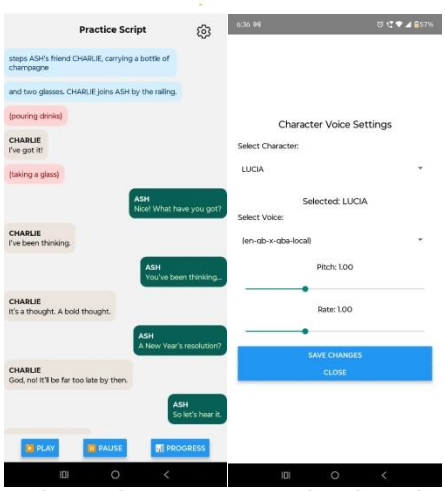


Figure 13: Rehearsal Screen to practice the selected script

The core of the frontend implementation centers around the Rehearsal screen, a dynamic and interactive interface built with React Native and Expo, allowing users to practice their script lines using both speech recognition and text-to-speech technologies. Upon entering the screen, users are presented with a scrollable chat-like view (ScrollView) populated by parsed script lines, each rendered via a custom ChatBubble component. The screen enables real-time dialogue playback, where non-user lines are read aloud using expo-speech, and user lines are validated through voice input using expo-speech-recognition. The user has to score a similarity percentage of greater than 50 to progress to next line(see Appendix C.2). A key feature is the voice settings modal, accessible via a settings icon, where users can configure pitch, rate, and voice for each character using a combination of the Slider, Picker, and expo-speech APIs. The screen tracks user progress and pronunciation accuracy through a similarity comparison feature, which communicates with a backend similarity model via axios (please see Appendix C.1). User rehearsal results are locally saved using AsyncStorage, maintaining a history of the last five attempts. This screen also allows users to pause, resume, and stop rehearsal at any time, and provides visual feedback like "Listening..." indicators and colored similarity results to enhance user experience.

The chat bubble components play a major role in this screen. It represents a single message bubble in a chat interface, commonly seen in messaging apps. The component accepts several props: type, content, character, isUser, read, and onPress. These allow it to dynamically render different types of messages. Specifically, it adjusts the alignment and styling based on whether the message is from the user (isUser) and optionally displays the character's name (character) and a read receipt (double tick) if read is true. The use of TouchableOpacity makes the entire bubble interactive, triggering onPress when tapped.

In order to populate these chat bubbles from screenplay-like content, regex can be used to programmatically extract the following:

- Character names (usually in uppercase and centered or line-start or end with '**:**')
- Dialogues (the spoken lines following the character names)
- Actions (descriptive narrative lines not attributed to any character)

Key Parsing Techniques

- Character names and Dialogues, The parser supports two dialogue formats:
    - Character followed by colon (:):Entire line must contain only capital letters, Spaces, apostrophes, or hyphens, between 1 to 30 characters long and ends with a colon.

```
const colonMatch = line.match(/^([A-Z][A-Z\s'-]{1,30}):/);
```

Listing 25: Extracting Character name with colon

    - ALL caps Character: • Entire line must contain only capital letters spaces, apostrophes, or hyphens between 2 to 30 characters long. And the next line should be all CAPS.

```
const allCapsName = /^[A-Z\s'-]{2,30}$/.test(line);
  const nextLine = i + 1 < lines.length ? lines[i + 1].trim() : '';
```

Listing 26: Extracting ALL CAPS Character name

- Actions or Scene Descriptions: Matches parenthesized instructions like (walks) or Case-insensitive match for common action verbs like begins, enters, exits and holds.

```
if (
      (line.startsWith('(') && line.endsWith(')')) ||
      /(begins|holds|enters|exits)/i.test(line)
    )
```

Listing 27: Extracting Actions from the line

- Scene or description : If no current character then it is consider as scene.
- An object of type ScriptPart is returned.

```
type ScriptPart = {
  type: 'dialogue' | 'action' | 'scene';
character?: string;  content: string;};
```

Listing 28: Return Type from the script parser function

Appendix C.3 can be viewed for complete script parser function

5.3.8 Progress Screen

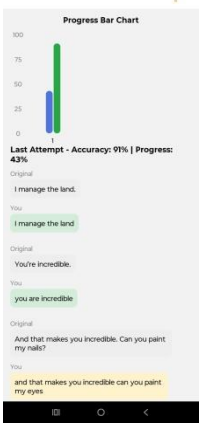


Figure 14: Progress Screen

The ProgressScreen component is a React Native screen designed to display a user's recent rehearsal performance for a specific script. It uses AsyncStorage to retrieve and display the last five attempts associated with a given scriptId, showing both progress and accuracy through a grouped bar chart using the react-native-gifted-charts library. The screen is initialized using React hooks (useEffect, useState) to fetch and process stored results, extract the latest attempt, and format the data into visual and textual elements. For visual clarity, two bar types are rendered per attempt: one for progress (in blue) and one for accuracy (in green), plotted together for easy comparison. Below the chart, a

detailed breakdown of the latest attempt is displayed using custom ProgressBubble components, which compare the original script lines to the spoken lines and highlight differences with color-coded feedback.

5.3.9 View Cloud script Screen

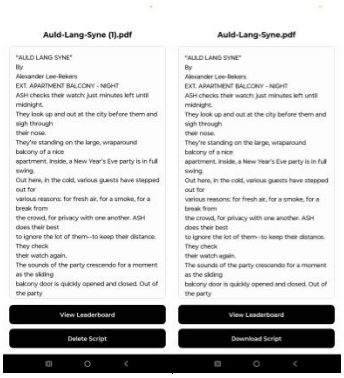


Figure 15: View Cloud script Screen

The ViewScript screen in this React Native application is responsible for displaying the full content of a selected script along with relevant actions like viewing the leaderboard or downloading/deleting the script. Upon mounting, it fetches the script data from Appwrite's database using the document ID passed via route parameters and determines whether the currently logged-in user is the original uploader. The UI consists of the script's title, its full text inside a scrollable view, and action buttons. If the current user is the uploader, a "Delete Script" button is shown; otherwise, a "Download Script" button is provided. Additionally, the user can navigate to a leaderboard screen specific to the script using the "View Leaderboard" button.

5.3.10 Leaderboard screen

Figure 16: Leaderboard screen

The Leaderboard screen displays a ranked list of users based on their performance in a specific script. This screen serves as a motivational tool, encouraging users to improve their scores and engage in friendly competition. It fetches progress data for a given docID (script identifier) and displays each user's progress score alongside their username and rank. Users are sorted primarily by their progress percentage, then by accuracy, and finally alphabetically by username to break any ties. The current user's entry is visually highlighted using a different background color to help them easily identify their position. At the top, the screen also shows the title of the script associated with the leaderboard. The use of FlatList ensures efficient rendering of the ranked data, and useFocusEffect guarantees fresh data is loaded whenever the screen is focused.

```
docs.sort((a, b) => {
  if (b.progress !== a.progress) return b.progress - a.progress;
  if (b.accuracy !== a.accuracy) return b.accuracy - a.accuracy;
  return a.username.localeCompare(b.username);
```

Listing 29: Logic to calculate the ranking

5.4 Notification service Implementation

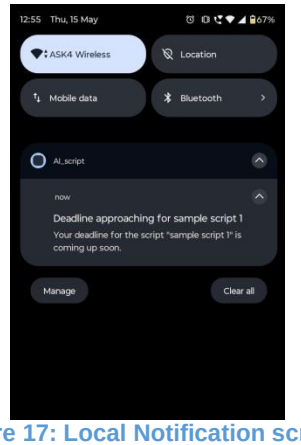


Figure 17: Local Notification screen

Notification service is designed to remind users about approaching script deadlines in a timely, non-repetitive manner enhancing the overall user experience and supporting better rehearsal planning.. It works by first retrieving all stored scripts and checking each one to determine whether a notification should be scheduled. Specifically, it verifies two conditions: (1) the script's deadline is still in the future, and (2) the user hasn't already been notified about it today. If both conditions are met, the service uses Expo's notification API to schedule a local notification, which is triggered 30 later (please see Appendix C.4). The message informs the user that their deadline is near and encourages action. After scheduling, it updates the script's lastnotified field in a local SQLite database to avoid redundant reminders on the same day.

```
if (deadlineDate > currentDate && !DateFns.isToday(script.lastNotifiedDate))
```

Listing 30: Condition check before scheduling a notification

6 Testing and Evaluation

6.1 Frontend Testing

Frontend testing was conducted to ensure that all user-facing features of the app function as intended across various use cases, both online and offline. The testing focused on user interactions, UI behaviour, voice features, data display, and overall app flow. Each feature was tested systematically, and results were compared against expected outcomes. The goal was to verify that the app provides a

smooth, intuitive, and reliable experience across script management, rehearsal playback, speech recognition, similarity scoring, and progress tracking.

Feature	Test Description	Expected Result	Status
Script Upload (Local)	User selects and uploads a local script file	Script is parsed and displayed in rehearsal screen	Pass
Chat Bubble Display	Script lines render as chat bubbles with speaker names	Bubbles show in sequence, styled correctly, and show speaker identity	Pass
Line completion	Double tick is displayed once the line is completed	Double tick visible after line completion	Pass
TTS Playback	Tap-to-play activates character voice with selected pitch and rate	Voice plays as configured	Pass
Speech Recognition	User speaks a line; app listens	Speech is transcribed and matched with expected line	Pass
Similarity Scoring	Feedback for user-spoken line is shown	Score (in %) is displayed and stored	Pass
Progress Tracking	Navigate to progress screen after rehearsal	Shows comparison of spoken vs actual lines + bar chart of 5 attempts	Pass
Character Voice Settings	User adjusts pitch, rate, and voice type	Settings update and reflect in TTS playback	Pass
Offline Mode	User opens app without internet	Local scripts load correctly	Pass
Screen Navigation	Navigate between Home, Settings, Rehearsal, and Progress	Transitions are smooth and accurate	Pass
Cloud Script List Display	User views list of public cloud scripts	All public scripts are listed with relevant details	Pass
Cloud Script Download	User selects and downloads a public script	Script is saved locally and opened for rehearsal	Pass
Cloud Script Deletion	User deletes a previously downloaded script	Script is removed from local storage and UI updates	Pass
Cloud Upload	User uploads a new script	Script appears in public list and is available for others	Pass
Authentication	User logging in logging out	Successful google authentication and logging out	Pass

Table 8: Manual Testing of frontend

All frontend components performed successfully during testing. Script uploads, both local and cloud-based, were processed correctly, displaying content in the rehearsal screen as chat bubbles with accurate speaker labels. TTS playback worked consistently, honouring customised pitch, rate, and voice settings. Speech recognition correctly transcribed user input, and similarity scoring provided real-time feedback that was logged for progress tracking. Navigation between app screens was seamless, and offline mode allowed for uninterrupted access to locally stored scripts. Authentication and cloud operations—like uploading, downloading, and deleting scripts—functioned as expected, confirming a stable and user-ready frontend interface.

6.2 Database Testing

6.2.1 SQLite Database Testing

SQLite database testing was carried out to validate the app's ability to manage and persist data locally across user sessions. The tests focused on core database operations such as creating tables, inserting and retrieving data, updating and deleting records, and ensuring data integrity. These tests were essential for confirming that script details and rehearsal progress are accurately stored, retrieved, and maintained across app restarts and offline usage scenarios. All operations were tested using real scenarios within the app, and their results can be referenced in the appendix for detailed logs.

Feature	Test Description	Expected Result	Status
Database Connection	Open SQLite database successfully	Database connection opens without errors	Pass (see appendix D.1)
Create Table	Create scripts table with required columns	Table created with columns: id, title, deadline, lastnotified, etc.	Pass (see Appendix D.1)
Insert Script	Insert a new script record	New script inserted with correct data	Pass (see Appendix D.2)
Read Scripts	Retrieve all scripts	All scripts returned as an array of objects	Pass (see Appendix D.3)
Update Script	Update an existing script's fields (e.g., deadline)	Script record updated correctly, changes reflected in subsequent reads	Pass (see Appendix D.4)
Delete Script	Delete a script by ID	Script removed, no longer retrievable	Pass (see Appendix D.5)
Insert Progress	Insert rehearsal progress data with script ID and user ID	Progress record saved with correct progress and accuracy values	Pass
Read Progress	Retrieve progress by script ID	Correct progress records returned for the given script ID	Pass
Update Progress	Update progress entry for a script and user	Old entry replaced or updated, changes correctly stored	Pass
Delete Progress	Delete progress entry	Progress entry removed, cannot be fetched afterward	Pass (see Appendix D.3)
LastNotified Update	Update the lastnotified timestamp field	Timestamp updated to current date/time for the specified script	Pass (see Appendix D.6)
Data Persistence	Restart app and reopen database	Previously stored data persists and is accessible	Pass (see Appendix D.7)

Table 9: Manual Testing of Local SQLite database

The SQLite database passed all functional tests, confirming its reliability as the local storage engine for the application. The connection to the database was successfully established, and all required tables were created with the correct schema. CRUD operations for scripts and rehearsal progress were executed without errors, reflecting accurate data retrieval and updates. The lastnotified timestamp was correctly updated, and data persisted as expected even after the app was closed and reopened. These results confirm that the database layer provides a stable and robust foundation for local storage,

supporting seamless offline functionality and user progress tracking. For detailed testing outcomes, refer to the Appendix sections D.1 through D.7.

6.2.2 Appwrite Database Testing

Appwrite database testing was conducted to validate the integration between the mobile app and the cloud backend, ensuring all server-side operations behave as expected. The tests covered user creation, script uploads, authorisation checks, and progress tracking. A special focus was placed on validating user permissions for sensitive operations like deletion and download tracking. The Appwrite backend forms the core of the app's cloud functionality, enabling features such as public script sharing, centralised user progress tracking, and cloud-to-local synchronisation

Feature	**Test Description**	**Expected Result**	**Status**
getcloudscripts()	Fetch all scripts when documents exist	Returns an array of scripts	Pass (see Appendix D.7)
adduser()	Add a new user	User is created in Appwrite and userId saved to AsyncStorage	Pass (see Appendix E.1 and E.2)
adduser()	Add an existing user	Existing userId is fetched and saved, no duplicate created	Pass (see Appendix D.7)
addscript()	Upload a new unique script	Script is created and stored	Pass
addscript()	Upload a duplicate script	Returns message "Script already exists"	Pass (see Appendix E.3)
deletescript()	Delete script as the owner	Script is deleted successfully	Pass (see Appendix E.4)
deletescript()	Attempt to delete someone else's script	Returns "You are not authorized to delete this script"	Pass (see Appendix E.10)
updateDownloadedBy()	Valid user downloads someone else's script	Adds userId to downloadedby field	Pass (see Appendix E.5)
updateDownloadedBy()	User downloads own script	Returns "You cannot download your own script"	Pass (taken care in frontend)
updateDownloadedBy()	User downloads the same script again	Returns "You have already downloaded this script"	Pass (see Appendix E.6)
getprogress()	Fetch progress for a script with entries	Returns a list of progress documents	Pass
getprogress()	Fetch progress for script with no entries	Returns an empty array	Pass (see Appendix E.7)
updateProgress()	Create progress for a new user/script pair	New document is created with progress data	Pass (see Appendix E.8)
updateProgress()	Update existing progress document	Document is updated with new progress and accuracy	Pass (see Appendix E.9)

deleteprogress()	Delete all progress entries for a script	All matching documents are deleted	Pass (see Appendix E.4)
downloadScriptFromApp writeToLocal()	Download valid script and characters	Data is saved in local SQLite and download tracked	Pass (see Appendix E.5)

Table 10: Online Appwrite database Manual Testing

All Appwrite features performed as intended during testing. Functions like adduser() correctly handled both new and existing users without duplication. Script uploads and deletions responded appropriately to ownership constraints, reinforcing access control. Progress data was accurately stored, retrieved, and updated using updateProgress() and getprogress(). The system correctly prevented duplicate downloads and self-downloads using updateDownloadedBy(). Additionally, scripts downloaded from the cloud were successfully written to local storage for offline use. These outcomes confirm that the Appwrite integration is robust, secure, and well-suited to support the app's public script sharing and user tracking features. Refer to Appendix sections 1 through 3 for detailed test logs

6.3 Backend Load Testing using JMeter

To evaluate the performance and reliability of the backend APIs in our AI script rehearsal app, we performed load testing using Apache JMeter, an open-source tool designed for simulating multiple users sending requests to a server. JMeter allows us to assess how the system behaves under load by replicating concurrent user traffic. We focused on testing two key FastAPI endpoints: /similarity and /extracttext.

A Thread Group was configured with the following settings: 100 threads (users) to simulate 100 concurrent users, a ramp-up period of 1 second to initiate all threads nearly simultaneously, and a loop count of 1 to run each request once per user. This setup provided a stress test scenario to ensure the backend can handle high concurrency in a short timeframe. The test results were successful, with the server consistently returning HTTP response code 200 for all requests. This confirms that the endpoints are stable, responsive, and capable of handling multiple simultaneous requests without failure.

The Figure (18): below shows the thread properties used for the testing.

- Number of threads: This is the number of virtual users JMeter will simulate.
    - A value 100 is set, JMeter will act like 100 users hitting the FastAPI endpoint. Each thread acts independently and sends its own request(s).
- Ramp-up Period : This is the time JMeter takes to start all threads (users).
    - It's set as 1 second this jmeter will start all 100 threads within 1 seconds
    - If it's set as 10 seconds then, 10 user per second will start (100/10). Thus helps simulate more realistic traffic instead of sudden spikes.
- Loop count : This defines the number of times each user/thread will repeat the request.
    - If 1 : Each user sends 1 request.
    - If 10: Each user sends 10 request thus total request sent = Threads X loop count = 10X10 =100

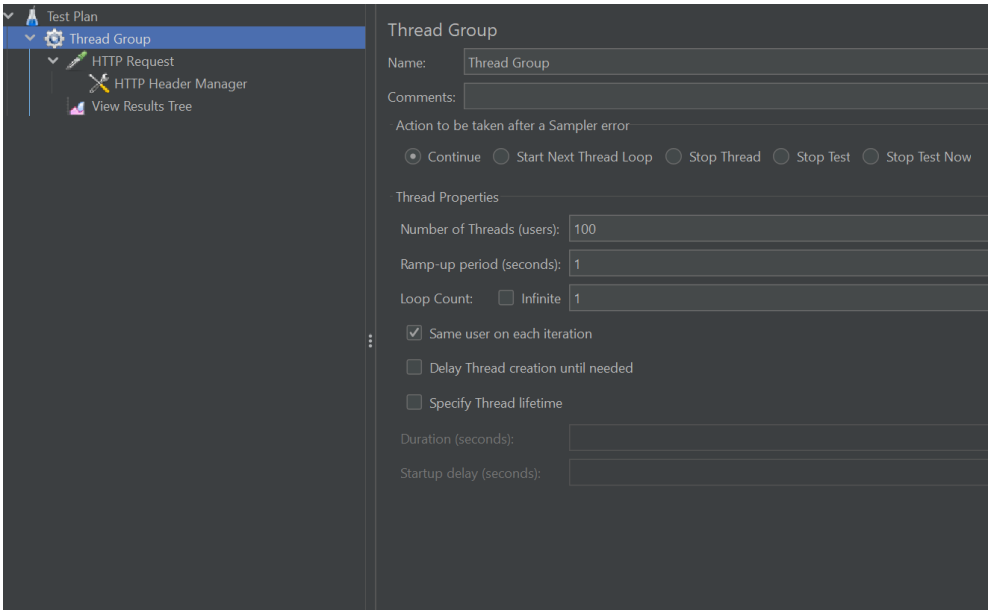


Figure 18: Thread properties for the backend endpoints

1. /similarity endpoint:

From the Figure (19) and Figure (20) we could see for the given request body we got a response body as expected and response code 200.

```
POST http://10.235.199.199:8000/similarity

POST data:
{
  "sentence1": "Hello world",

  "sentence2": "Hi world"
}
```

Listing 31: Payload for /similarity endpoint

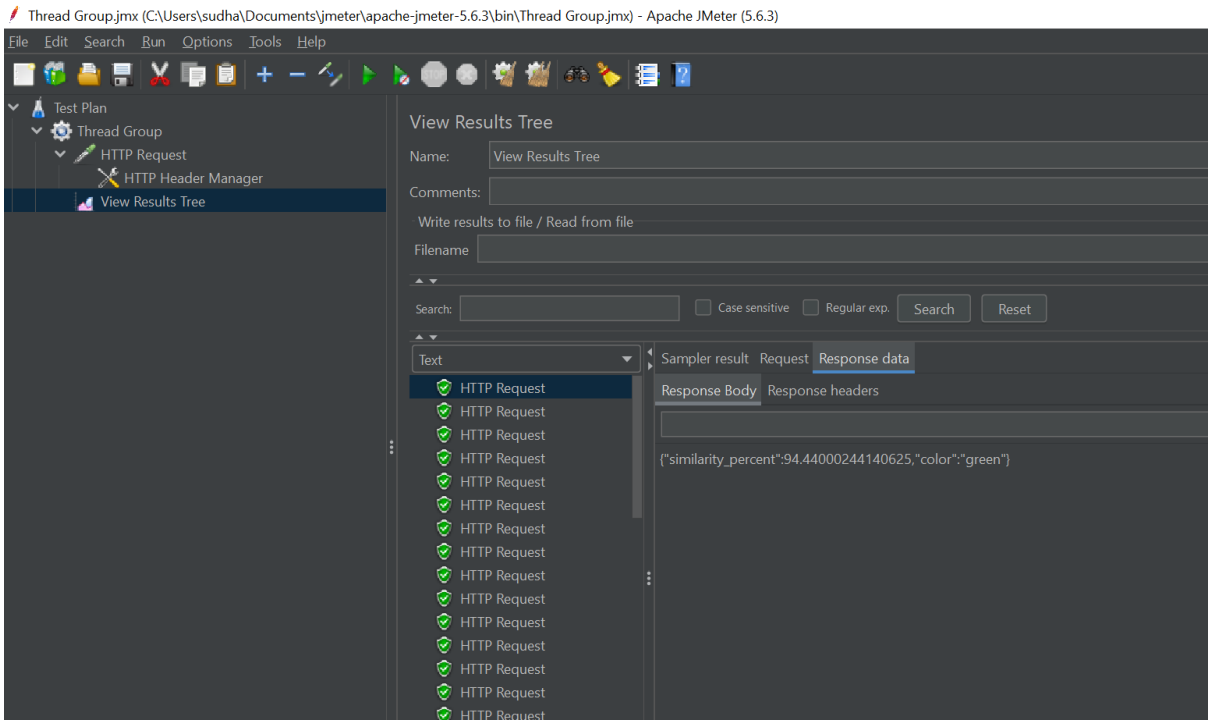


Figure 19: Screen shot of the response body of /similarity endpoint

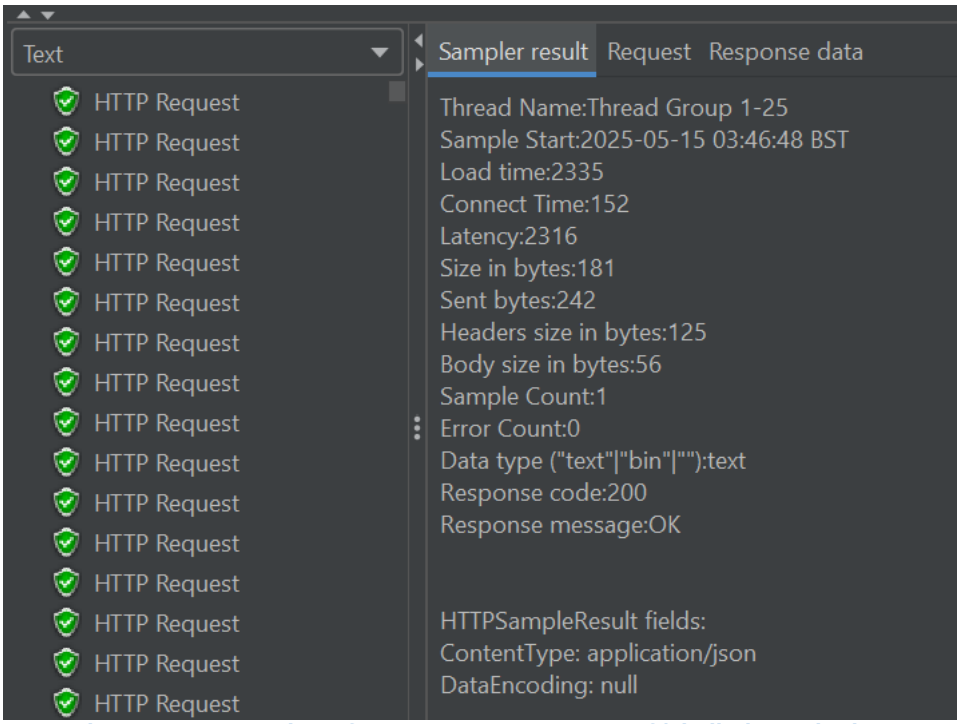


Figure 20: Screenshot of response message-200 of /similarity endpoint.

2. /extracttext endpoint :

From the Figure (21) and Figure (22) we could see for the given request body (a PDF file) we got a response body as expected and response code 200. The number characters extracted matched with the original script additionally ensuring the character extraction logic is working as expected.

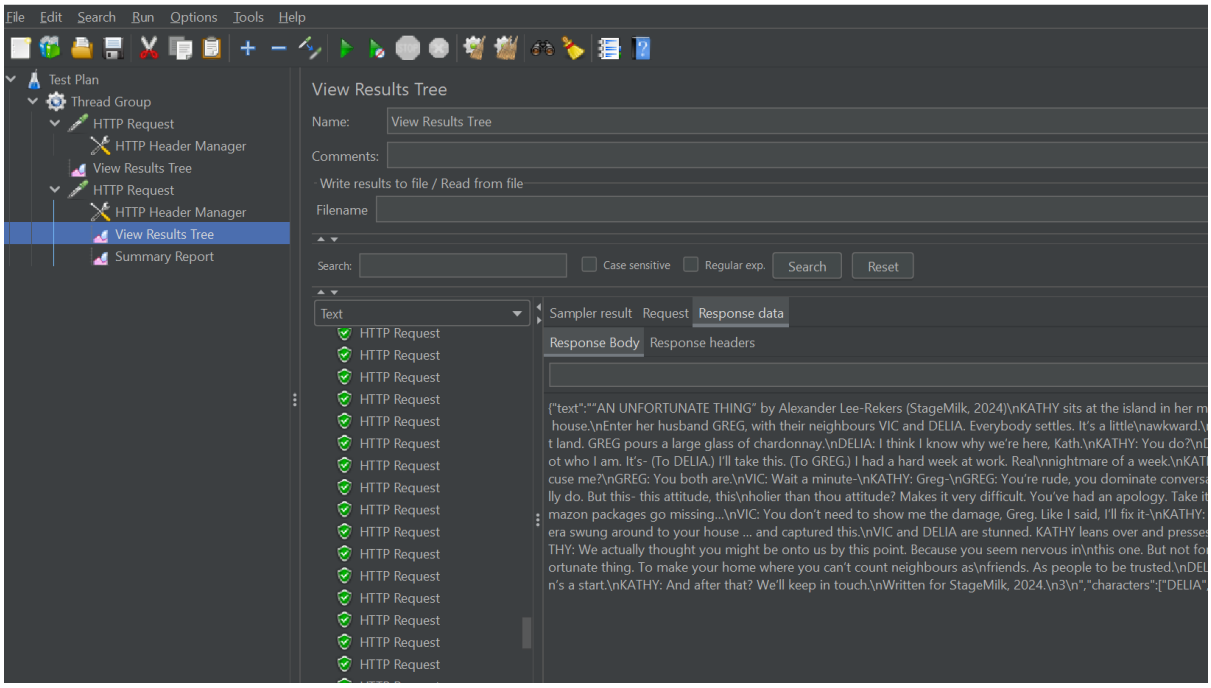


Figure 21: Screen shot of the response body of /extracttext endpoint

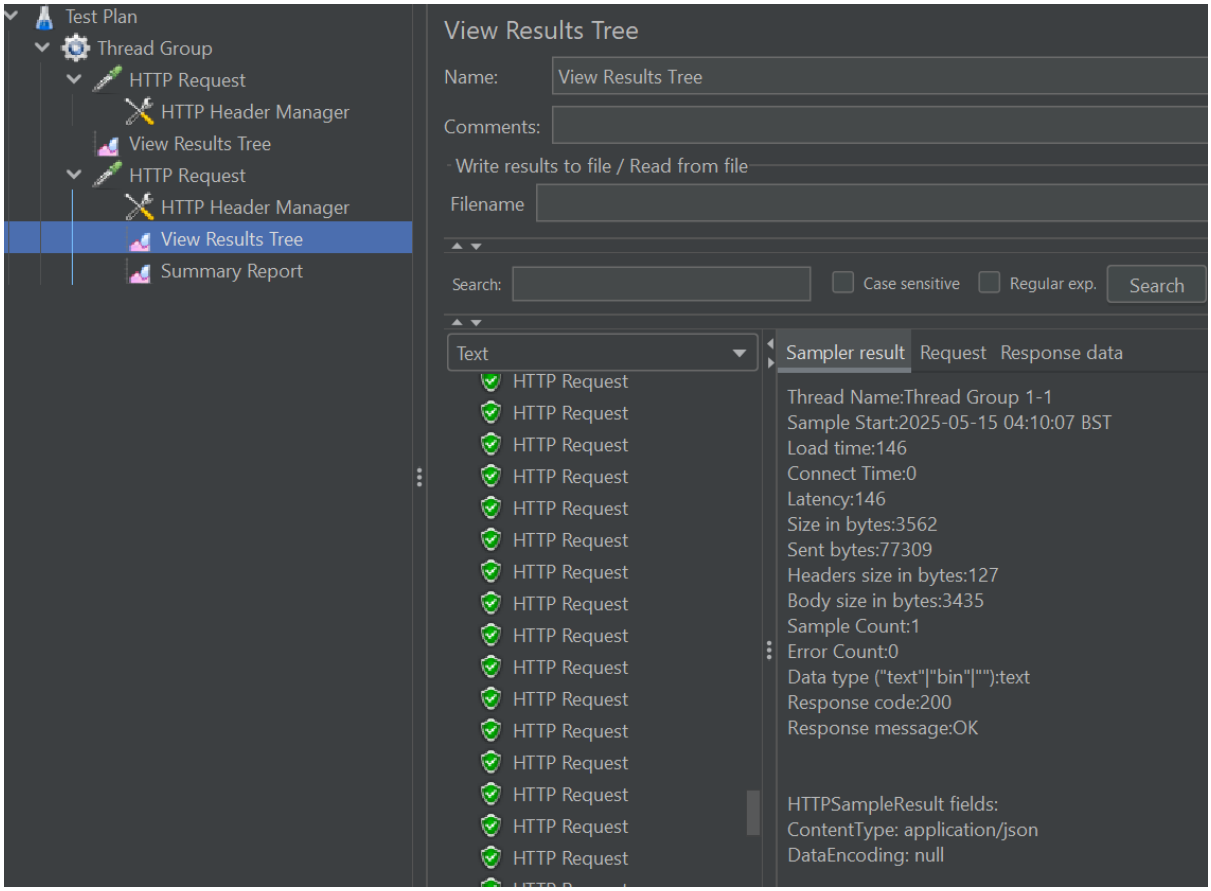


Figure 22: Screenshot of response message-200 of /extracttext endpoint

7 Results and Discussion

7.1 Analysis of Testing Outcomes

The AI script rehearsal app underwent multiple rounds of functional and user-centric testing to ensure reliability, usability, and integration of all core components. These components include the speech recognition system, similarity scoring via FastAPI, dynamic text-to-speech (TTS) playback, WhatsApp-style script interface, and progress tracking stored locally for offline access.

Speech recognition was tested extensively with real user input. Results showed acceptable accuracy under controlled conditions. However, challenges were observed in cases involving unclear pronunciation, low voice levels, or ambient background noise. While these issues did not critically hinder functionality, they occasionally led to incorrect line recognition.

Integration testing of the rehearsal screen confirmed that character-specific TTS settings—such as voice pitch, speech rate, and gender—were reliably loaded from the characters SQLite table and dynamically applied during rehearsals. The app successfully vocalized lines with the appropriate voice configuration, adding to the realism of the rehearsal experience. However, limitations were noted in the Android TTS engine, where no native male voice was available. This resulted in gender-specific constraints when voicing male characters on Android devices. A future enhancement could involve integrating Google Cloud Text-to-Speech or other third-party APIs to provide a broader range of voices across platforms.

Similarity scoring via FastAPI was tested both with dummy values and actual spoken inputs. Real-time feedback was verified through logging, and results were successfully stored in an object array to be used later in the progress.tsx screen. This setup allowed users to receive immediate feedback on how closely their spoken lines matched the original script, enabling performance improvements in successive attempts.

Overall, the testing process confirmed that all major functionalities are working as intended, with acceptable performance in most environments. Minor limitations identified—such as TTS voice availability and sensitivity of speech recognition—were noted for future improvement but did not significantly impact the core user experience.

7.2 Comparison with Existing Rehearsal Methods and Tools

Traditional script rehearsal techniques typically involve practicing with a partner, reading from printed scripts, or recording one's voice for later playback. While these methods can be effective for memorisation, they lack interactivity, real-time feedback, and automated progress tracking. As a result, performers often have limited insight into the accuracy or fluency of their delivery during solo rehearsal sessions.

Commercial tools like Rehearsal Pro offer some enhancements, such as the ability to record video, highlight lines, and rehearse cue lines independently. However, such tools generally lack intelligent feedback systems or analytics that assess line accuracy in real time. Moreover, most existing solutions are paid applications, which may limit accessibility for students or emerging artists.

From a research standpoint, many academic solutions in related fields—particularly in language learning—have explored the use of speech recognition and feedback for improving pronunciation and fluency [21][25]. However, few systems apply these techniques specifically to the performing arts. The AI-powered rehearsal app developed in this project

addresses this gap by combining natural language processing (NLP), speech recognition, and text-to-speech technologies in a unified mobile experience designed for actors and performers.

Compared to existing tools and methods, this app introduces several unique advantages:

- Real-time feedback on line delivery: By integrating sentence similarity scoring via a custom FastAPI backend, users receive immediate insights into how closely their spoken lines match the script, which is generally absent in traditional rehearsal tools.

- Autonomous rehearsal: The app eliminates the need for a physical rehearsal partner by enabling character-specific text-to-speech playback, allowing users to rehearse solo with simulated dialogue.

- Progress tracking and analytics: Using local storage (via AsyncStorage), the system visualises rehearsal performance over time, providing motivational and instructional value through performance charts.

Flexible, character-based voice playback: Customisable TTS settings—including pitch, rate, and gender—support immersive, personalised rehearsal experiences, approximating real interactions with co-actors. Furthermore, the system's offline-first architecture and use of entirely free tools and APIs make it highly accessible. Unlike many commercial apps that rely on subscriptions or locked features, this solution ensures broader usability without compromising on interactivity or feedback.

7.3 Challenges, Resolutions and lessons learned

Developing the AI-powered script rehearsal app involved navigating a range of technical and architectural challenges, especially given the use of React Native with Expo—a framework with certain limitations. One of the key hurdles was the lack of robust PDF libraries within the Expo ecosystem, which made importing and parsing scripts in PDF format difficult. As a workaround, text-based formats were prioritised, and custom parsing logic was implemented using regular expressions to extract characters, dialogue, actions, and scene descriptions from raw text input. This process required repeated testing and refinement to handle the variability in script formatting.

Additionally, the limited documentation for Expo posed significant challenges during integration of more advanced features such as speech recognition, SQLite, and offline storage. Setting up the application flow from scratch required designing a completely new development environment, including manual routing between screens, modal handling, and structured data passing across components. Another major challenge was synchronising the local and online databases, where ensuring consistency between Appwrite's cloud database, SQLite and AsyncStorage required careful planning. Balancing offline-first functionality with the need for API access—especially for real-time similarity scoring—proved to be nearly impossible for some features, which led to a hybrid architecture prioritising online access for NLP processes.

From a project management perspective, resolving these challenges required a more thoughtful and agile approach. One key lesson learned was that testing each component in isolation—such as speech recognition or TTS playback—does not always reveal integration issues that emerge during end-to-end testing. Moving forward, greater emphasis will be placed on holistic testing earlier in the development cycle. The project also highlighted the importance of accurately estimating task effort to better manage time and expectations, especially when dealing with unfamiliar tools or libraries.

To streamline development and reduce bugs, agile principles were adopted, with features implemented in small iterations rather than relying on extensive upfront reading or planning. Reading documentation alongside coding allowed for faster feedback and better adaptation to unexpected challenges. Git branching strategies were also introduced—features were developed in isolated branches, thoroughly tested, and only merged into the main branch once confirmed to be stable. This workflow helped reduce code conflicts and ensured a more maintainable and organised development process. These insights will guide future iterations of the app and inform architectural decisions in future projects.

7.4 Comparison with Aims and Objectives

The completed version of the AI-driven script rehearsal application closely adheres to the initial goals and objectives established at the project's outset. A cross-platform mobile application was successfully created utilising the React Native Expo framework, ensuring compatibility with both Android and iOS platforms. Essential features such as speech recognition, similarity scoring via an integrated FastAPI backend, and real-time feedback on spoken lines were effectively executed. sss application boasts a streamlined, chat-like interface that simulates a conversation, enhancing the rehearsal experience to be more intuitive and user-friendly for actors. Visual feedback tools were integrated through comprehensive progress reports and bar charts that illustrate user performance across multiple attempts. Furthermore, customisable text-to-speech options—including pitch, speed, and voice gender—were introduced to tailor the playback experience. Although certain limitations persist, such as limited offline capabilities and the absence of native male TTS voices on Android, the developed system successfully addresses the majority of the specified objectives and achieves the primary goal of assisting users in independent, AI-enhanced script rehearsal.

8. Conclusion and Future Work

8.1 Achievements

In terms of achievements, the project met most of the essential and recommended goals set at the start. Users can upload, organise, and rehearse scripts line by line, receive immediate feedback on errors, and monitor their improvement through a bar chart of recent attempts. The similarity scoring system distinguishes between minor paraphrasing and actual errors, making the feedback more intelligent and user centric. A leader board was integrated for public scripts, promoting community engagement, while customisable playback options added realism to rehearsals. Despite platform limitations—such as the lack of native male TTS voices on Android—the app provides a solid and practical rehearsal tool that is freely accessible and offline-capable for basic use.

8.2 Future Work

Looking forward, several enhancements can further elevate the app's capabilities. Multi-character rehearsal features can be developed to simulate full scene practice, allowing users to switch roles or rehearse with AI-generated co-actors. Emotion detection during speech recognition could offer deeper performance analysis by evaluating delivery and tone. The use of generative AI to auto-create scripts based on a scene description or theme would open new creative possibilities. Improved TTS models—possibly via integration with services like Google Cloud TTS could deliver more expressive and diverse voices. Enhanced collaboration features, such as script invitations or live rehearsal modes with other users, would enrich the app’s social and educational potential. These future developments aim to transform the app into a comprehensive rehearsal platform for actors.

8.3 Conclusion

The AI-powered script rehearsal mobile application successfully fulfills the majority of its essential and recommended requirements, demonstrating the feasibility of integrating artificial intelligence into performing arts practice. The app enables users to rehearse scripts independently through real-time speech recognition, natural language understanding, and AI-generated voice playback. Customisable TTS settings, an intuitive WhatsApp-style script interface, and semantic error detection provide a tailored and interactive experience. The app also features a script management system, similarity-based performance scoring, and a progress tracker with visual feedback, all of which were achieved using open-source and free tools to ensure accessibility.

Overall, the project demonstrates how modern AI tools can be applied meaningfully outside traditional domains. By combining natural language processing, speech technologies, and mobile development, this app offers a novel approach to solo script rehearsal. The lessons learned in designing, testing, and iterating this system highlight the importance of modular design, agile development, and careful balancing of offline-first principles with API dependencies. With a solid foundation now in place, the application is well-positioned for the future.

9. References

[1] D. McGaw, "The Importance of Rehearsal for Actors," *Backstage Magazine*, 2020. [Online]. Available: https://www.backstage.com/magazine/article/importance-actors-rehearsal-tips-67578/. [Accessed: 26-Apr-2025].
[2] S. Campbell, 'Can you practice acting on your own?', *StageMilk*, 26-Jan-2012. [Online]. Available: https://www.stagemilk.com/can-you-practice-acting-on-your-own/. [Accessed: 26-Apr-2025].

[3] P. Lam, 'The Impact Of Artificial Intelligence On The Art World', *Forbes.com*, 02-Feb-2024. [Online]. Available: https://www.forbes.com/councils/forbesbusinesscouncil/2024/02/02/the-impact-of-artificial-intelligence-on-the-art-world/. [Accessed: 26-Apr-2025].

[4] A. K. Jain, J. Mao, and K. M. Mohiuddin, "Artificial Neural Networks: A Tutorial," *Computer*, vol. 29, no. 3, pp. 31-44, Mar. 1996.

[5] M. Vukovic, "The Role of Mobile Apps in Education," *International Journal of Interactive Mobile Technologies (iJIM)*, vol. 11, no. 4, pp. 65-73, 2017.
[6] P. Gorla and V. Merugu, "Offline Mobile Applications: Benefits and Challenges," *International Journal of Computer Applications*, vol. 120, no. 2, pp. 20–25, 2015.
[7] Google AI, "Speech-to-Text: Automatic Speech Recognition," Google Cloud, 2024. [Online]. Available: https://cloud.google.com/speech-to-text . [Accessed:26-Apr-2025].
[8] L. Ramirez, "Round-Up: 5 Rehearsal Problems… and Solutions!" *Theatrefolk*, [Online]. Available: https://www.theatrefolk.com/blog/round-5-rehearsal-problems-solutions. [Accessed: Apr. 27, 2025].

[9] D. Genik, "How To Come Back To Rehearsal Improved And Better," *David Genik*, [Online]. Available: https://www.davidgenik.com/blog/how-to-come-back-to-rehearsal-improved-and-better.

[10] "The Challenges of Acting | Common Challenges for Actors," *StageMilk*, [Online]. Available: https://www.stagemilk.com/the-challenges-of-acting/. [Accessed: Apr. 27, 2025].

[11] "Experiencing Theatre: Challenges Actors Face in Bringing," *Course Sidekick*, [Online]. Available: https://www.coursesidekick.com/english/1682941. [Accessed: Apr. 27, 2025].

[12] M. McCormack, "Artificial Intelligence in Creative Industries," *AI & Society*, vol. 37, no. 2, pp. 645–658, 2022.

[13] S. Wang, C. Yu, and Z. Chen, "Democratizing Artificial Intelligence Through Mobile Platforms," *IEEE Access*, vol. 9, pp. 112345–112355, 2021.

[14] A. Graves et al., "Speech Recognition with Deep Recurrent Neural Networks," *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.* (ICASSP), 2013, pp. 6645–6649.

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *Proc. NAACL-HLT*, pp. 4171–4186, 2019.

[16] S. Ruder, "An Overview of Multi-Task Learning in Deep Neural Networks," *arXiv preprint* arXiv:1706.05098, 2017.

[17] P. Taylor, *Text-to-Speech Synthesis*, Cambridge University Press, 2009.

[18] M. Lane, "Mobile Technology for Learning: Benefits and Challenges," *Educational Technology Research and Development*, vol. 69, no. 4, pp. 1895–1910, 2021.

[19] "Script Rehearser - Helping actors rehearse their lines," *ScriptRehearser.com*, 2024. [Online]. Available: https://www.scriptrehearser.com/#about [Accessed on 28-Apr-2025].
[20] "Rehearsal® Pro — The App For Actors," *RehearsalPro.com*, 2024. [Online]. Available: https://rehearsal.pro/
[21] C. Cucchiarini, H. Strik, and L. Boves, "Speech recognition for second language learning: How to develop an application for speech training," *Speech Communication*, vol. 30, no. 2–3, pp. 131–153, 2000.
[22] "LineLearner — Memorize Lines for Plays, Film and TV," *LineLearnerApp.com*, 2024. [Online]. Available: https://apps.apple.com/us/app/linelearner/id368070258 [Accessed: 28-Apr-2025]
[23] "Scene Partner – The Rehearsal App," MyTheaterApps.com, 2020. [Online]. Available: https://scenepartner.ai. [Accessed: 28-Apr-2025]
[24] "ColdRead Mobile App," ColdRead App, 2021. [Online]. Available: https://coldreadapp.com/ [Accessed: 28-Apr-2025].
[25]K. M. Li and Q. H. Hegelheimer, "Review of research on applications of speech recognition technology to assist language learning," *ReCALL*, vol. 25, no. 1, pp. 21–38, Jan. 2013. [Online]. Available: https://www.cambridge.org/core/journals/recall/article/abs/review-of-research-on-applications-of-speech-recognition-technology-to-assist-language-learning/5E15DEA15B24F210B095A799708AD00B [Accessed: 28-Apr-2025].
[26] Expo Documentation, "Speech Recognition API," 2025. [Online]. Available: https://docs.expo.dev/versions/latest/sdk/speech/ [Accessed: 28-Apr-2025].
[27] Apple Developer, "Speech Framework," 2024. [Online]. Available: https://developer.apple.com/documentation/speech [Accessed: 28-Apr-2025].
[28] S. Deerwester et al., "Indexing by Latent Semantic Analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391-407, 1990.
[29] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, vol. 10, pp. 707–710, 1966.
[30] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," in *Proceedings of EMNLP-IJCNLP*, 2019.
[31] C. K. Lo and K. Hew, "The impact of web-based collaborative learning on students' academic achievement: A meta-analysis," *Educational Research Review*, vol. 24, pp. 52–68, 2018.
[32] D. McTear, "The Rise of Conversational Interfaces: A New Kid on the Block?," in *Proceedings of the International Workshop on Multimodal Interaction*, 2017.
[33] J. Sweller, "Cognitive load during problem solving: Effects on learning," *Cognitive Science*, vol. 12, no. 2, pp. 257-285, 1988.
[34] Expo Documentation, "AsyncStorage API," 2025. [Online]. Available: https://docs.expo.dev/versions/latest/sdk/async-storage/ [Accessed: 28-Apr-2025].
[35] M. Offline, "Building Offline-First Applications," 2023. [Online]. Available: https://offlinefirst.org/ [Accessed: 28-Apr-2025].
[36] J. Martin, "Designing Data-Intensive Applications," *O'Reilly Media*, 2017.
[37] Expo, "What is Expo?", Expo, 2025. [Online]. Available: https://expo.dev. [Accessed: 28-Apr-2025].
[38] React Native, "Learn React Native", React Native, 2025. [Online]. Available: https://reactnative.dev. [Accessed: 28-Apr-2025].
[39] FastAPI, "FastAPI Documentation", FastAPI, 2025. [Online]. Available: https://fastapi.tiangolo.com. [Accessed: 28-Apr-2025].

[40] SQLite, "SQLite Documentation", SQLite, 2025. [Online]. Available: https://www.sqlite.org. [Accessed: 28-Apr-2025].
[41] “Screenplay Format: 6 Elements You Have to Get Right,” *ScreenCraft*, published approximately 2.4 years ago. [Online]. Available: https://screencraft.org/blog/elements-of-screenplay-formatting/.
[Accessed: 16-May-2025]
[42] Appwrite, End-to-end backend server for web, mobile, and flutter developers,*2025*. [Online]. Available: https://appwrite.io/docs [Accessed: 28-Apr-2025].
[43] Hugging Face, "Sentence Transformers for Semantic Search", Hugging Face, 2025. [Online]. Available: https://huggingface.co/sentence-transformers. [Accessed: 28-Apr-2025].
[44] Google AI, "Universal Sentence Encoder", TensorFlow, 2025. [Online]. Available: https://www.tensorflow.org/hub. [Accessed: 28-Apr-2025].
[45] AsyncStorage documentation. "React Native AsyncStorage." [Online]. Available: https://react-native-async-storage.github.io/async-storage/ [Accessed: 28-Apr-2025].

APPENDIX

A. Backend Fast API code Snippets

A.1 Character and Text extraction

```
def extract_character_names(script_text: str):
    lines = script_text.splitlines()
    possible_names = set()

    for i, line in enumerate(lines):
        line_stripped = line.strip()

        if not line_stripped or len(line_stripped) > 40:
            continue

        name = None

        #COLON style example NAME:
        if re.match(r"^[A-Z][A-Z\s\-']{1,30}:", line_stripped):
            name = line_stripped.split(":")[0].strip()

        #ALL CAPS name followed by dialogue
        elif (
            line_stripped.isupper()
            and i + 1 < len(lines)
            and lines[i + 1].strip()
            and not lines[i + 1].strip().isupper()
        ):
            name = line_stripped

        if name:
```

```
            # Normalize name
                    name   =   re.sub(r"\s*\(?(CONT'?D|CONT)\)?",   "",   name,
flags=re.IGNORECASE)
            name = re.sub(r"[^\w\s'-]", "", name).strip()

            #Allow 1 or 2 word names only
            word_count = len(name.split())
            if word_count == 0 or word_count > 2:
                continue

            # ⃣ Filter out scene directions / headers
              if any(word in name for word in ["INT", "EXT", "BY", "SCENE",
"ACT", "DAY", "NIGHT"]):
                continue

            possible_names.add(name)

    return sorted(possible_names)

@app.post("/extracttext")
async def extract_text(file: UploadFile = File(...)):
    """API endpoint to extract text from a Base64 PDF string."""
    try:
        # Read the uploaded file
        pdf_bytes = await file.read()
        extracted_text = extract_text_from_pdf(pdf_bytes)
        character_names = extract_character_names(extracted_text)

        return {"text": extracted_text, "characters": character_names}
    except Exception as e:
        return {"error": f"Failed to process PDF: {e}"}
```

A.2 Similarity Check endpoint

```
@app.post("/similarity")
async def calculate_similarity(data: SimilarityRequest):
    try:
        # Encode the sentences
        vectors = similarity_model.encode([data.sentence1, data.sentence2])
        similarity = cosine_similarity([vectors[0]], [vectors[1]])[0][0]
        similarity_percent = float(round(similarity * 100, 2))

        # Determine color based on the similarity percentage
        if similarity_percent >= 90:
            color = "green"
        elif similarity_percent >= 70:
            color = "orange"
        else:
            color = "red"
```

```
        return {
            "similarity_percent": similarity_percent,
            "color": color
        }
    except Exception as e:
        return {"error": f"Similarity calculation failed: {str(e)}"}
```

B. Database Code

B.1 Code snippet for database and table creation

```
export const createDatabase = async() => {

try{
    const db = await SQLite.openDatabaseAsync("localdb");
        await db.execAsync(
            `
            PRAGMA journal_mode = WAL;
            CREATE TABLE IF NOT EXISTS scripts (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            userID TEXT,
            docID TEXT,
            createdby TEXT,
            title TEXT,
            file BLOB,
            fileURI TEXT,
            content TEXT,
            character TEXT,
            lastnotified TEXT DEFAULT CURRENT_TIMESTAMP,
            dateCreated TEXT DEFAULT CURRENT_TIMESTAMP,
            deadline TEXT DEFAULT CURRENT_TIMESTAMP);`

        );
        //const schema = await db.getAllAsync('PRAGMA table_info(scripts);');
        //console.log("TABLE SCHEMA:",schema);
        console.log("script table created succesfully" );

        await db.execAsync(
            `
            PRAGMA journal_mode = WAL;
            CREATE TABLE IF NOT EXISTS characters (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            scriptID INTEGER,
            userID TEXT,
            name TEXT,
            pitch FLOAT,
            rate FLOAT,
```

```
            voice TEXT,
            UNIQUE(name, scriptID)
        );`

        );
        console.log("characters table created succesfully" );

    }catch(e){
        console.error("Error creating DB", e);
    }
    };
```

B.2 Code snippet to update lastnotified field

```
// Function to update lastnotified field
const updateLastNotified = async (scriptId) => {
  try {
    const db = await SQLite.openDatabaseAsync("localdb");
    const currentTimestamp = new Date().toISOString();
    await db.runAsync(
      `UPDATE scripts SET lastnotified = ? WHERE id = ?`,
      [currentTimestamp, scriptId]
    );
    console.log(`Last notified timestamp updated for script ID: ${scriptId}`);
  } catch (error) {
    console.error("Error updating lastnotified", error);
  }
};
```

B.3 Code snippets for all the database related functions

```
export const insertScript = async(title, fileBlob, fileURI, content, docID =
null, createdby = null) =>{
    try{
        const userID = await getCurrentUserID();
        const db = await SQLite.openDatabaseAsync("localdb");
        console.log("Inserting Script - Title:", title);
        const insertedScript=  await db.runAsync(
         `INSERT INTO scripts (userID, title, file, fileURI, content, docID,
createdby)
         VALUES (?, ?, ?, ?, ?, ?, ?)`,[userID, title, fileBlob, fileURI,
content, docID, createdby]
        );
        const allRows = await db.getAllAsync('SELECT * FROM scripts');
     console.log("last inserted id :", insertedScript.lastInsertRowId )
    return insertedScript.lastInsertRowId;
    }catch(error){
```

```
        console.log("Upload failed", error);

    }
    };

export const insertCharacters = async(characters, scriptID, pitch , rate) =>{

    // Convert pitch and rate to floats
    const parsedPitch = parseFloat(pitch as string);
    const parsedRate = parseFloat(rate as string);
    const userID = await getCurrentUserID();
    const db = await SQLite.openDatabaseAsync("localdb");
    for (const name of characters){
    try{

        console.log("Inserting characters of ", scriptID);
    await db.runAsync(
         'INSERT OR IGNORE INTO characters ( userID, name, scriptID, pitch,
rate ) VALUES (?,?,?,?,?)',userID, name, scriptID, parsedPitch, parsedRate

        );

        console.log(`Inserted: ${name} (Script ${scriptID})`);
    }catch(error){
        console.log("Upload failed", error);

    }
    };
}

export const updatecharactername = async(scriptID, character)=>{
    try{
        const db = await SQLite.openDatabaseAsync("localdb");
         const result = await db.runAsync('UPDATE scripts SET character = ?
WHERE id = ?', character, scriptID);
        console.log(character, " INSERTED");
    }catch(error){
        console.error("Error inserting character", error);
    }
}

export const updateCharacterVoice = async (scriptID: number, name: string,
pitch, rate, voice) => {
    try {
      const db = await SQLite.openDatabaseAsync("localdb");
      await db.runAsync(
        'UPDATE characters SET pitch = ?, rate = ?, voice = ? WHERE scriptID =
? AND name = ?',
```

```
        pitch, rate, voice, scriptID, name
      );
          console.log(`${name}  updated  voice:${voice},  pitch  ${pitch}:,
rate:${rate} `);
    } catch (error) {
      console.error("Error updating character's voice", error);
    }
  };

export const getAllCharacters = async(scriptID) =>{
    try{
        const db = await SQLite.openDatabaseAsync("localdb");
        const results = await db.getAllAsync(
            'SELECT * FROM characters WHERE scriptID =?',scriptID
        );
        console.log(" all characters are retrieved");
        return results;
    }catch(e)
    {
        console.error("ERROR fetching characters name:",e);
        return [];
    }

}


export const getAllScripts = async () => {
    const userID = await getCurrentUserID();
    console.log("current user ID: ", userID)
  try {
    const db = await SQLite.openDatabaseAsync("localdb");
    const result = await db.getAllAsync("SELECT * FROM scripts WHERE userID =
?", userID);
    console.log(`script of userID: ${userID} retrieved`)
    return result
  } catch (error) {
    console.error("Error retrieving scripts", error);
    throw error;
  }
};

export const deleteScript = async(id)=>{
    try{
        const db = await SQLite.openDatabaseAsync("localdb");
        await db.runAsync("DELETE FROM scripts WHERE id = ?",id);
        console.log("script deleted successfully:",id);
        await db.runAsync("DELETE FROM characters WHERE scriptID =?",id);
    const progressKey = `rehearsalResults_${id}`;
    await AsyncStorage.removeItem(progressKey);
```

```
    console.log("Progress deleted from AsyncStorage:", progressKey);
    }catch(error){
        console.error("Error deleting script",error);
        throw error;

    }
}

export const deletecharacter = async(id)=>{
    try{
        const db = await SQLite.openDatabaseAsync("localdb");
        await db.runAsync("DELETE FROM characters WHERE scriptID =?",id);
        console.log("character deleted successfully:",id);

    }catch(error){
        console.error("Error deleting script",error);
        throw error;

    }
}


export const getScriptById = async(id)=>{
    try{
        console.log("script retrieving :",id);
        const db = await SQLite.openDatabaseAsync("localdb");
        const result= await db.getFirstAsync("SELECT * FROM scripts WHERE id =
?;",id);
        console.log("script retrieved successfully:",id);
        console.log("Title of the script :",result.title);
        return result;
    }catch(error){
        console.error("Error deleting script",error);
        throw error;

    }
}

//update deadline
export const updateScriptDeadline = async (id, deadline) => {
    const db = await SQLite.openDatabaseAsync("localdb");
    console.log("script deadline updated succesfully")
    return await db.runAsync(
      'UPDATE scripts SET deadline = ? WHERE id = ?',
      deadline, id
    );
  };
```

B.4 adduser function code snippet

```
export const adduser = async (email: string, username: string) => {
  try {
    console.log("Checking if user already exists...");

    // Check if email already exists in the collection
    const existing = await database.listDocuments(
      config.db,
      config.col.user,
      [Query.equal("email", email)]
    );

    if (existing.total > 0) {
      const existingId = existing.documents[0].$id;
      console.log("User already exists with ID:", existingId);

      // save it in AsyncStorage anyway
      await AsyncStorage.setItem(email, existingId);

      return existingId;
    }

    // Email not found — create a new user
    const response = await database.createDocument(
      config.db,
      config.col.user,
      ID.unique(),
      {
        email,
        username,
      }
    );

    const newId = response.$id;
    await AsyncStorage.setItem(email, newId);

    console.log(`New user created with ID: ${newId}`);
    return newId;

  } catch (error) {
    console.error("Error in adduser:", error);
    return null;
  }
};
```

B.5 Checking for duplicate script before adding

```
export const addscript = async (
```

```
  uploadedby: string,
  useremail: string,
  content: string,
  characterlist: string,
  title: string
) => {
  try {
    // Check if script with same useremail, title and characterlist already
exists
    const existing = await database.listDocuments(
      config.db,
      config.col.cloudscripts,
      [
        Query.equal('useremail', useremail),
        Query.equal('title', title),
        Query.equal('characterlist', characterlist)
      ]
    );

    if (existing.total > 0) {
      console.log('Duplicate script. Upload skipped.');
      return { status: 'duplicate', message: 'Script already exists' };
    }
```

B.6 Update downloadedby array function

```
export const updateDownloadedBy = async (userId: string, scriptId: string) =>
{
  try {
    // Fetch the script by scriptId
          const     script     =     await     database.getDocument(config.db,
config.col.cloudscripts, scriptId);

    // Check if the uploadedby is not the same as the userId
    if (script.uploadedby === userId) {
      console.log('You cannot download your own script');
      return { status: 'error', message: 'You cannot download your own script'
};
    }

    // Check if the userId is already in the downloadedby array
    if (script.downloadedby && script.downloadedby.includes(userId)) {
      console.log('User has already downloaded this script');
       return { status: 'error', message: 'User has already downloaded this
script' };
    }

    // Update the downloadedby array to include the userId
```

```
    const updatedDownloadedBy = script.downloadedby ? [...script.downloadedby,
userId] : [userId];

    // Update the script document with the new downloadedby field
        const   updatedScript   =   await   database.updateDocument(config.db,
config.col.cloudscripts, scriptId, {
      downloadedby: updatedDownloadedBy
    });

    console.log('Downloadedby updated:', updatedScript.downloadedby);
    return { status: 'success', message: 'Downloadedby updated successfully',
data: updatedScript };

  } catch (error) {
    console.error('Error updating downloadedby:', error);
    return { status: 'error', message: 'An error occurred while updating' };
  }
};
```

B.7 Code snippet for updateProgress function

```
export const updateProgress = async (
  scriptID: string,
  userID: string,
  username: string,
  title: string,
  progress: number,
  accuracy: number
) => {
  try {
    // 1. Check if document exists for this script + user
          const    existing    =    await    database.listDocuments(config.db,
config.col.progress, [
      Query.equal('scriptid', scriptID),
      Query.equal('userid', userID),
    ]);

    if (existing.total > 0) {
      // 2. If exists, update the first matching document
      const docId = existing.documents[0].$id;
             const   updated   =   await   database.updateDocument(config.db,
config.col.progress, docId, {
        title,
        progress,
        accuracy,
      });

      console.log('Progress updated new progress', progress );
      return updated;
```

```
    } else {
      // 3. If not exists, create new document
              const    newDoc    =    await    database.createDocument(config.db,
config.col.progress, ID.unique(), {
        scriptid: scriptID,
        userid: userID,
        username,
        title,
        progress,
        accuracy,
      });

      console.log('Progress created:', newDoc);
      return newDoc;
    }
```

B.8 Code snippet of downloadScriptFromAppwriteToLocal function

```
xport const downloadScriptFromAppwriteToLocal = async (scriptID: string) => {
  try {
    const { userID } = await getCurrentUserdetails();

    // Fetch the script document from Appwrite
    const script = await database.getDocument(
      config.db,
      config.col.cloudscripts,
      scriptID
    );

    if (!script) {
      console.warn("Script not found with ID:", scriptID);
      alert("Script not found.");
      return;
    }

    // Early check: prevent re-downloads
    if (script.uploadedby === userID) {
      alert("You cannot download your own script.");
      return;
    }

    if (script.downloadedby && script.downloadedby.includes(userID)) {
      alert("You have already downloaded this script.");
      return;
    }

    const { content, title, characterlist } = script;

    // Insert script locally
```

```
    const localScriptID = await insertScript(
      title,
      "", // No fileBlob
      "", // No file URI
      content,scriptID,script.uploadedby
    );

     await  insertCharacters(characterlist,  parseInt(localScriptID.toString(),
10), 1.0, 1.0);

    // Mark as downloaded
    const updateResult = await updateDownloadedBy(userID, scriptID);
    if (updateResult.status === 'success') {
      console.log(`Marked script "${title}" as downloaded by ${userID}`);
    } else {
      console.warn(`Download tracking failed: ${updateResult.message}`);
    }

    alert(`Script "${title}" downloaded successfully.`);
  }
```

C. Frontend Development

C.1 Code snippet to calculate Accuracy and Progress

```
const calculateAccuracy = () => {
    if (similarityResults.length === 0) return 0;
    const total = similarityResults.reduce((sum, r) => sum + r.similarity, 0);
    return Math.round(total / similarityResults.length);
  };

  const calculateProgress = () => {
    const userDialogues = parsedScript.filter(
      (line) => line.type === 'dialogue' && line.character === character
    );
    const completed = readLines.filter(
      (index) =>
        parsedScript[index]?.type === 'dialogue' &&
        parsedScript[index]?.character === character
    ).length;
    // (spokenlines/total lines of the user)

    return Math.round((completed / userDialogues.length) * 100);
  };
```

C.2 Code snippet to check similarity before moving to next line

```
if (similarity_percent >= 50) {
        setSpokenLines((prev) => [...prev, transcript]);
        setReadLines((prev) => [...prev, currentIndex]);
        setSimilarityResults((prev) => [
          ...prev,
          {
            original: originalLine,
            spoken: transcript,
            color,
            similarity: similarity_percent,
          },
        ]);
        setCurrentIndex((idx) => idx + 1);
      } else {
          Alert.alert("Try Again", "Your response wasn't close enough. We'll
retry in 5 seconds.");
```

C.3 Script Parser Code snippet

```
export function parseScript(script: string): ScriptPart[] {
  const lines = script.split('\n');
  const parsed: ScriptPart[] = [];

  let currentCharacter = '';
  let currentDialogue: string[] = [];

  const flushDialogue = () => {
    if (currentDialogue.length && currentCharacter) {
      parsed.push({
        type: 'dialogue',
        character: currentCharacter,
        content: currentDialogue.join(' ').trim(),
      });
      currentDialogue = [];
    }
  };

  for (let i = 0; i < lines.length; i++) {
    let line = lines[i].trim();
    if (!line) continue;

    // type 1: Character with colon (e.g., JOHN:)
    const colonMatch = line.match(/^([A-Z][A-Z\s'-]{1,30}):/);
    if (colonMatch) {
      flushDialogue();
      currentCharacter = colonMatch[1].trim();
      const dialogue = line.replace(`${currentCharacter}:`, '').trim();
      if (dialogue) currentDialogue.push(dialogue);
```

```
      continue;
    }

    // type 2: Character in all caps, next line is dialogue
    const allCapsName = /^[A-Z\s'-]{2,30}$/.test(line);
    const nextLine = i + 1 < lines.length ? lines[i + 1].trim() : '';
    if (
      allCapsName &&
      nextLine &&
      !/^[A-Z\s'-]{2,30}$/.test(nextLine)
    ) {
      flushDialogue();
      currentCharacter = line.trim();
      continue;
    }

    // Actions like (Smiles) or enters/exits
    if (
      (line.startsWith('(') && line.endsWith(')')) ||
      /(begins|holds|enters|exits)/i.test(line)
    ) {
      flushDialogue();
      parsed.push({
        type: 'action',
        content: line,
      });
      continue;
    }

    // Scene description (no current character)
    if (!currentCharacter) {
      flushDialogue();
      parsed.push({
        type: 'scene',
        content: line,
      });
      continue;
    }

    // Add to current character's dialogue
    currentDialogue.push(line);
  }

  flushDialogue(); // Final cleanup
  return parsed;
}
```

C.4 Code snippet for Notification service

```
export const scheduleNotificationsForScripts = async () => {
  const scripts = await getAllScripts();
  const currentDate = new Date();
  // Loop through each script
  for (const script of scripts) {
    // Parse the deadline and last notified date
    const deadlineDate = new Date(script.deadline);
    const lastNotifiedDate = new Date(script.lastnotified);
    // Check if the deadline is in the future and if lastnotified is not today
                if        (deadlineDate        >        currentDate        &&
!DateFns.isToday(script.lastNotifiedDate)) {
      console.log(`Scheduling notification for script ID: ${script.id}`);
      // Schedule notification 30 minutes after the current time
      const identifier = await Notifications.scheduleNotificationAsync({
        content: {
          title: `Deadline approaching for ${script.title}`,
           body: `Your deadline for the script "${script.title}" is coming up
soon.`, // Customise this body text
        },
        trigger: {
          seconds: 30 *60, // 30 minutes
        },
      });
      // Optionally, you can store this identifier or update the lastnotified
timestamp
      await updateLastNotified(script.id); // Function to update last notified
timestamp in the database
    }
  }
};
```

C.5 Frontend Folder Structure

D SQLite database Testing

D.1 Database and table Creation

```
(NOBRIDGE) LOG  script table created succesfully
(NOBRIDGE) LOG  characters table created succesfully
(NOBRIDGE) LOG  Database initialized successfully
```

D.2 Script insertion confirmation

```
(NOBRIDGE) LOG  Document ID: 68261fed00095c24b858
(NOBRIDGE) LOG  doc ID after uploading :  68261fed00095c24b858
(NOBRIDGE) LOG  Inserting Script - Title: Auld-Lang-Syne.pdf
(NOBRIDGE) LOG  last inserted id : 5
(NOBRIDGE) LOG  Inserting characters of  5
(NOBRIDGE) LOG  Inserted: ASH (Script 5)
(NOBRIDGE) LOG  Inserting characters of  5
(NOBRIDGE) LOG  Inserted: CHARLIE (Script 5)
(NOBRIDGE) LOG  Script inserted successfully: Auld-Lang-Syne.pdf
```

D.3 Successful script retrieve

```
(NOBRIDGE) LOG  script of userID: 681fc98c003ba60d2e77 retrieved
(NOBRIDGE) LOG  datalength 2
```

D.4 Script updated successfully

```
(NOBRIDGE) LOG  script deadline updated succesfully
(NOBRIDGE) LOG  Deadline updated successfully
```

D.5 Script Deleted successfully

```
(NOBRIDGE) LOG  datalength 2
(NOBRIDGE) LOG  script deleted successfully: 5
(NOBRIDGE) LOG  Progress deleted from AsyncStorage: rehearsalResults_5
(NOBRIDGE) LOG  current user ID:  681fc98c003ba60d2e77
(NOBRIDGE) LOG  script of userID: 681fc98c003ba60d2e77 retrieved
(NOBRIDGE) LOG  datalength 1
```

D.6 Last Notified field updated successfully

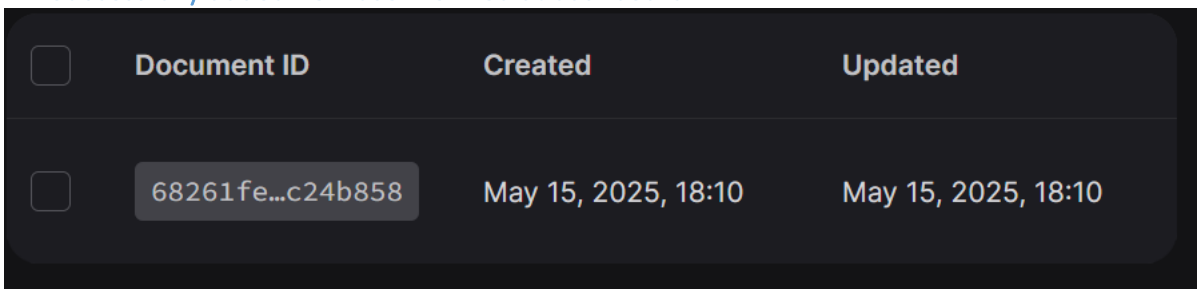
```
(NOBRIDGE) LOG  Scheduling notification for script ID: 4
(NOBRIDGE) LOG  Last notified timestamp updated for script ID: 4
```

D.7 Previous data retrieved and app restart

```
(NOBRIDGE) LOG  current user ID:  681fc98c003ba60d2e77
(NOBRIDGE) LOG  script table created succesfully
(NOBRIDGE) LOG  characters table created succesfully
(NOBRIDGE) LOG  Database initialized successfully
(NOBRIDGE) LOG  script of userID: 681fc98c003ba60d2e77 retrieved
(NOBRIDGE) LOG  Sample script already uploaded, skipping...
(NOBRIDGE) LOG  response: {"message": "connection established"}
(NOBRIDGE) LOG  current user ID:  681fc98c003ba60d2e77
(NOBRIDGE) LOG  script of userID: 681fc98c003ba60d2e77 retrieved
(NOBRIDGE) LOG  Scheduling notification for script ID: 4
(NOBRIDGE) LOG  Last notified timestamp updated for script ID: 4
(NOBRIDGE) LOG  cloud script retrieved
(NOBRIDGE) LOG  current user ID:  681fc98c003ba60d2e77
(NOBRIDGE) LOG  script of userID: 681fc98c003ba60d2e77 retrieved
(NOBRIDGE) LOG  datalength 2
```

E. Appwrite database testing

E.1 Successfully added new user verified at dashboard




```
4
 (NOBRIDGE) LOG  cloud script retrieved 2
```

E.2 New User Added

```
 (NOBRIDGE) LOG  current user ID:  681fc98c003ba60d2e77
 (NOBRIDGE) LOG  script of userID: 681fc98c003ba60d2e77 retriev
ed
 (NOBRIDGE) LOG  Scheduling notification for script ID: 4
 (NOBRIDGE) LOG  Last notified timestamp updated for script ID:
 4
 (NOBRIDGE) LOG  cloud script retrieved 2
 (NOBRIDGE) LOG  Checking if user already exists...
 (NOBRIDGE) LOG  New user created with ID: 682629e9001308decb83
```

E.3 Skipping Duplicate scripts

```
(NOBRIDGE) LOG  datalength 2
(NOBRIDGE) LOG  Character list:  ["ASH", "CHARLIE"]
(NOBRIDGE) LOG  Duplicate script. Upload skipped.
```

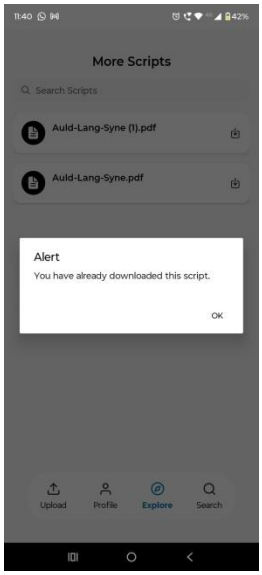
E.4 Successful deletion of a Script

```
 (NOBRIDGE) LOG  datalength 3
 (NOBRIDGE) LOG  cloud script retrieved 3
 (NOBRIDGE) LOG  cloud script retrieved 3
 (NOBRIDGE) LOG  Deleting progress for script: 682668c90010260
5abf
 (NOBRIDGE) LOG  No progress found for this script ID
 (NOBRIDGE) LOG  Script deleted: 682668c9001026095abf
 (NOBRIDGE) LOG  cloud script retrieved 2
```

E.5 Added user to downloadedby Attribute

```
(NOBRIDGE) LOG  Downloadedby updated: ["682629e9001308decb83"]
(NOBRIDGE) LOG  Marked script "Auld-Lang-Syne.pdf" as downloaded by 682629e9001308decb83
```

E.6 Alert message displayed if already downloaded



E.7 Accessing script with no progress data

```
(NOBRIDGE) LOG  current user ID:  682629e9001308decb83
(NOBRIDGE) LOG  script of userID: 682629e9001308decb83 retrieved
(NOBRIDGE) LOG  cloud script retrieved 2
(NOBRIDGE) LOG  Fetching progress for script: 682667050024ec86fc3e
(NOBRIDGE) LOG  Fetched documents: []
(NOBRIDGE) LOG  progress data : []
```

E.8 Created a new progress document

```
(NOBRIDGE) LOG  updated progress collection
(NOBRIDGE) LOG  Progress created: {"$collectionId": "681fd26f001a7434cb2c", "$createdAt": "202
5-05-15T22:43:48.355+00:00", "$databaseId": "681f410e003153629b54", "$id": "68266e23001ca294c9d
f", "$permissions": [], "$updatedAt": "2025-05-15T22:43:48.355+00:00", "accuracy": 89, "progres
s": 25, "scriptid": "682667050024ec86fc3e", "title": "Auld-Lang-Syne.pdf", "userid": "682629e90
01308decb83", "username": "Jing-tsong jeng"}
```

E.9 Updating existing Progress Document

```
(NOBRIDGE) LOG  Progress updated: {"$collectionId": "681fd26f001a7434cb2c", "$createdAt": "202
5-05-15T22:43:48.355+00:00", "$databaseId": "681f410e003153629b54", "$id": "68266e23001ca294c9d
f", "$permissions": [], "$updatedAt": "2025-05-15T22:49:34.290+00:00", "accuracy": 90, "progres
s": 40, "scriptid": "682667050024ec86fc3e", "title": "Auld-Lang-Syne.pdf", "userid": "682629e90
01308decb83", "username": "Jing-tsong jeng"}
```

E.10 Protected from deleting other user's script