ATTENDSCAN MOBILE APPS: UNIVERSITY STUDENT FINAL EXAM ATTENDANCE USING DEEP LEARNING

By

VICKY YII SHU CHI

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONOURS)
Faculty of Information and Communication Technology
(Kampar Campus)

FEBRUARY 2026

COPYRIGHT STATEMENT

This Final Year Project report is submitted in partial fulfillment of the requirements for the degree of Bachelor of Computer Science (Honours) at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project report represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project report may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Ms. Tsue Kwan Lee, for giving me the opportunity to design this HTR model integrated with a mobile application to automate university exam attendance. Her patient guidance, continuous encouragement, and insightful advice have been invaluable throughout the course of this project.

I am also sincerely grateful to my family and classmates for their constant support, encouragement, and care during the development of this project.

ABSTRACT

This project presented AttendScan, an automated attendance marking system for university final examinations that incorporates Deep Learning-based Handwritten Text Recognition (HTR) into a mobile application. In large scale examination venues, manual checking of attendance is normally time-consuming and prone to human error. To address these inefficiencies, the proposed system used a CNN-BiLSTM-CTC architecture to recognise unsegmented and variable length handwritten fields such as index numbers, seat numbers and subject names. This design specifically targets two critical HTR challenges: handwriting variability and limited character scope. To provide strong recognition of various writing styles and alphanumeric characters, the model was trained on a hybrid dataset that included real exam attendance slips and publicly available handwriting datasets. To obtain optimal performance, the system employed Optuna to automatically optimise hyperparameters, which significantly reduced Character Error Rate and Word Error rate, and improved Exact Match Accuracy. To illustrate the practical application of HTR towards automation, the model was implemented as a RESTful API to be called by mobile applications. A student-facing mobile application can be used to capture attendance slips, send them to the API, and submit the recognised results to mark attendance. Meanwhile, a dedicated web platform allows administrators to manage student attendance records. This project combines the deep learning recognition with the practical use and thereby reduces the administrative burden and offers a scalable solution to the modern education institutions.

Area of Study: Handwritten Text Recognition, Mobile Application Development

Keywords: Handwritten Text Recognition, CNN-BiLSTM-CTC, Deep Learning, Optuna Optimization, Android Mobile Application, Administrator Web Platform, Attendance Automation, FastAPI, Firebase Firestore, SharedPreferences

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
ASGI	Asynchronous Server Gateway Interface
BiLSTM	Bidirectional Long Short-Term Memory
CER	Character Error Rate
CLAHE	Contrast Limited Adaptive Histogram Equalization
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSV	Comma-Separated Values
CTC	Connectionist Temporal Classification
DB	Database
FYP	Final Year Project
GPU	Graphics Processing Unit
GPS	Global Positioning System
HTR	Handwritten Text Recognition
HTTP	HyperText Transfer Protocol
HTTPS	HTTP Secure
IAM	Institute of Advanced Studies
IDE	Integrated Development Environment
iOS	iPhone Operating System
JSON	JavaScript Object Notation
MNIST	Modified National Institute of Standards and Technology
NoSQL	Not Only Structured Query Language
NVMe	Non-Volatile Memory Express
OCR	Optical Character Recognition
OS	Operating System
RAM	Random Access Memory

RBF	Radial Basis Function
ReLU	Rectified Linear Unit
REST	Representational State Transfer
RESTful	Representational State Transfer-compliant
RNN	Recurrent Neural Network
SDK	Software Development Kit
SQL	Structured Query Language
SSD	Solid State Drive
SVM	Support Vector Machine
TPE	Tree-structured Parzen Estimator
UI	User Interface
UTAR	Universiti Tunku Abdul Rahman
VRAM	Video Random Access Memory
WER	Word Error Rate

Chapter 1
Introduction

1.1 Problem Statement and Motivation

In most university final examinations, attendance is still verified manually by invigilators through the collection of attendance slips, followed by manual data entry into the system. This process is time-consuming, error-prone and disruptive, particularly in large halls with hundreds of students. To overcome this, deep learning offered a promising solution through Handwritten Text Recognition (HTR), which is a technology that automatically identifies and converts handwritten input into machine-readable digital text, enabling automatic extraction from attendance slips and direct upload to the system. However, the current HTR systems have two severe limitations that make them unreliable in this context.

The first challenge is handwriting variability, arising from inconsistencies in stroke thickness, slant, letter formation, and cursive connections [1], [2], [3]. For instance, the character “r” may look like “n”, “u” may be confused with “v”, “l” can be misidentified as “1”, and “O” may resemble “0” [1], [2], [3], as shown in Figure 1.1 and Figure 1.2. This leads to misinterpretations during actual examinations and results in invalid or delayed attendance.

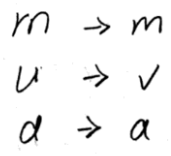

Figure 1.1 Common misinterpretations

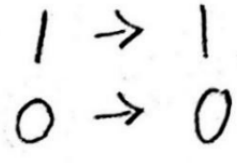

Figure 1.2 Ambiguities between “l” / “1” and “O”/ “0”

The second challenge is the limited scope of characters supported, as many existing solutions recognise only digits or letters but not both. For example, Wang et al. [4] developed a CNN-Transformer based HTR model for English text using the TAL_OCR_ENG dataset, which is not explicitly trained for digits. Likewise, Chauhan et al. [5] proposed a CNN model on the MNIST dataset, which supports digits only. Table 1.1 highlights these limitations in recognising letters compared to digits. This restricted character set is problematic for real-world applications such as attendance slips, which commonly require a mix of uppercase letters, lowercase letters, and digits, as seen in index numbers and course codes.

Table 1.1 Character recognition comparison across HTR models

Model	Dataset	Letters	Digits
CNN + Transformer [4]	TAL_OCR_ENG	Yes	Implied
CNN [5]	MNIST	No	Yes

With these limitations, the current HTR systems cannot be directly used to automate attendance marking in exam environments. Therefore, this project was motivated to develop a domain-specific HTR solution that is both reliable and efficient. Integrated with an Android mobile application for practical use, it can be used to process attendance slips electronically, eliminating invigilator workload and minimising data entry errors.

1.2 Objectives

The aim of this project was to develop an automated attendance marking system for university examinations using deep learning-based HTR technology that addresses the challenges of handwriting variability and limited character scope. A domain-specific HTR solution was therefore created for exam attendance slips and integrated with an Android application to enable accurate extraction of student information.

To achieve the main objective, the study undertook the following sub-objectives:

1. To Propose a CNN-BiLSTM-CTC-based HTR Framework

The end-to-end HTR model was designed to identify full-line handwritten fields on attendance slips without performing character-level segmentation, thus eliminating two major challenges. A Convolutional Neural Networks-Bi-directional Long Short-Term Memory-Connectionist Temporal (CNN-BiLSTM-CTC) architecture was used to address the variability of handwriting, including cursive, inclined, and irregular handwriting types. To overcome limited character scope, a specialised character set was targeted, consisting of uppercase letters, lowercase letters, digits, and spaces commonly found on exam attendance slips.

2. To Implement a Trainable HTR Model with API Deployment

The CNN-BiLSTM-CTC model was constructed and trained on a hybrid dataset, a combination of publicly available handwriting datasets and custom exam attendance slips, to guarantee that a variety of handwriting styles and alpha numerical characters are covered. A preprocessing pipeline and data augmentation were applied to improve robustness. Hyperparameter optimisation was performed to enhance model performance. The model was then integrated as an API service, which was accessible from an Android mobile app to perform handwriting recognition from attendance slips.

3. To Evaluate HTR Model Accuracy and Practical Usability

The HTR model's effectiveness was evaluated through both quantitative and qualitative assessments. Quantitatively, the model was measured by Character Error Rate (CER), Word Error Rate (WER) and Exact Match Accuracy, with the goal of achieving low CER and WER and high Exact Match Accuracy. Qualitatively, the model's predictions were reviewed to assess its robustness across diverse handwriting styles and mixed alphanumeric characters. The practical usability of the model was also demonstrated through API deployment and integration with an Android application for automated attendance marking.

1.3 Project Scope and Direction

This project focused on developing a deep learning-based HTR model to automate student attendance tracking during university final examinations, which would be accessed through an Android mobile application. The solution was designed to address two challenges facing existing HTR systems in the context of examination attendance, namely handwriting variability and limited character support.

For the model, an end-to-end HTR model based on CNN-BiLSTM-CTC architecture was built and trained on a hybrid dataset consisting of open-source handwriting datasets and real exam attendance slips, ensuring coverage of diverse handwriting styles and alphanumeric characters. The images were processed through a standardised preprocessing pipeline, and various data augmentation techniques were used to improve model adaptability to different

handwriting styles. The optimised and trained model was then integrated as an API with mobile devices.

The project's outcomes included a mobile application for students to upload scanned attendance slips that communicates with the HTR API to extract and store attendance data in Firebase Firestore, and a website platform for administrators to manage exams, students, and attendance. iOS was not supported because of the platform restrictions and cost, as focusing on Android ensured wider accessibility and development speed.

Overall, this project involved dataset preparation, model implementation, API and database implementation, as well as both mobile and web application development, providing an applied solution that increases efficiency and accuracy of examination attendance.

1.4 Contributions

The project was a contribution toward enhancing the examination attendance tracking in universities by integrating a deep learning-based HTR technology in an Android-based application. First, it addressed two key challenges in HTR within the context of examination attendance processing, which are handwriting variability and character set limitations. Through a training set that includes digits, upper and lower case letters, and different handwriting styles, the model could successfully and consistently extract student information from handwritten slips. Second, it demonstrated the integration and deployment of the HTR model accessible through a mobile application, allowing students to scan attendance slips and enabling automatic information extraction, thereby reducing errors and reliance on manual verification. Finally, the project led to the greater digitalisation of the sphere of educational administration, as it offers a scaling framework that can be applied to the areas of assignment checking, student registration, and record-keeping.

1.5 Report Organization

This report is organised into 7 chapters. Chapter 1 describes the problem, objectives, scope and contributions of the project. Chapter 2 provides a survey of HTR approaches and the research

limitations that this project intends to address. Chapter 3 explains the system methodology, covering the development process, workflow, system design diagrams, and test case design. Chapter 4 presents the system design, including the hardware, software, data preparation, model, API, database and user interfaces. Chapter 5 describes the system implementation, including data processing, model training, deployment and system operation. Chapter 6 evaluates the system, detailing the test and results of the model, API and mobile application, as well as challenges and objective evaluation of the project. Chapter 7 provides the conclusion of the report and discusses future works.

Chapter 2 Literature Reviews

2.1 Previous Works

As deep learning strides have been made, researchers are moving into more advanced instances of deep learning in HTR. What began as simple feature-based methods has transitioned into more advanced end-to-end methods that analyse difficult patterns in handwriting. The following subsections review specific projects and methodologies used to develop HTR technologies.

2.1.1 Cursive Handwriting Recognition Using CNN with VGG-16

Rangari et al. [6] proposed a CNN-VGG16 model to recognise cursive handwritten English text. The method used data augmentation, image segmentation, and an image data generator to enhance input quality and minimise overfitting. The IAM Handwriting Database was used, containing 13,353 handwritten line images from 657 writers with 115,320 words.

The VGG-16 model, illustrated in Figure 2.1, has 13 convolutional layers with 3 fully connected layers, using 3×3 filters and 2×2 pooling to capture detailed hierarchical features for cursive handwriting recognition.

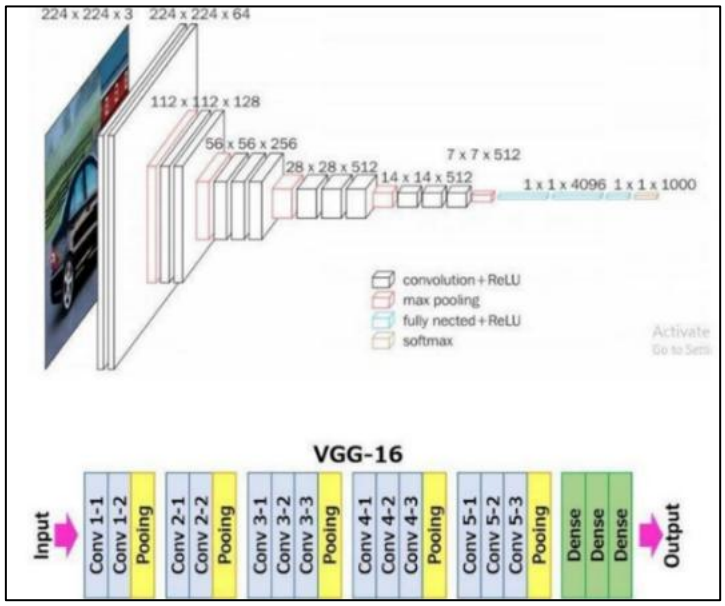

Figure 2.1 VGG-16 system architecture [6]

Three experiments were performed using one preprocessing technique, combining two techniques, and combining all three. As shown in Table 2.1, data augmentation alone achieved the best result with 98.36% training accuracy and 95.1% validation accuracy.

Table 2.1 VGG-16 experimental results [6]

Expt. No	Deep Learning method used	Training accuracy	Validation accuracy
1	CNN with VGG16 and data augmentation	98.36%	95.1%
	CNN with VGG16 and image segmentation	95.25%	91.8%
	CNN with VGG16 and image data generator	92.47%	89.36%
2	CNN with VGG16 and data augmentation & image segmentation	96.02%	92.1%
	CNN with VGG16 and image segmentation & data image generator	94.75%	90.95%
	CNN with VGG16 and data augmentation & image data generator	93.27%	90.67%
3	CNN with VGG16 and data augmentation, image segmentation & image data generator	97.85%	94.58%

The main innovation of this work is the systematic evaluation of preprocessing techniques for cursive handwriting recognition, demonstrating that data augmentation alone is more effective than combining multiple techniques. The study provides practical insights into how VGG-16, a deep CNN architecture originally designed for image classification, can be adapted for handwriting recognition tasks.

2.1.2 Hybrid CNN-SVM Model for Handwritten Digit Recognition with Application to Cerebral Palsy

Jetlin et al. [7] proposed a hybrid CNN-SVM model that combines CNN to extract hierarchical features and a Support Vector Machine with an RBF kernel to classify them, as shown in Figure 2.2. This hybrid architecture will address the shortcomings of traditional models that are unable to cope with the varied writing styles and contextual complexities.

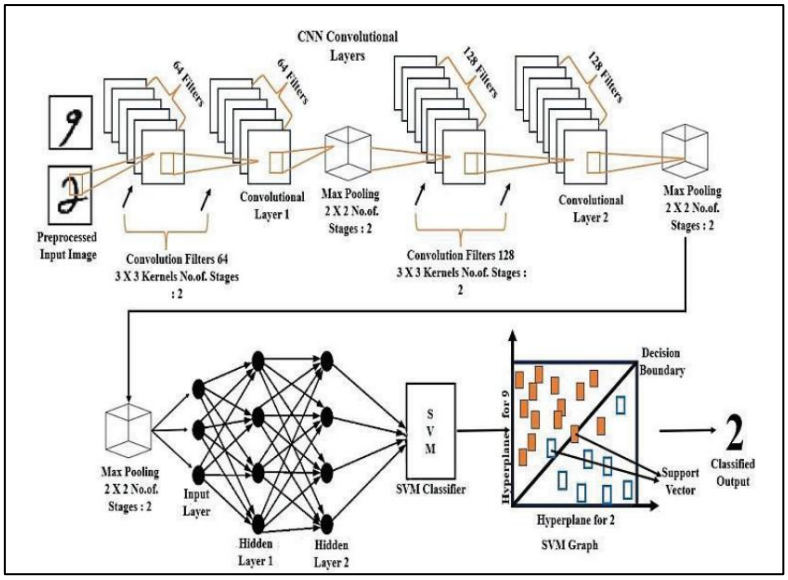

Figure 2.2 Proposed Hybrid CNN-SVM architecture [7]

The workflow involves collecting data based on standardised datasets such as MNIST, and image pre-processing using size normalisation followed by smoothing using a Gaussian kernel as shown in Figure 2.3. The CNN layers extract features, and the feature vectors are classified using SVM. Training is conducted using gradient descent and backpropagation for CNN, and grid search and cross validation for SVM.

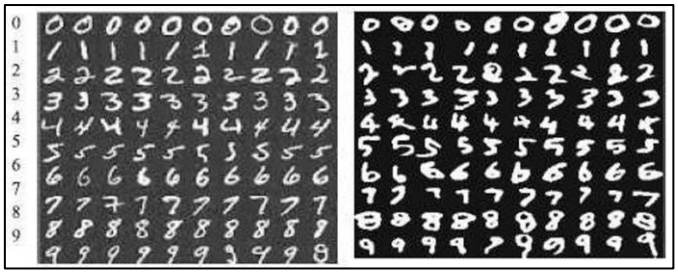

Figure 2.3 Cerebral Palsy MNIST Dataset

Evaluated on different writing styles, the CNN-SVM achieved 93.5% recognition for cursive handwriting, 96.7% for print, and 91.2% for mixed styles, as shown in Table 2.2.

Table 2.2 Writing style recognition accuracy [7]

Writing Style	Accuracy
Cursive	93.5%
Print	96.7%
Mixed Styles	91.2%

The main innovation of this work is the successful integration of a traditional machine learning classifier with a deep learning feature extractor, demonstrating that a hybrid machine learning and deep learning approach can effectively recognise diverse handwriting styles without the need for manual feature engineering.

2.1.3 Exploring Recursive Neural Networks for Compact Handwritten Text Recognition Models

Mas-Candela and Calvo-Zaragoza [8] developed Recursive Neural Networks (RecNN) for memory-efficient HTR. Unlike standard models where each layer has its own parameters, RecNN reuses the same weights across multiple layers. This reduces model size significantly while maintaining similar accuracy.

As illustrated in Figure 2.4, conventional layers use independent weights for each layer, while recursive layers reuse the same weights across successive layers. This weight-sharing mechanism allows greater depth without increasing parameter count. The authors further proposed kernel scaling, shown in Figure 2.5, where a learnable multiplier adjusts the convolutional kernel at each recursive call. This technique allows recursive layers to achieve greater expressiveness with only one additional weight per layer.

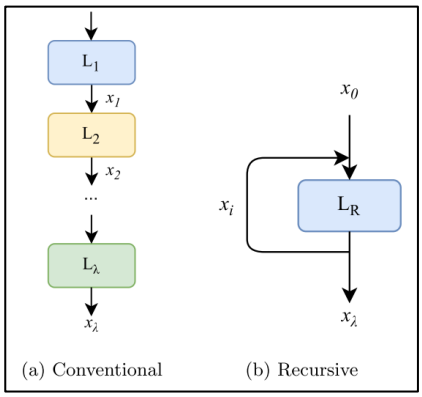

Figure 2.4 Comparison of conventional layers and a recursive [8]

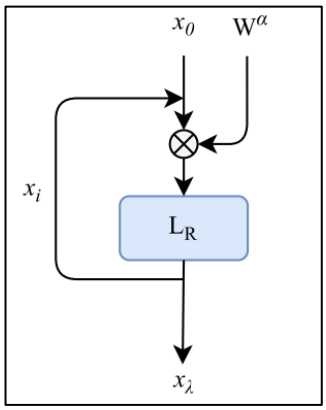

Figure 2.5 Kernel scaling technique [8]

The model was then tested on the IAM dataset. A RecNN configuration with 340,000 parameters achieved 10.5% CER, while a larger configuration with 3.8 million parameters achieved 6.4%. The authors found that recursive models often achieved optimal trade-offs between accuracy and memory usage.

The main innovation of this work is the weight-sharing mechanism through the RecNN, which greatly decreases model size while preserving competitive recognition accuracy. The kernel scaling technique further enhances expressiveness with minimal parameter overhead, making RecNN particularly suitable for memory-constrained deployment scenarios.

2.1.4 Handwritten Text Recognition using Deep Learning Algorithms

Ansari et al. [9] introduced a hybrid CNN-RNN model combined with CTC for handwritten English text recognition, integrating CNN for feature extraction, bidirectional LSTM for sequence modelling, and CTC for handling unsegmented sequences. The IAM dataset was employed for experimentation, containing over 800,000 samples of handwritten words, letters, and numerals written in diverse styles, as shown in Figure 2.6.

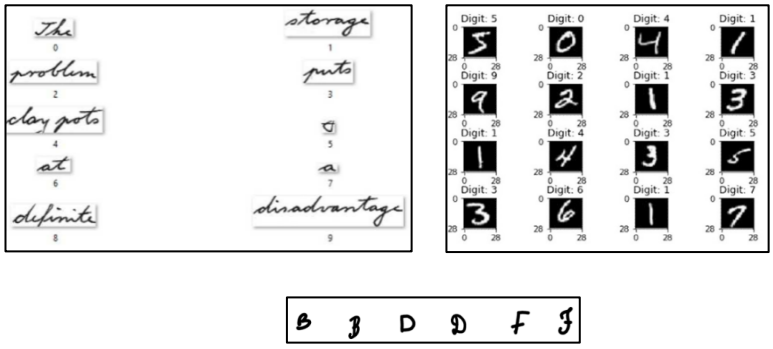

Figure 2.6 Sample images from IAM dataset: words, digits, and alphabet [9]

The model architecture, illustrated in Figure 2.7, comprises five CNN layers for hierarchical feature extraction, together with a two-layer bidirectional LSTM network to capture sequential context, and a CTC layer for alignment-free decoding.

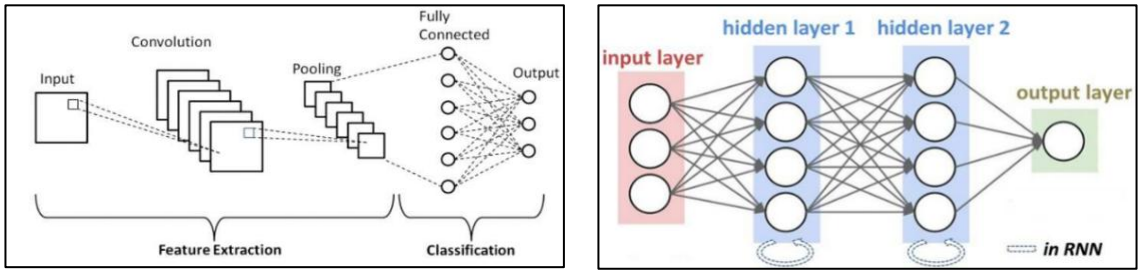

Figure 2.7 CNN and BiLSTM layers architecture [9]

This model achieved 95.1% accuracy after 20 epochs, with the loss dropping from 13.66 to 0.91, as illustrated in Figure 2.8 and Figure 2.9. Figure 2.10 shows that testing on

random images demonstrated that most predictions were quite accurate despite some errors, proving the model's ability to handle different handwriting styles [9].

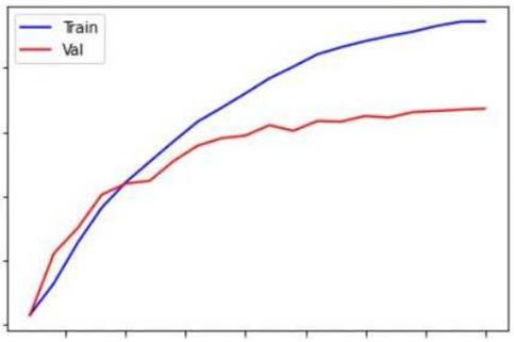

Figure 2.8 Accuracy of the model [9]

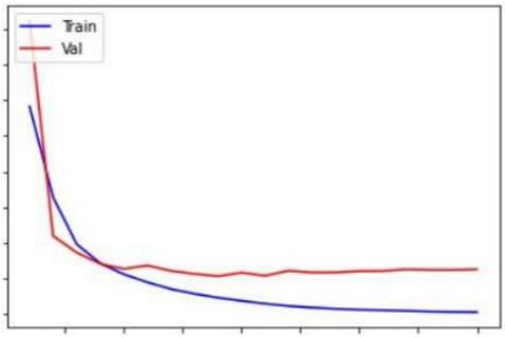

Figure 2.9 Loss of the model [9]

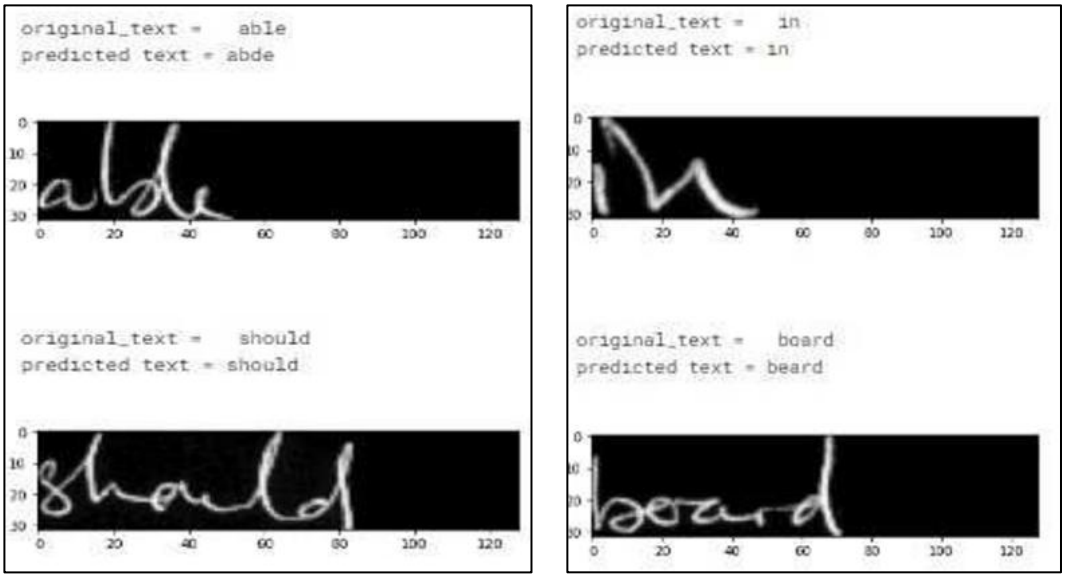

Figure 2.10 Result images [9]

The main innovation of this work is the seamless integration of CNN for spatial feature extraction, bidirectional LSTM for sequential context modelling, and CTC for alignment-free sequence recognition. This model can perceive handwriting in a more holistic way without the explicit need to break up characters explicitly, unlike traditional methods that recognise isolated characters one by one.

2.2 Strength and Weakness of Previous Works

Table 2.3 Strengths and weaknesses of previous works

Study	Architecture	Dataset	Strengths	Weaknesses
Rangari et al. [6]	VGG-16 (Pure CNN)	IAM (word-level)	Achieved 95.1% validation accuracy on IAM; systematically evaluated preprocessing techniques	Processes characters independently without sequential context; cannot handle cursive or connected writing; limited to isolated word recognition
Jetlin et al. [7]	CNN + SVM	MNIST (character-level)	Achieved 93.5% accuracy on cursive and 91.2% on mixed styles; hybrid approach reduces manual feature engineering	SVM cannot model sequential dependencies; requires explicit character segmentation; limited to MNIST digits only
Mas-Candela & Calvo-Zaragoza [8]	RecNN	IAM (line-level)	Memory efficient with weight sharing (340K parameters achieved 10.5% CER); kernel scaling enhances expressiveness	Weight sharing limits adaptability to diverse handwriting styles; only covers A-Z, a-z without digit support
Ansari et al. [9]	CNN-BiLSTM-CTC	IAM (word-level)	Achieved 95.1% accuracy on IAM; bidirectional LSTM captures sequential context; CTC enables alignment-free recognition	Limited to word-level recognition; cannot handle multi-word text lines with spaces; lacks digit recognition capability

2.3 Limitations of Previous Works

HTR systems have made great advancements in digitising and interpreting human handwriting. Nevertheless, their universal application and effectiveness are limited by some serious constraints, including variability of handwriting styles and range of characters, which are critical challenges to their wider adoption.

2.3.1 Handwriting Variability

Rangari et al. [6] developed a VGG-16 based CNN architecture for cursive handwriting recognition. Despite its effectiveness, the pure CNN model processes each character independently without sequential context. When confronted with slanted letters, irregular spacing, or overlapping strokes, the model cannot disambiguate ambiguous characters using surrounding information. A poorly written "e" may be misclassified as "a" simply because the model lacks access to neighbouring character context [6]. Jetlin et al. [7] proposed a hybrid CNN-SVM model, but the SVM classifier only operates on fixed-size feature vectors and cannot model sequential dependencies between characters. This approach also requires explicit character segmentation, which is problematic for connected handwriting [7]. Mas-Candela and Calvo-Zaragoza [8] presented RecNN with weight sharing for memory efficiency. However, reusing the same weights across recursive layers restricts the model's adaptability to diverse handwriting styles, as the same transformation is applied repeatedly across all strokes regardless of their variation [8].

In contrast, Ansari et al. [9] proposed a CNN-BiLSTM-CTC architecture that directly addresses handwriting variability. The CNN layers extract spatial features and stroke patterns from handwritten input, while the bidirectional LSTM layers capture sequential context from past and future characters, enabling the model to resolve ambiguous characters using surrounding information. The CTC component eliminates the need for explicit character segmentation, allowing word-level recognition on the IAM dataset [9]. This architecture effectively overcomes the limitations of pure CNN, CNN-SVM, and RecNN models by jointly modelling spatial and sequential information. However, as a word-level model, it cannot directly handle multi-word text lines containing spaces. This gap motivated the need for line-level recognition in the proposed solution.

2.3.2 Limited Character Scope

Rangari et al. [6] and Ansari et al. [9] used the IAM dataset, which mainly comprises English word transcriptions with a small number of digits. Therefore, these models lack enough exposure to numerical patterns to learn how to recognise them. This limitation directly impacts their ability to accurately extract numeric fields such as index numbers in figures or seat numbers from exam attendance slips. Mas-Candela and Calvo-Zaragoza [8] evaluated their RecNN model on IAM line-level samples, which include uppercase and lowercase letters as well as spaces, but this model still lacks digit recognition capability. Jetlin et al. [7] evaluated their CNN-SVM model on the MNIST dataset, which consists solely of digits. While this model can recognise isolated digits, it completely lacks letter recognition capability, rendering it incapable of handling alphanumeric fields such as subject names that require mixed character recognition. None of these previous studies address the need for unified line-level alphanumeric recognition where uppercase letters, lowercase letters and digits need to be recognised within the same text line. This motivated the need for a hybrid dataset that includes both letters and digits as a unified character set.

2.4 Proposed Solutions

The aim of the project was to improve the robustness and applicability of HTR systems by addressing two key limitations noted in the prior works: variability of handwriting and limited scope of characters. The suggested solutions are tailored to the situation of the university examination attendance automation, where handwritten attendance slip must be entered correctly and efficiently.

2.4.1 Addressing Variability in Handwriting Styles

To deal with handwriting variability, this project built upon the work of Ansari et al. [9] using a hybrid CNN-BiLSTM-CTC architecture while extending it from word-level to line-level recognition. CNNs learn spatial features from handwritten images, and BiLSTMs learn temporal features about the sequence of characters. The CTC component aligns input images with text outputs without explicit segmentation, enabling recognition of complete text lines, including spaces. This architecture effectively overcomes the limitations of previous

approaches, such as those proposed by Rangari et al. [6], Jetlin et al. [7], and Mas-Candela and Calvo-Zaragoza [8], which lack sequential modelling capabilities.

2.4.2 Expanding the Scope of Supported Characters

To address the limited character coverage in previous studies [6], [7], [8], [9], this project constructed a hybrid dataset that combined real-world university exam attendance slips with publicly available handwriting datasets. The custom dataset was derived directly from UTAR exam attendance slips, containing handwritten index numbers, seat numbers, and subject names that naturally mix uppercase letters, lowercase letters, and digits within full text lines. To further improve generalisation, the IAM dataset [10] was incorporated to provide varied English handwriting samples including spaces, while the Multi-digit MNIST dataset [11] was added to supply numeric sequences. Training on this combined dataset enabled the system to accurately recognise characters as they appear in real exam scenarios, improving the extraction of critical fields such as index numbers, seat numbers, and subject names with higher accuracy and contextual reliability at the line-level.

2.5 Comparison of Previous Works and Proposed Solutions

Table 2.4 Strengths and weaknesses of previous works

Study	Architecture	Handles Handwriting Variability	Dataset	Character Set Coverage	Domain-Specific
Rangari et al. [6]	VGG-16 (Pure CNN)	No (isolated characters)	IAM (word-level)	No (A-Z, a-z)	No
Jetlin et al. [7]	CNN + SVM	No (requires segmentation)	MNIST (character-level)	No (0-9)	No
Mas-Candela & Calvo-Zaragoza [8]	RecNN	Partial (limited adaptability)	IAM (line-level)	No (A-Z, a-z, space)	No
Ansari et al. [9]	CNN-BiLSTM-CTC	Yes	IAM (word-level) (A-Z, a-z)	No (word-level)	No
Proposed Solution	CNN-BiLSTM-CTC	Yes	UTAR Slips + IAM + Multi-digit MNIST (line-level)	Yes (A-Z, a-z, 0-9, space)	Yes (UTAR slips)

Chapter 3

System Methodology/Approach

3.1 Development Methodology

Agile methodology is an iterative and incremental approach that focuses on rapid delivery, feedback and planning. As shown in Figure 3.1, Agile divides a project into small phases called sprints, allowing adjustments based on ongoing evaluation rather than following a fixed plan [12]. Therefore, Agile was adopted for this project, which comprises a full system stack including the HTR model, API backend, database, mobile application, and web platform.

On the model side, development involved continuous cycles of training, validation, optimisation, and evaluation. Agile enabled iterative adjustments to achieve optimal model performance. For other system components, features were developed incrementally for early testing and feedback. In brief, the project was carried out in sprints and assessed at the end of each sprint to prioritise tasks, so the final system would be accurate, reliable, and provide an efficient automated attendance marking workflow.

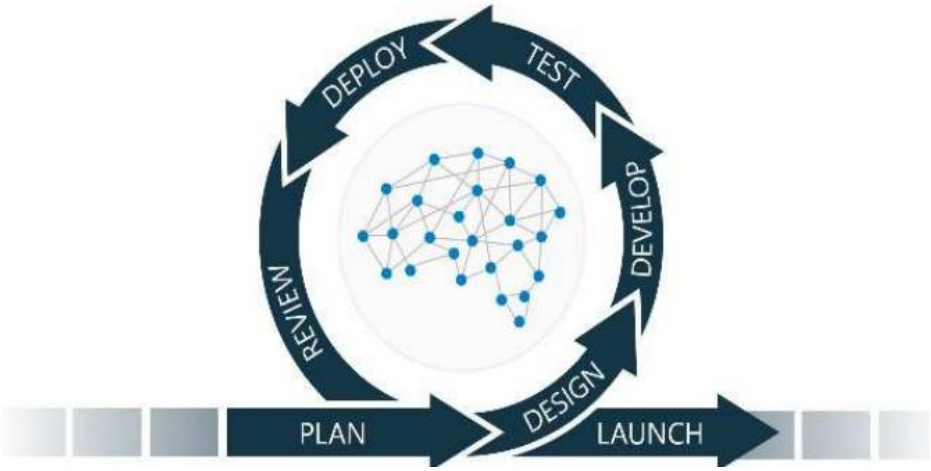

Figure 3.1 Agile development cycle [12]

3.2 Project Workflow

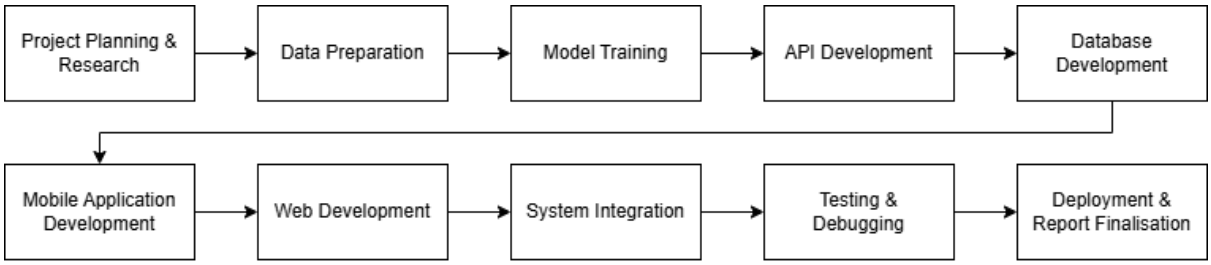

Figure 3.2 Project workflow

Figure 3.2 shows the project workflow. The project began with planning and research, where problem statements were identified and HTR literature was reviewed, focusing on handwriting variability and limited character support. Based on this review, the project scope, objectives, proposed solution, methodology, and initial design of the deep learning model, mobile application, and website platform were determined.

Following the planning, a custom dataset was constructed from exam attendance slips and integrated with public datasets, followed by preprocessing and data augmentation. Then the model was constructed, trained, optimised and tested. Once the optimal model was achieved, a RESTful API was developed to host the model, which received an image via POST request and returned the recognised text in JSON format. A secure data storage mechanism with offline support and real-time synchronisation was also established. After having the API and the database, an Android application was built for image capture and attendance submission, while a website platform was developed for administrators to manage exams, students, and attendance records.

All components were then integrated to facilitate the entire automated attendance process of image capture to data storage and synchronisation. Key functionalities were iteratively tested. Finally, once major bugs were resolved, the system was deployed and the project was completed with documentation.

3.3 System Design Diagram

This section offers visual aids of system structure, component interactions, and user-system relationships, acting as a gateway before the actual implementation details are presented.

3.3.1 System Architecture Diagram

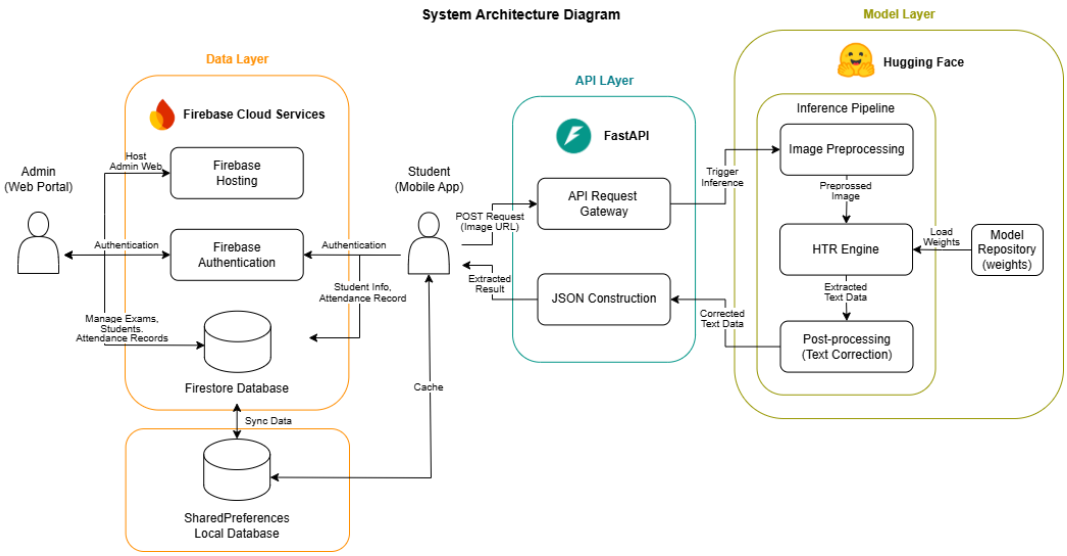

Figure 3.3 System architecture diagram

The system architecture diagram in Figure 3.3 shows the interaction between the user interface and system processes. On the client side, the architecture defines customised interfaces for students via mobile application and administrators via website platform. The data layer manages all data persistence and synchronisation. SharedPreferences handles local caching, while Firestore provides cloud storage, with automatic synchronisation between them. The data layer also uses Firebase Hosting for web deployment and Firebase Authentication for role-based access control.

The server-side architecture centres on the API and model layers. Built with FastAPI, the API layer acts as a bridge between the mobile application and the inference engine, handling POST requests and data exchange. Upon receiving a request, the model layer hosted on Hugging Face runs an automated pipeline that preprocesses images, applies the HTR engine for text extraction, and performs post-processing for correction. The delegation of these complex tasks from the client-side allows for smooth processing and data transformation from scanned attendance slips to digital records.

3.3.2 Use Case Diagram and Description

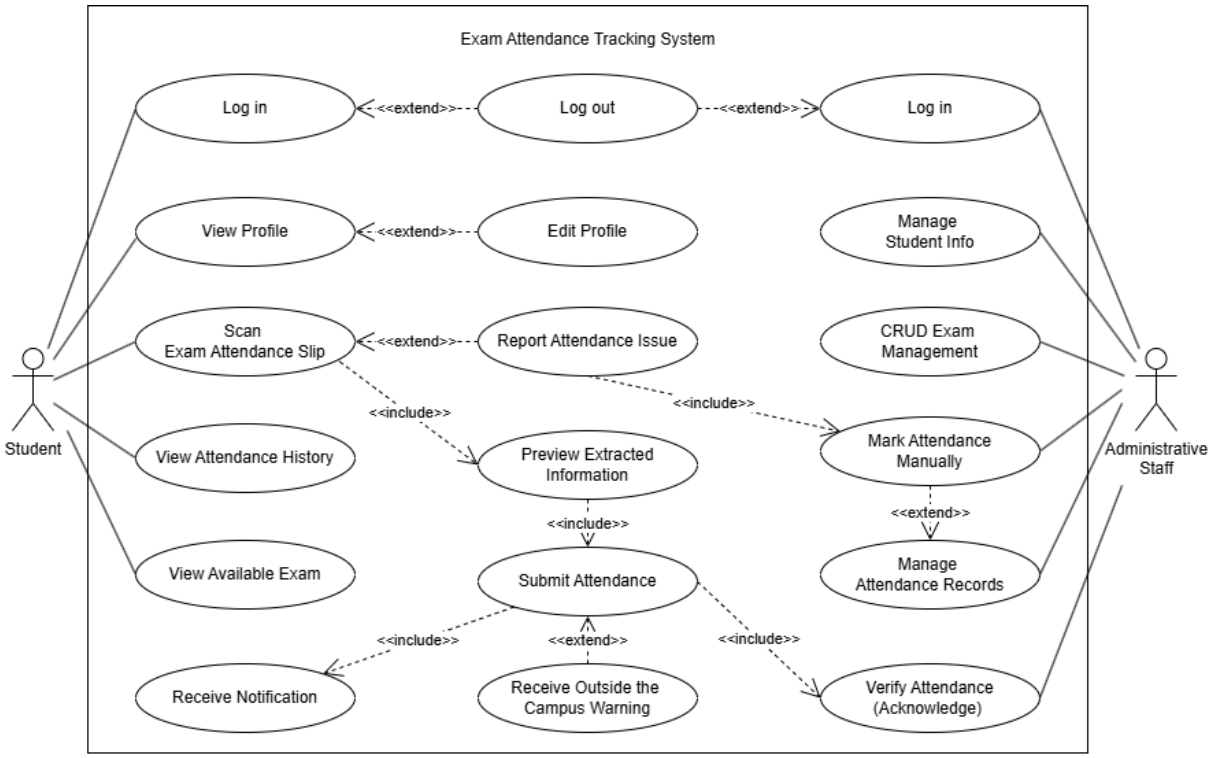

Figure 3.4 Use case diagram of the system

Table 3.1 Use case description of the system

Use Case Name:	Scan Exam Attendance Slip	
Triggering Event:	Student scans and uploads the attendance slip through the mobile application	
Brief Description:	This use case explains how a student scans the attendance slip; the system uses HTR to extract the attendance slip data and cross-validates to mark the attendance with geofencing check.	
Actor:	Student	
Pre-condition:	1. Student is authenticated and successfully logged into the application. 2. An active exam session has been initiated by the Administrative Staff. 3. Student has previously uploaded their Index Number in the Profile section for verification.	
Post-condition:	A validated attendance record is successfully generated in the Firebase database. Student's attendance status is updated to "Present" (with a location flag).	
Normal flow:		
Step	Actor Action	System Response
1	Student navigates to the scan page to activate the camera.	System displays the camera interface for scanning.
2	Student captures an image of the exam attendance slip.	System receives the image file.
3		System invokes the HTR API service to perform image preprocessing, text extraction, and post-processing.
4		System displays the Preview Recognized Information screen with the extracted data.
5	Student reviews the information and confirms the accuracy of the data.	System enables the submission action.

6	Student clicks the Submit Attendance button.	System verifies: 1. Subject is active and assigned to the student. 2. Index No (Figures) matches Index No (Words). 3. Extracted Index No matches the registered Index No. 4. GPS coordinates for campus proximity.
7		System synchronizes the validated data with the cloud database.
8		System displays a success message and sends a notification.
Alternative flow:		
2a. If the student fails to scan or upload the slip, they may report the issue to the invigilator to request that the administrators mark the attendance manually. 6a. If the Subject is not assigned to the student, the Index Number (Figures) does not match the Index Number (Words), or the Index Number does not match the registered Index Number, the system flags the error and asks the student to scan again or edit the information manually. 8a. If the student is detected outside the campus coordinates (UTAR Kampar/Sg Long), the system records the attendance but marks the entry with an "Outside the UTAR Campus" flag for administrative review.		

Figure 3.4 illustrates the use case diagram for the project, defining interactions between students and administrators. For students, the workflow centres on scanning attendance slips, which includes automated extraction, data preview, and submission as detailed in Table 3.1. After submission, a geofencing check is performed, with a warning issued if the scan occurs outside campus to prevent fraud. Students receive a notification once data is successfully uploaded. Beyond scanning, students can maintain profiles, view schedules, and track attendance records.

The administrator manages system integrity through CRUD operations for exam management, student records, and attendance records. After an exam session ends, they verify and acknowledge attendance records to confirm validity. In exceptional cases where a student reports an attendance problem, they can also mark attendance manually on the website. This architecture makes sure that while attendance marking is automated for a smooth student experience, the administrative component provides both routine verification and exception handling to maintain accurate and secure records.

3.3.3 Activity Diagram

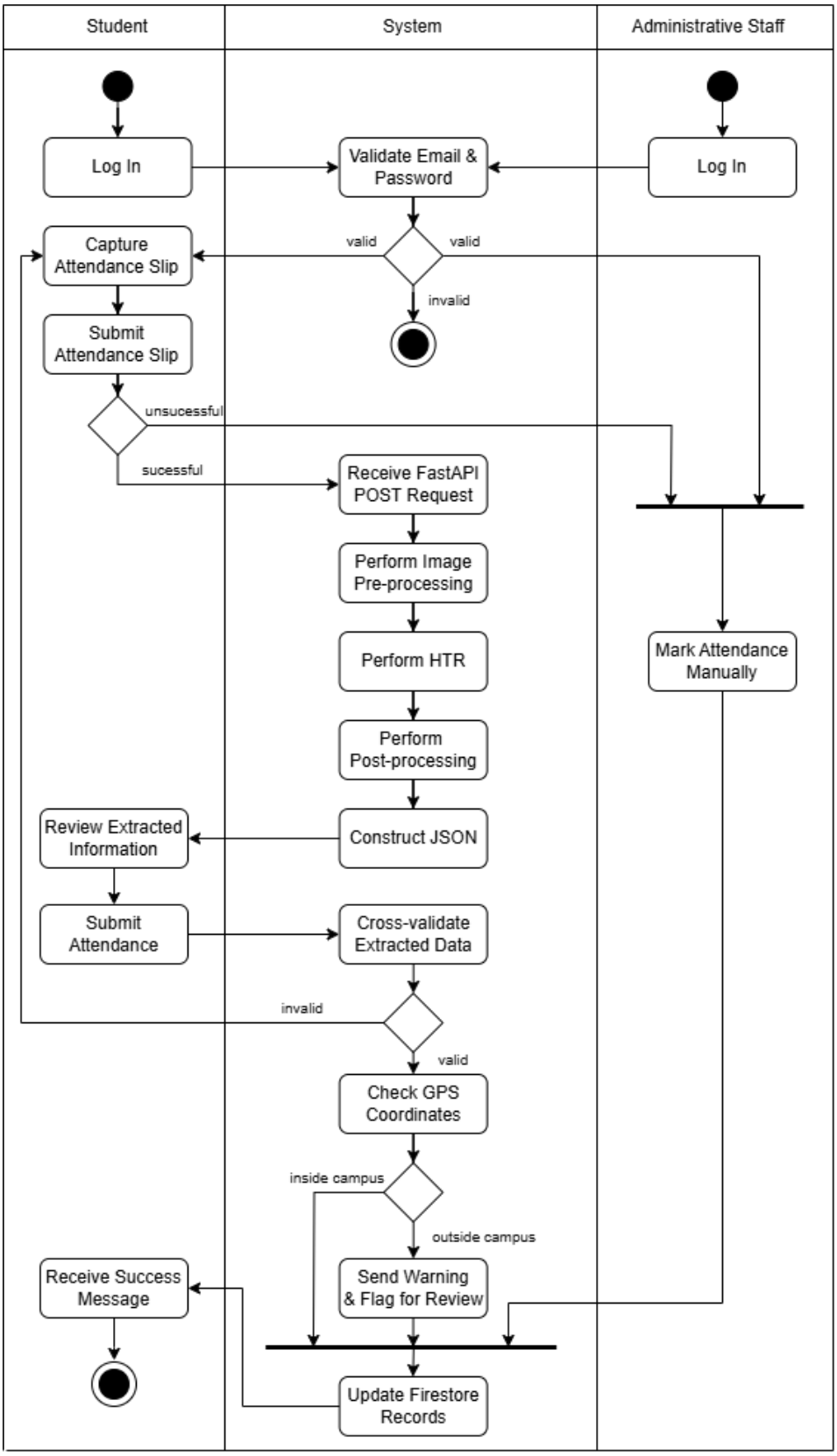

Figure 3.5 Activity diagram of the system

The activity diagram in Figure 3.5 starts with a student or admin logging into the system. An email and password are first verified. An authentication failure will end the flow and prompt the user to retry logging in. If authentication is successful, the student then takes and sends the attendance slip. In case of any error, the student can ask the administrator to manually mark the attendance. The administrator then manually updates the Firestore record and the flow is complete.

For a successful automated submission, the system sends a POST request to the FastAPI backend, which performs image preprocessing, HTR recognition, and post-processing. The extracted data is returned to the student for confirmation. Once confirmed, the system validates the index number and exam assignment. A validation failure will prompt the student to recapture the slip. If valid, the system checks the GPS location. In case it is not recorded on campus, a warning is sent to the administrator to check and verify but the attendance is still registered. Lastly, the Firestore database is updated, a confirmation message is displayed to the student and the flow is complete.

3.4 Software Testing Design

Structured testing strategies were designed to validate the HTR model, API, and mobile application, ensuring the overall system is accurate and functions correctly across all components.

3.4.1 Model Evaluation Design

To ensure the HTR model is reliable for automated attendance marking in university examinations, the model was evaluated on custom UTAR attendance slips not seen during training. The evaluation combined quantitative and qualitative approaches. Quantitatively, CER, WER and Exact Match Accuracy were calculated to measure recognition accuracy. Qualitatively, predicted outputs were compared against ground truth labels to review recognition results, and misclassified characters were analysed to identify common error patterns.

3.4.2 API Testing Design

To ensure the application works seamlessly with the model, the API bridge was verified to confirm it could handle different types of requests while maintaining acceptable response times. Four API test cases were designed to validate error handling, as outlined in Table 3.2.

Table 3.2 API test case design

Test Case ID	Test Case Name	Input	Expected Output
TC-API-01	Valid image upload	JPG image of attendance slip	JSON with extracted fields
TC-API-02	Missing file field	POST request without file	HTTP 422 error
TC-API-03	Invalid image format	Text file instead of image	HTTP 400 error
TC-API-04	Empty image	Blank white image	Empty string fields

3.4.3 Mobile Application Testing Design

To ensure students can perform attendance marking conveniently and effectively, the entire submission workflow was validated through compatibility testing across different Android devices and network conditions, as well as functional testing with 5 test cases. Each functional test case was designed with a primary flow for the intended successful path and a secondary flow for error handling. Table 3.3 summarises the designed test cases.

Table 3.3 Mobile application test case design

Test Case ID	Test Case Name	Main Flow	Alternative Flow	Expected Result
TC-APP-01	Login	Valid credentials	-	Home screen displayed
TC-APP-02	Register new account	Valid registration	Existing student ID, invalid email, existing email, short password, mismatched password	Account created successfully or appropriate error message shown
TC-APP-03	Submit exam attendance slip	Valid submission	Index mismatch, subject mismatch, not enrolled, already marked, outside campus,	Success dialog with notification, or validation error dialog, or "Outside Campus" warning flag
TC-APP-04	Offline view cached data	Offline data access	-	Home screen displayed using cached data
TC-APP-05	Offline modify profile	Offline modification with auto-sync	-	Changes stored in sync_queue, auto-sync to Firestore when online

3.5 Gantt Chart

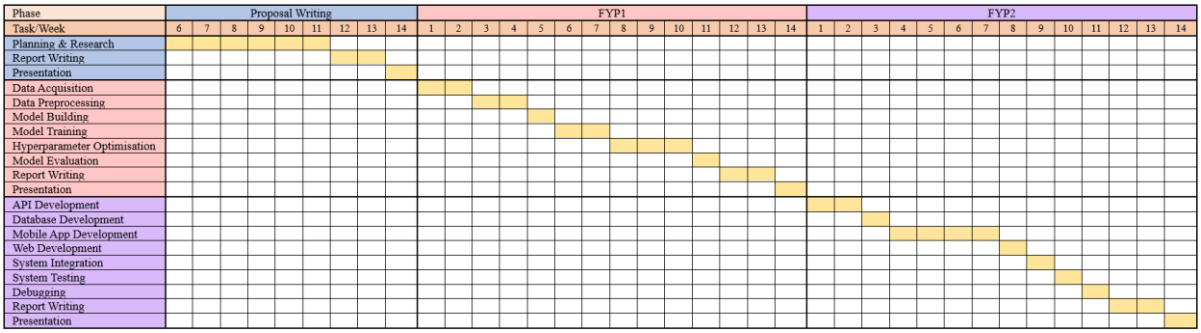

Figure 3.6 Gantt chart

As shown in Figure 3.6, the project was designed in three major phases. The first phase was Proposal Writing, which involved intensive research and methodology design to establish a solid theoretical foundation. This was followed by FYP1, a technical phase focused on the deep learning lifecycle, including data preparation and HTR model development to prove the practical feasibility of the solution. Finally, FYP2 finished the project through full-stack engineering, in which API, database, mobile application and website platforms were developed and integrated with the HTR model into a complete end-to-end system finalised through extensive testing and debugging.

Chapter 4
System Design

4.1 Hardware and Software Specifications

In constructing the automated attendance marking system, appropriate hardware and software were configured for the entire system stack.

4.1.1 Hardware

To facilitate seamless workflow and integration of the system components, the laptop was necessary in data collection, model implementation, API and database development, and cross-platform application development. Table 4.1 displays the laptop specifications.

Table 4.1 Development laptop specifications

Description	Specifications
Model	Asus TUF F15
Processor	Intel Core i5-10300H
Operating System	Windows 11
Graphic	NVIDIA GeForce RTX1650 with 4GB VRAM
Memory	16GB DDR4 RAM
Storage	1TB NVMe SSD

Android-based smartphones were required for the mobile application development, testing and deployment. The cameras were used to take photos for the attendance process. The minimum specifications of test mobile phone are tabulated in Table 4.2.

Table 4.2 Test mobile device specifications

Description	Specifications
Operating System	Android 8.0 (Oreo) or above
RAM	Minimum 2 GB RAM
Storage	At least 100 MB free space
Internet Access	Wi-Fi or mobile data
Camera	Minimum 8 MP camera with autofocus

4.1.2 Software

Table 4.3 summarises the software environment of each component.

Table 4.3 Software specifications

Component	Software / Library	Version	Description
HTR Model Development	Python	3.8	Primary programming language for model training
	TensorFlow [13]	2.8.0	Deep learning framework for CNN-BiLSTM-CTC model
	Keras	2.8.0	High-level API for building neural network layers
	OpenCV [14]	4.5.4	Image preprocessing and augmentation
	NumPy [15]	1.24.3	Numerical array operations
	Pandas [16]	1.5.3	Data manipulation and CSV handling
	Optuna [17]	3.1.0	Hyperparameter optimisation framework
	Jupyter Notebook	Latest	Interactive development environment
API Development	FastAPI	0.95.0	Web framework for RESTful API
	Uvicorn	0.21.0	ASGI server for running FastAPI
	EasyOCR [18]	1.7.0	Dynamic text detection for cropping
Mobile App Development	Flutter	3.0.0	Cross-platform UI framework
	Dart	2.17.0	Flutter Programming language
	Android Studio	Hedgehog	IDE for Android development and testing
Database	Firebase Firestore [19]	-	Cloud NoSQL database for attendance records
	SharedPreferences	Flutter plugin	Local key-value storage for caching
Web Platform	Flutter Web	3.0.0	Web deployment from Flutter codebase
	Firebase Hosting	-	Static web hosting for admin interface

4.2 Data Preparation

The data preparation stage was crucial, as it directly influenced the HTR model's ability to recognise diverse writing styles and character ranges. As illustrated in Figure 4.1, the pipeline worked through a systematic workflow: raw images were gathered from three sources, preprocessed, split into three subsets, then augmented, padded, and normalised.

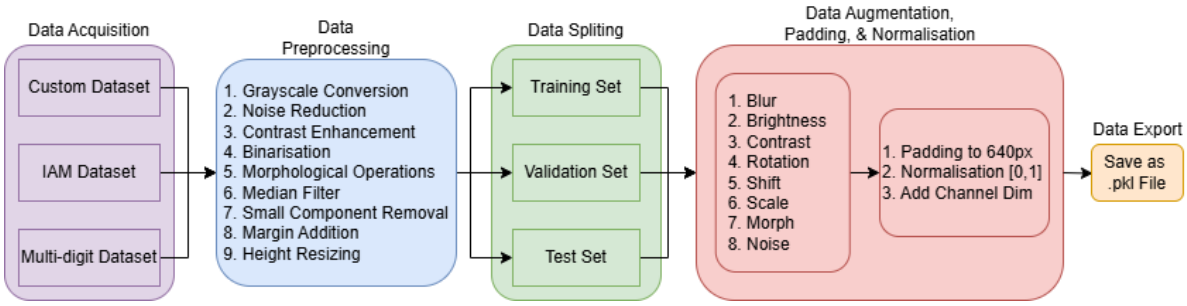

Figure 4.1 Data preparation pipeline

4.2.1 Data Acquisition

This project used both custom and open-source data to ensure domain-specific recognition, guarantee handwriting variability, and accommodate the limited character set supported by the model.

Custom Dataset

The custom dataset is derived from actual UTAR exam attendance slips, containing handwritten index numbers, seat numbers, and subject names with a mix of uppercase letters, lowercase letters, and digits, as illustrated in Figure 4.2. This domain-specific dataset captures diverse writing styles from the target examination environment, enabling the model to process real-world exam data reliably.

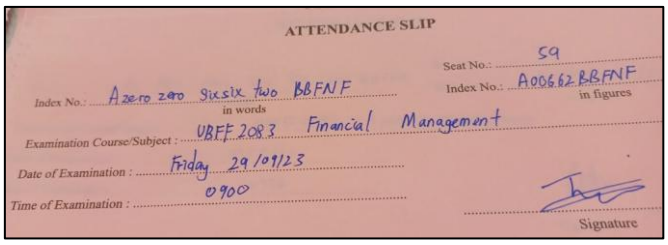

Figure 4.2 Sample UTAR exam attendance slip

IAM Dataset

The IAM dataset, developed by Ashish Goswami on Kaggle [10], consists of 5,389 handwritten English text lines containing uppercase and lowercase letters, along with some digits, in various writing styles, as illustrated in Figure 4.3. The scale of such a large dataset enhances generalisation of the model that addressed variability in handwriting and expanded recognition capabilities. It is particularly useful in identifying the subject names with its concise sentence structure.

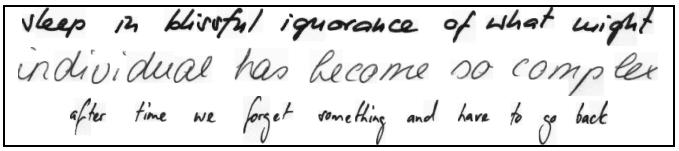

Figure 4.3 Sample IAM dataset

Multi-digit MNIST Dataset

The Multi-digit MNIST dataset, created by AbhinavPrasoon141 on Kaggle [11], consists of 3,000 multi-digit sequence images commonly found in exam attendance slips such as seat numbers, as presented in Figure 4.4. This dataset enables the model to handle purely numeric text lines effectively, rather than only alphabetic or alphanumeric content.

Figure 4.4 Sample Multi-digit MNIST dataset

4.2.2 Data Preprocessing

All images were preprocessed using a standardised pipeline to ensure consistency and increase the recognition accuracy before being inputted into the model. The preprocessing phase involved:

1. Grayscale Conversion: RGB images were converted to single-channel grayscale to reduce computation complexity.
2. Noise Reduction: Gaussian filtering was used to suppress high-frequency noise.
3. Contrast Enhancement: CLAHE was applied to enhance local contrast of images with uneven lighting.

4. Binarisation: OTSU thresholding was applied to convert images to binary format, separating handwritten text from the background.
5. Morphological Operations**:** Opening and closing operations were applied to eliminate small noise and bridge broken strokes.
6. Median Blur: A median blur filter with kernel size of 3 was used to remove salt-and-pepper type of noise.
7. Small Component Removal: Small connected components (noise) of less than 15 pixels were removed.
8. Margin Addition: A 10-pixel white margin was added to the left or right of the image to prevent text from touching image borders.
9. Height Resizing: Images were rescaled to have a fixed height of 32 pixels while maintaining the aspect ratio.

Following preprocessing, images were stored in uint8 format with pixel values between 0 and 255 and a dimension of (32, variable_width).

4.2.3 Dataset Splitting

The dataset was split into training, validation as well as test subsets after preprocessing. The splitting followed a domain-aware approach, where the test set consisted exclusively of 100 custom UTAR attendance slip images to evaluate real-world performance. The remaining 222 custom samples were combined with the IAM and Multi-digit MNIST datasets. Subsequently, this merged collection was divided into training and validation subsets at an 80:20 ratio, employing stratified sampling according to data source to preserve a representative distribution.

4.2.4 Data Augmentation, Padding, and Normalisation

To enhance model robustness against handwriting variability and scanning artefacts, a stochastic augmentation pipeline was applied to the dataset. The original images were retained, and each image was processed by the augmentation pipeline to produce an augmented version. This effectively doubled the size of the dataset. The augmentation pipeline involved eight transformations with calibrated probabilities and parameters to simulate real-world variations, as summarised in Table 4.4.

Table 4.4 Data augmentation techniques

Technique	Probability	Parameters	Purpose
Gaussian blur	30%	Kernel (3,3)	Simulate defocusing
Brightness adjustment	30%	Factor [0.7, 1.3]	Simulate uneven lighting
Contrast adjustment	30%	Alpha [0.8, 1.2], Beta [-20,20]	Simulate different ink/paper
Random rotation	30%	Angle ±3°	Correct alignment errors
Horizontal shift	30%	Shift ±10 pixels	Simulate off-centre cropping
Horizontal scaling	20%	Scale [1.1, 1.3]	Modify character spacing
Morphological operations	20%	Kernel (1,2)	Simulate stroke thickness
Random noise dots	10%	5-20 dots per image	Simulate scanning artefacts

After augmentation, images were padded to a fixed width of 640 pixels using white padding on the right or cropping from the left. Pixel values were divided by 255 to normalise them into the range [0, 1], and a channel dimension was introduced, yielding a shape of (32, 640, 1).

4.2.5 Data Export

After completing the data preparation pipeline, the processed data was exported and saved as one file serialised using the Python pickle format. The processed arrays of images, the associated text labels, source identifiers, a character mapper and configuration parameters were contained in this file. This enabled loading the dataset in an efficient manner without having to re-process the dataset.

4.3 Model Design

This project involved model design and development, from architecture building to baseline establishment, hyperparameter optimisation, retraining with optimised configuration, and final evaluation, as shown in Figure 4.5. This iterative workflow aimed to obtain the best optimal model through an efficient and agile development flow.

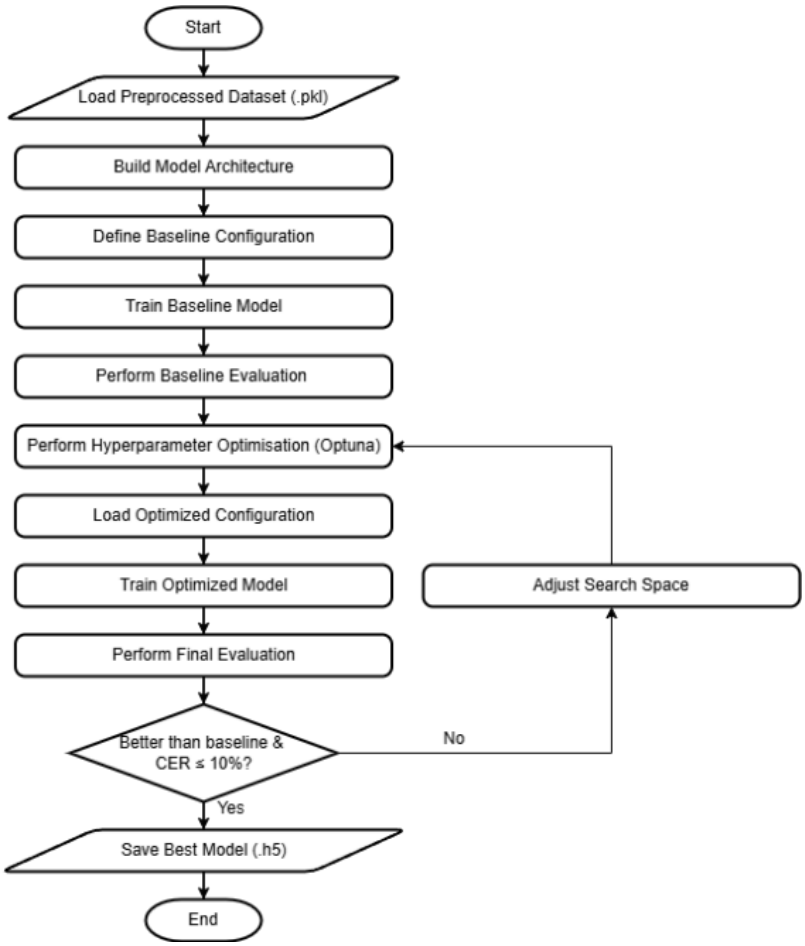

Figure 4.5 Model design and development flowchart

4.3.1 Baseline Model Architecture

To address handwriting variability and limited character scope, this project implemented a lightweight CNN-BiLSTM-CTC based end-to-end HTR system. Following [9], the model required no explicit character segmentation and handled cursive, irregularly spaced, or distorted handwriting. Trained on UTAR attendance slips and open-source datasets like IAM and Multi-digit MNIST, it extracted index numbers, seat numbers, and subject names from attendance slips. Figure 4.6 shows the baseline model architecture.

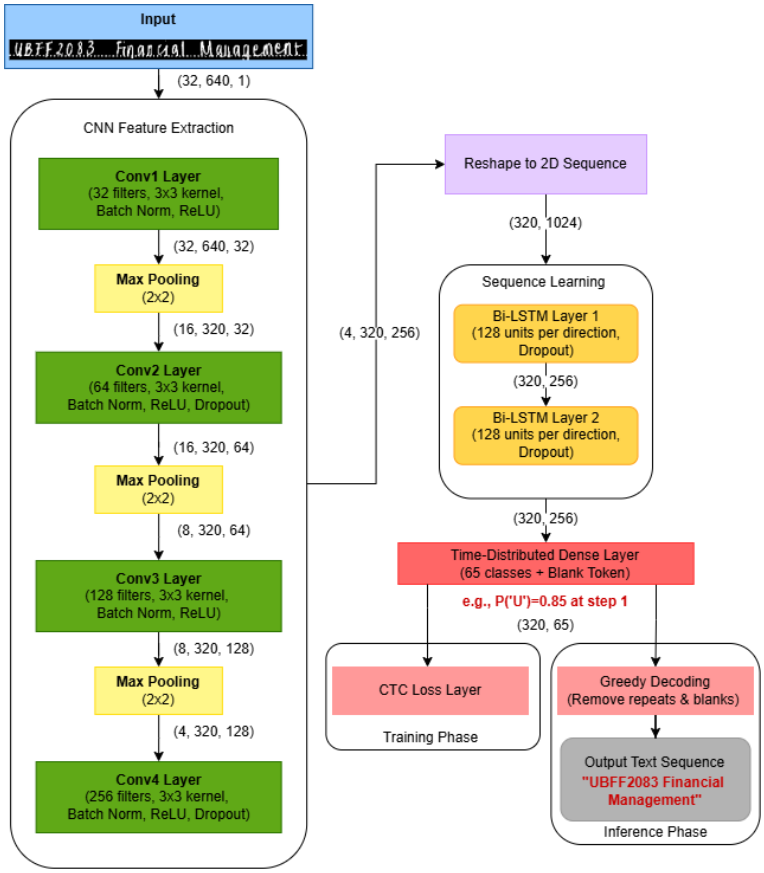

Figure 4.6 Baseline HTR model architecture

Input Layer

The model received a 32 × 640 greyscale image as input. For an example, the input was a cropped image of the handwritten subject name “UBFF2083 Financial Management” from a UTAR exam attendance slip. The input could include cursive, slanted or uneven handwriting after a preprocessing pipeline, which was the issue of handwriting variability described in Section 1.1.

Convolutional Neural Network (CNN)

Four convolutional layers (Conv1-Conv4) with 32, 64, 128 and 256 filters of 3×3 kernel size were applied to extract features from the image. These were responsible for detecting the stroke, gap and deformity patterns. Training was stabilised through batch normalisation, non-linearity was added through ReLU and spatial dimensions were reduced through max pooling. Conv2 and Conv4 were followed by a dropout to avoid overfitting. The output was then restructured into a 2D sequence for the RNN.

In the example “UBFF2083 Financial Management”, the CNN learned to distinguish visually similar characters such as “B” and “8” by observing stroke shape, direction, and spacing. The CNN also recognised that the vertical line in “F” differed from the curved stroke in lowercase “f” when present.

Recurrent Neural Network (RNN) with Bidirectional LSTM

The reshaped sequence of features was subsequently processed by two BiLSTM layers with 128 units in both forward and backward directions. Dropout was applied after each layer for regularisation.

The BiLSTM analysed the sequence in both directions, capturing past and future context. In “UBFF2083 Financial Management”, the slanted “e” in “Management” or the closely spaced “08” in “2083” could be difficult to recognise visually. However, the BiLSTM used surrounding characters: the “e” was predicted because it followed “Manag” and preceded “ment”, while “08” was recognised as two digits because of the numeric context of “2083”.

Connectionist Temporal Classification (CTC)

The final BiLSTM outputs were passed through TimeDistributed Dense layer with softmax activation, producing a probability distribution over 65-character classes, including mixed-case letters, digits, space, period and a CTC-required blank token. This structure allowed a variable character range, which was ideal for UTAR attendance characters with mixed-case letters and digits.

During training, the ctc_batch_cost loss function of TensorFlow was used to compute the difference between the predicted sequence and the ground truth label “UBFF2083 Financial Management”. The loss function allowed the model to discover how the input sequences were aligned to the labels without manually dividing sequences into characters.

During inference, the ctc_decode method with greedy decoding of TensorFlow was applied to determine the most likely characters at each time step. It eliminated repetition in characters and got rid of blank tokens. For instance, an output like: [U, U, blank, B, B, F, F, blank, 2, 2, 0, 8, 3, blank, F, i, n, a, n, c, i, a, l, blank, M, a, n, a, g, e, m, e, n, t] was decoded as “UBFF2083 Financial Management”, showing the model's capacity to deal with variations in

handwriting, including different spacing, repeated characters, and a combination of alphabet and numbers.

Model Summary

The baseline model summary is given in Figure 4.7. This model contained approximately 1.98 million trainable parameters. The input to the model had the shape (batch size, 32, 640, 1) where 32 and 640 are the height and width with one grayscale channel. The output shape was (batch size, 320, 65), with 320 being the number of the time steps after CNN downsampling and 65 being the number of character classes, 64 of which are printable characters and 1 CTC blank character.

Layer (type)	Output Shape	Param #
image_input (InputLayer)	[(None, 32, 640, 1)]	0
conv2d_12 (Conv2D)	(None, 32, 640, 32)	320
batch_normalization_12 (BatchNormalization)	(None, 32, 640, 32)	128
re_lu_12 (ReLU)	(None, 32, 640, 32)	0
max_pooling2d_9 (MaxPooling2D)	(None, 16, 320, 32)	0
conv2d_13 (Conv2D)	(None, 16, 320, 64)	18496
batch_normalization_13 (BatchNormalization)	(None, 16, 320, 64)	256
re_lu_13 (ReLU)	(None, 16, 320, 64)	0
max_pooling2d_10 (MaxPooling2D)	(None, 8, 320, 64)	0
dropout_12 (Dropout)	(None, 8, 320, 64)	0
conv2d_14 (Conv2D)	(None, 8, 320, 128)	73856
batch_normalization_14 (BatchNormalization)	(None, 8, 320, 128)	512
re_lu_14 (ReLU)	(None, 8, 320, 128)	0
max_pooling2d_11 (MaxPooling2D)	(None, 4, 320, 128)	0
conv2d_15 (Conv2D)	(None, 4, 320, 256)	295168
batch_normalization_15 (BatchNormalization)	(None, 4, 320, 256)	1024
re_lu_15 (ReLU)	(None, 4, 320, 256)	0
dropout_13 (Dropout)	(None, 4, 320, 256)	0
permute_3 (Permute)	(None, 320, 4, 256)	0
reshape_3 (Reshape)	(None, 320, 1024)	0
bidirectional_6 (Bidirectional)	(None, 320, 256)	1180672
dropout_14 (Dropout)	(None, 320, 256)	0
bidirectional_7 (Bidirectional)	(None, 320, 256)	394240
dropout_15 (Dropout)	(None, 320, 256)	0
char_probs (TimeDistributed)	(None, 320, 65)	16705

Total params: 1,981,377
Trainable params: 1,980,417
Non-trainable params: 960

Figure 4.7 Baseline model summary

4.3.2 Evaluation Metrics

Character Error Rate (CER)

The primary metric was CER, which is adopted from the jiwer Python library [19]. It is calculated as the number of character-level deletions, insertions, and substitutions needed to change the predicted text to the ground truth divided by the reference's character count. Less CER implies greater recognition accuracy.

$$\text{CER} = \frac{\text{Insertions + Deletions + Substitutions}}{\text{Total Characters in Ground Truth}} \times 100\% \qquad (4.1)$$

Word Error Rate (WER)

WER is an expansion of CER to the word level, computing the number of word-level changes like insertions, deletions, and substitutions required to make the prediction match the reference [19]. It is widely used in sequence recognition issues and complements CER by capturing semantic errors beyond character-level mistakes.

$$\text{WER} = \frac{\text{Insertions+Deletions+Substitutions}}{\text{Total Words in Ground Truth}} \times 100\% \qquad (4.2)$$

Exact Match Accuracy

This metric is the fraction of predictions that correspond to the ground truth sequences without errors. Even more serious than CER and WER, exact match accuracy provides the clue as to how many samples were correct, which is crucial in the context of high-stakes applications where a single wrong answer is unacceptable.

$$\text{Exact Match Accuracy} = \frac{\text{Number of Sequences with Exact Match}}{\text{Total Number of Sequences}} \times 100\% \qquad (4.3)$$

Confusion Matrix

A character-level confusion matrix was used to examine systematic model errors as well. It shows the most mistakenly classified characters, which includes trends like the mix of visually similar numbers (e.g., “1” and “7”) or letters (e.g., “O” and “0”). This diagnostic instrument helps to detect the areas of weaknesses in the model and possible paths to take in augmenting datasets or improving architecture.

4.3.3 Baseline Model Configuration

The baseline model was used to compare the performance of hyperparameter optimisation. The architecture followed the CNN-BiLSTM-CTC design described in Section 4.3.1 and used manually chosen hyperparameters. The entire setup of the baseline is summarised in Table 4.5.

Table 4.5 Baseline model hyperparameters

Parameter	Value	Description
CNN filters (block 1-4)	32, 64, 128, 256	Number of CNN filters
CNN kernel size	3 × 3	Size of CNN filter
CNN dropout	0.7	Dropout rate after CNN layers
LSTM units	128	Number of LSTM hidden units per direction
LSTM layers	2	Number of stacked BiLSTM layers
LSTM dropout	0.6	Dropout rate after LSTM layers
L2 regularisation	0.001	L2 regularisation strength
Optimiser	Adam	Optimiser type
Learning rate	0.001	Initial learning rate
Batch size	16	Batch size for training

The baseline model relied on relatively aggressive dropout (0.7 after CNN blocks, 0.6 after LSTM layers) and L2 regularisation (0.001) to prevent overfitting because the training set included a mix of custom and publicly available handwriting datasets. Adam optimiser was applied with its default learning rate of 0.001 in TensorFlow [20].

4.3.4 Baseline Model Training

Figure 4.8 illustrates the complete training pipeline for the baseline HTR model. The pipeline consisted of eight sequential stages, progressing from data loading to final model saving.

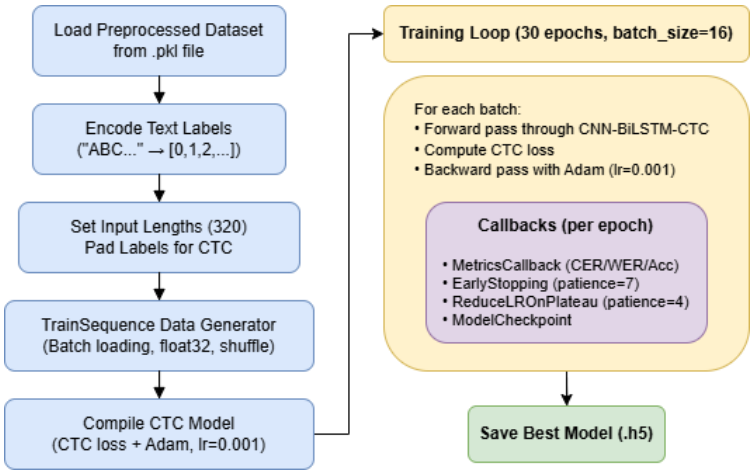

Figure 4.8 Baseline model training pipeline

The baseline model was trained using the configuration in Section 4.3.3. Before training, the preprocessed dataset was loaded from a pickle file, and text labels were encoded into integer sequences using a character mapper to facilitate processing of textual data numerically by the model. Input lengths were fixed at 320 time steps, and labels were padded to meet CTC requirements. Batches of images were then loaded dynamically by a custom generator TrainSequence, which cast the data to float32 type and shuffled the samples after each epoch.

The model was trained with 30 epochs and batches of 16. A CNN-BiLSTM-CTC network was applied to each batch to run a forward pass to generate character probability sequences, calculate CTC loss, and run a backward pass using Adam optimiser with learning rate 0.001 to update network weights.

At the end of each epoch, four callbacks monitored performance. MetricsCallback computed CER, WER, and Exact Match Accuracy. EarlyStopping halted training if validation CER did not improve for 7 epochs. ReduceLROnPlateau kept halving the learning rate after 4 epochs of plateau in validation CER. ModelCheckpoint stored the best weights in an .h5 file. The final model served as the baseline for comparison against the optimised model.

4.3.5 Hyperparameter Optimisation

Although the baseline model formed a reasonable starting point, its manually selected hyperparameters might not be optimal. To systematically optimise performance, hyperparameter optimisation was performed using Optuna, a Python-based framework that automated the search for optimal hyperparameters by using a Tree-structured Parzen Estimator (TPE) sampler to balance exploration and exploitation [17]. Compared to manual tuning, grid search, or random search, Optuna iteratively learns from past evaluations, making it more effective for complex models like CNN-BiLSTM-CTC.

In this project, the optimisation objective was set to minimise the validation CER. A total of 20 optimisation trials were conducted, with each trial training the model for 10 epochs to evaluate performance. Table 4.6 defines the complete search space for this project.

Table 4.6 Hyperparameter search space

Hyperparameter	Search Space
filters1, filters2	[32, 64]
filters3, filters4	[128, 256]
kernel_size1, kernel_size2	[3, 5, 7]
CNN dropout	[0.3, 0.7]
LSTM units	[128, 256]
LSTM layers	[2, 3, 4]
LSTM dropout	[0.3, 0.7]
L2 regularisation	[1e-5, 1e-2], (log scale)
Optimiser	[Adam, Nadam, RMSprop]
Learning rate	[1e-4, 1e-2], (log scale)
Batch size	[8, 16, 32]

4.4 API Design

Embedding the HTR model directly on mobile devices was impractical due to its computational intensity and GPU dependency. Therefore, a RESTful API was developed to provide remote access, acting as a bridge between the mobile application and the model, allowing students to upload slips and receive results without local installation. Figure 4.9 presents the seven-step processing pipeline of the API, which was implemented in Python using EasyOCR for dynamic text detection, OpenCV for image preprocessing, TensorFlow for model inference, and Uvicorn as the ASGI server.

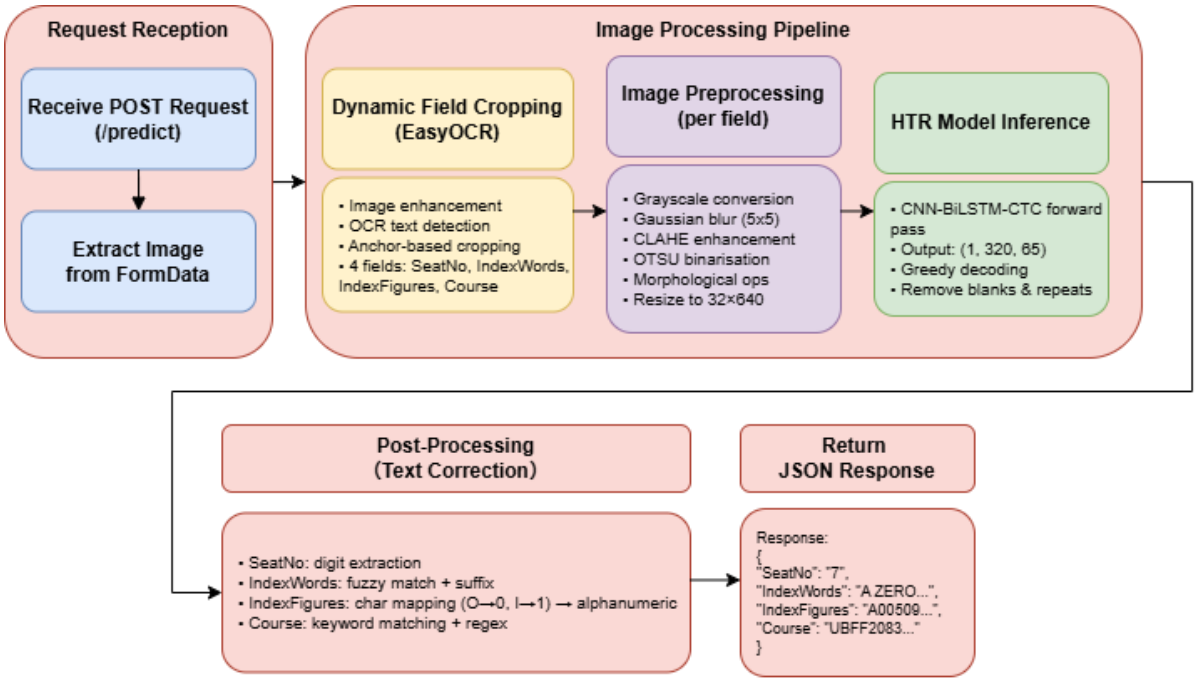

Figure 4.9 API processing pipeline

4.4.1 API Framework and Endpoint

In this project, FastAPI served as the Python web framework for API development. It offered automatic interactive documentation and native data validation, which reduced unnecessary inference calls from malformed inputs, preserving free-tier computational resources during exam sessions. The API exposed a single "/predict" endpoint that accepts POST requests with multipart form data containing an attendance slip image in JPEG, JPG or PNG format.

4.4.2 Image Processing Pipeline

Upon receiving a request, the API executed a three-stage pipeline for handwritten text extraction.

1. Dynamic Field Cropping: EasyOCR identified text blocks, found printed label anchors of each target field and cut handwritten areas based on spatial proximity.
2. Image Preprocessing: Each cropped field was converted to grayscale, blurred, enhanced using CLAHE, binarised using OTSU, smoothed with morphological operations and resized to 32×640 pixels with normalised values.
3. Model Inference and Decoding: The processed image was fed to the CNN-BiLSTM-CTC model, resulting in the character probability sequences of size (1, 320, 65) which were greedily decoded to remove the repeated characters and blank tokens to produce the final recognised text string.

4.4.3 Post-processing and Response Format

After HTR inference, the recognised text underwent field-specific post-processing to correct common recognition errors before being returned to the student, reducing the need for manual modification during confirmation.

1. SeatNo: Extracted numeric values from the recognised text.
2. IndexWords: Corrected misspelled number words using fuzzy matching and formatted the output with the letter "A" followed by five number words and a suffix such as "BBFNF" or "CBCSF", for example, "A ZERO ZERO ZERO ZERO ONE BBFNF".
3. IndexFigures: Applied character mapping to resolve ambiguous characters that are commonly misread, such as "O" to "0" and "I" to "1", and formatted the result as a compact alphanumeric string, for example, "A00001BBFNF".

4. Course: Matched the recognised text against a configurable list of courses using keyword matching and regular expressions to return the standard course code and full name, for example, “UBFF2083 FINANCIAL MANAGEMENT”.

Upon successful processing, the API returned a JSON response containing the four extracted fields, as shown in Figure 4.10.

```
json
{
“SeatNo”: “7”,
“IndexWords”: “A ZERO ZERO FIVE ZERO NINE BBFNF”,
“IndexFigures”: “A00509BBFNF”,
“Course”: “UBFF2083 FINANCIAL MANAGEMENT”
}
```

Figure 4.10 Sample API JSON response

4.4.4 Error Handling

As shown in Table 4.7, the API implemented comprehensive error handling to ensure robustness.

Table 4.7 API error handling scenarios

Error Scenario	HTTP Status	Response
No text detected in crop	200	Empty field values (e.g., {“SeatNo”: “”, ...})
Missing file in request	422	{“detail”: “Field required”}
Invalid image format	400	{“detail”: "Invalid image format: ...”}

4.4.5 Deployment

The API was deployed on Hugging Face Spaces, a cloud platform natively compatible with TensorFlow models that offered free hosting and automatically set up the environment using a requirements.txt file, eliminating manual server configuration. The deployment bundle included the FastAPI application code, the trained CNN-BiLSTM-CTC model weights, the Python dependency list, and all preprocessing and post-processing modules. The Hugging Face Space ran the FastAPI application with Uvicorn as the ASGI server.

4.5 Database Design

This project employed a hybrid database architecture with cloud and local storage to provide data availability, offline support and real-time data synchronisation, involving the local database SharedPreferences on the mobile phone and the cloud database Firebase Firestore. The architecture, as shown in Figure 4.11, was subdivided into four parts, which were mobile app, web management system, local database and cloud database.

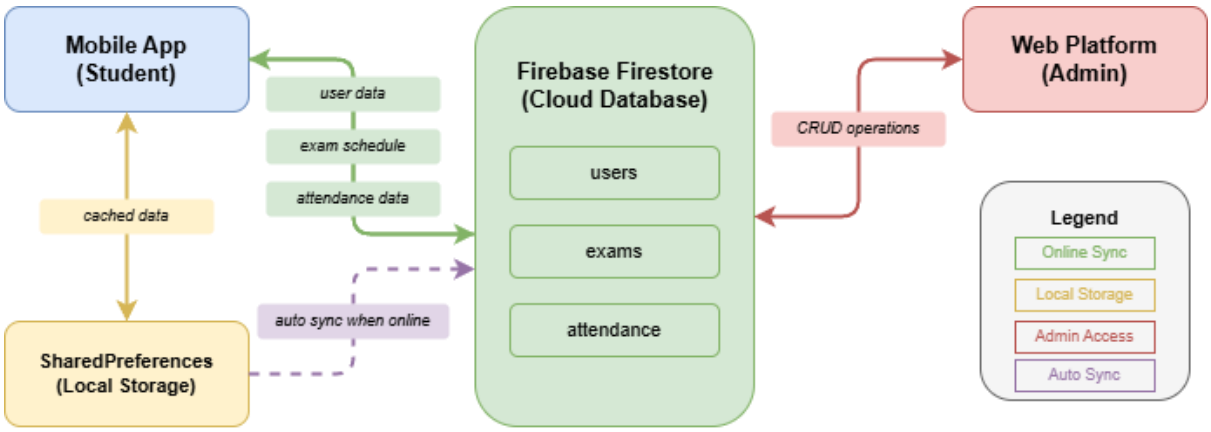

Figure 4.11 Database architecture

4.5.1 Cloud Database: Firebase Firestore

Firebase Firestore is a NoSQL document-based database offering a real-time synchronisation and scalable data storage [21]. Firestore was chosen based on this system since it did not require manual setup and maintenance of a backend server. It also did not need to predefine a schema, unlike traditional relational databases like MySQL or PostgreSQL, which was especially beneficial to this system since the structure of the attendance records might change over time. Moreover, Firestore could be easily combined with Firebase Authentication, allowing the system to enforce security rules based on user identity and role without writing custom backend code [21].

The database was structured into three major collections: users, exams, and attendance, as presented in the subsequent subsections:

1. Users Collection

The “users” collection stored all user accounts, using the student ID as the document ID for students to enable quick access without additional queries, while administrators were assigned a generated ID. As listed in Table 4.8, the “role” field enabled role-based access control, and the “indexNo” field stored the student’s registered index number for validating scanned attendance slips.

Table 4.8 Users collection fields

Field Name	Data Type	Description	Example
email	String	User’s email address	“lily@1utar.my”
fullName	String	User’s full name	“Lily”
role	String	User role (student or admin)	“student”
studentId	String	Student ID (for students only)	“23ACB00001”
indexNo	String	Registered index number	“A000001BBFNF”
faculty	String	Faculty name	“FICT”
programme	String	Enrolled programme	“Bachelor of Computer Science”
avatarStyle	String	Avatar style preference	“avataaars”
createdAt	String	Account creation timestamp	“2026-03-01T12:00:00.000000”
updatedAt	String	Last update timestamp	“2026-03-02T14:05:00.000000”
lastLogin	String	Last login timestamp	“2026-03-02T14:00:00.000000”

2. Exams Collection

The “exams” collection stored examination session information, with document IDs generated from the subject name in a slugified format (e.g., “financial-management”) to prevent duplicates and enable retrieval by subject name without additional queries. As listed in Table 4.9, the “allowedClasses” array stored enrolled student IDs for quick authorisation checks, and the “isActive” field allowed administrators to open or close attendance submission windows without deleting exam records.

Table 4.9 Exams collection fields

Field Name	Data Type	Description	Example
subjectName	String	Full subject name	“Financial Management”
location	String	Exam venue	“Block N, Room N009”
startTime	String	Exam start time	“2026-03-01T14:00:00.000000”
endTime	String	Exam end time	“2026-03-01T16:00:00.000000”
allowedClasses	Array	List of student IDs allowed	[“23ACB00001”, “23ACB00002”]
isActive	Boolean	Whether exam is currently active	true
createdAt	String	Creation timestamp	“2026-02-25T12:00:00.000000”
updatedAt	String	Last update timestamp	“2026-02-25T12:05:00.000000”

3. Attendance Collection

The “attendance” collection recorded each student's attendance submission, with the document ID constructed as “{examId}_{studentId}” to guarantee uniqueness and enable quick retrieval without complex filtering. As listed in Table 4.10, location fields captured from the device’s GPS helped prevent fraudulent submissions from outside the campus. The “autoCreated” flag indicated an automatically generated absent record, while the “isAcknowledged” flag showed whether an administrator had manually verified the attendance record to maintain accuracy and accountability.

Table 4.10 Attendance collection fields

Field Name	Data Type	Description	Example
attendanceId	String	Unique identifier	“ubff2083-financial-management_23ACB00001”
examId	String	Reference to exam document	“ubff2083-financial-management”
studentId	String	Reference to student	“23ACB00001”
indexNo	String	Extracted index number (compact)	“A000001BBFNF”
indexNoWords	String	Extracted index number (words)	“A Zero Zero Zero Zero One B B F N F”
seatNo	String	Extracted seat number	“12”
scannedAt	String	Timestamp of submission	“2026-03-01T14:00:00.000000”
status	String	Attendance status (present or absent)	“present”
scannedLocation	String	Human-readable location	“UTAR Kampar Campus”
scannedLatitude	Double	Latitude of scan location	4.2105
scannedLongitude	Double	Longitude of scan location	101.0892
autoCreated	Boolean	Whether record was auto-created for absent	false
createdAt	String	Record creation timestamp	“2026-03-01T14:00:00.000000”
updatedAt	String	Last update timestamp	“2026-03-01T14:00:00.000000”
isAcknowledged	Boolean	Whether the attendance has been verified by admin	true
acknowledgedAt	String	Timestamp when admin acknowledged the record	"2026-03-01T15:00:00.000000"
acknowledgedBy	String	Admin email who acknowledged the record	"admin@utar.edu.my"

4.5.2 Local Database: SharedPreferences

SharedPreferences is a small-scale key-value persistence of small data volumes on the device. This system was chosen because it had native Flutter support and no further configuration was required. To reduce server load and make loading faster, the mechanism cached the often-used data like login sessions, exam schedules, profile data, and attendance logs, as well as ensured a sync queue was used to store the operations that were awaiting execution during offline time which could be automatically uploaded once connectivity was restored. The storage keys used in the application are listed in Table 4.11.

Table 4.11 SharedPreferences storage keys

Key	Data Type	Description
is_logged_in	Boolean	User login status
user_data	JSON String	Cached user information
user_profile	JSON String	Cached student profile
cached_exams	JSON String	Cached exam schedule
exam_records	JSON String	Cached attendance records
last_sync_time	String	Timestamp of last successful sync
sync_queue	List<String>	Queue of pending offline operations

4.5.3 Data Synchronisation and Offline Queue

The synchronisation between local and cloud databases followed the workflow illustrated in Figure 4.12.

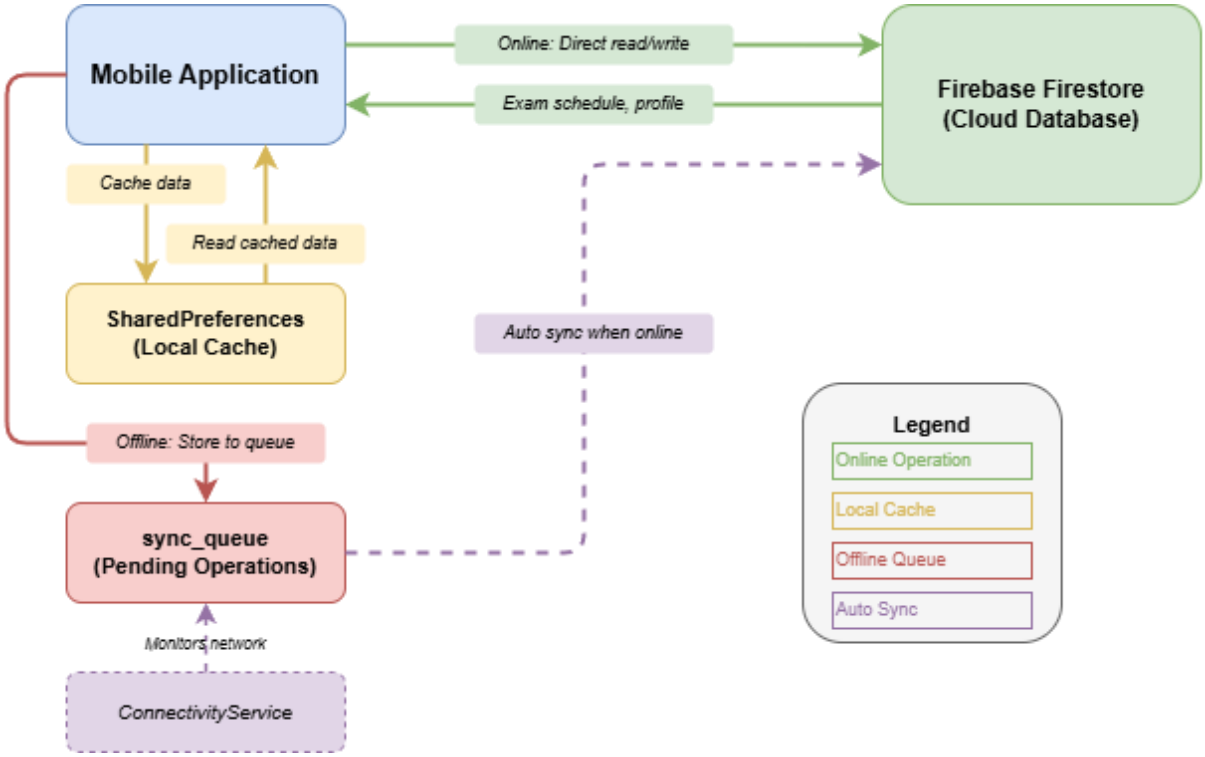

Figure 4.12 Data synchronisation workflow

With an active internet connection, the application would read from and write directly to Firestore and attendance submissions were immediately stored in the cloud. When the device was offline, the application continued to function by reading from cached data stored in SharedPreferences. User actions such as profile updates were serialised and stored in the sync_queue rather than being lost. Table 4.12 defines the structure of each queue item.

Table 4.12 Sync queue item structure

Field	Type	Description	Example
type	String	Operation type	"profile"
studentId	String	Student identifier	"23ACB00001"
data	Object	Operation payload	{"fullName": "Lily"}

Once the network was reconnected, the ConnectivityService notified the network change and called the SyncManager which executed the pending operations on the sync queue. The operations successful in the queue were pulled out of the queue and the failed operations were stored so they could be retried during the next synchronisation cycle. For read-only data, cached information was considered valid for one hour, after which fresh data was fetched from Firestore.

4.5.4 Security Rules

Firestore Security Rules were used to limit access to data depending on user authentication and role. The access control policies were summarised in Table 4.13.

Table 4.13 Firestore security rules

Collection	Read Access	Write Access
“users”	Student (own record only); Administrator (all)	Student (own profile only)
“exams”	Student and Administrator (all)	Administrator only
“attendance”	Student (own records only); Administrator (all)	Student (create own records only); Administrator (update status and acknowledgement)

These rules ensured that students could only view and submit their own attendance records and update their own profile. The administrators could have entire read access to the users, exams, and attendance history. They could create, update and delete exams, and manually mark the attendance of students in case of any problems with the system, and verify that records are accurate by setting the acknowledgement flag. Firebase Auth performed authentication, and the security rules confirmed the user’s identity and role on each database request.

4.6 Mobile Application and Web Platform Design

This section presents the UI wireframes of the web platform and mobile application. Both interfaces were developed using Flutter, a cross-platform system that allowed the use of one codebase to be compiled on Android and web browsers. Flutter was selected for this project because it offered a rich set of camera and image-handling plugins for capturing attendance slips; included built-in navigation components for managing multiple screens such as scan, records, and profile; and supported rapid UI prototyping through hot reload, a feature that accelerated the overall development process.

4.6.1 UI Wireframes for Mobile Application

Figure 4.13 illustrates the login, sign-up and home screens, where students can log in, sign up, or view their list of upcoming exams and attendance list after login. Figure 4.14 displays the scan, camera, loading, confirmation and success screens, where students can scan attendance slips, wait for results from HTR API, preview the extracted data, and submit the attendance. Figure 4.15 shows the attendance records and details screens, where students can see lists of attendance records. Figure 4.16 presents the profile screen, allowing students to view and update their profiles.

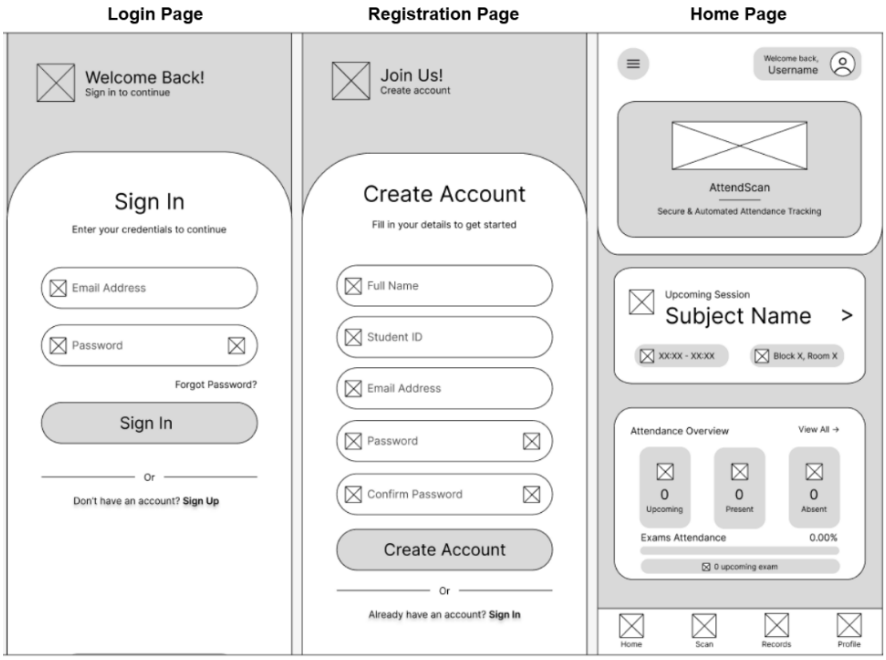

Figure 4.13 Student login, registration, and home screens

Figure 4.14 Scanning workflow screens

Figure 4.15 Attendance records and details screens

Figure 4.16 Profile screen

4.6.2 UI Wireframes for Web Platform

Figure 4.17 represents the screens of login and register where administrators can log in or register. Figure 4.18 depicts the dashboard display comprising an overview of the exams and action buttons. The exam management screen is shown in Figure 4.19 and it permits an administrator to view, search, filter and sort exams and to create new exams. Figure 4.20 depicts the create exam dialogue box which enables administrators to enter the subject, location, date and time and student enrolment information. The student management screen, as illustrated by Figure 4.21, allows administrators to search, filter, sort and view student profiles. Figure 4.22 depicts the attendance management screen where the administrators can select an exam, filter records, batch acknowledge and export reports as CSV files.

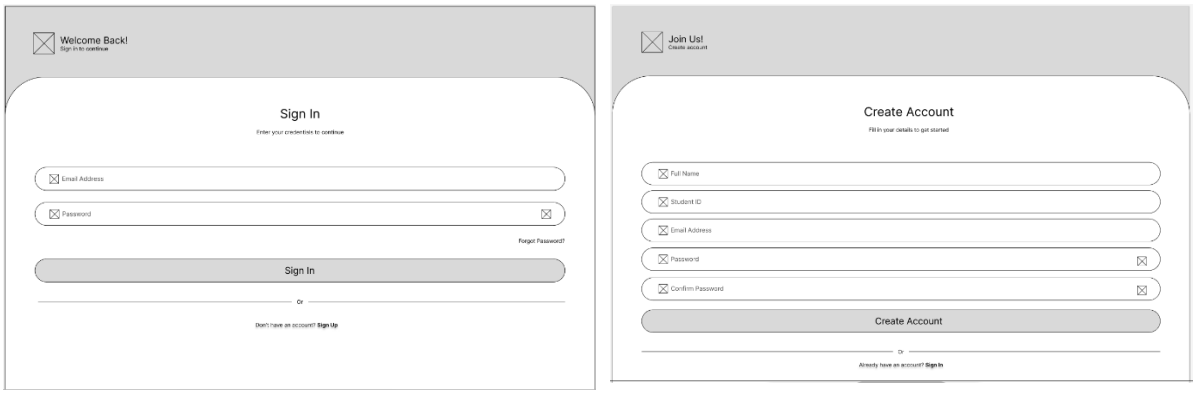

Figure 4.17 Admin login and registration screens

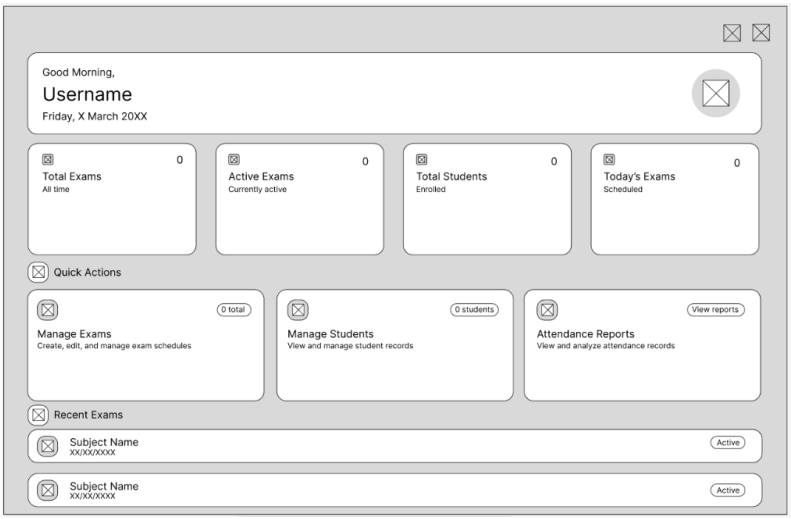

Figure 4.18 Dashboard screen

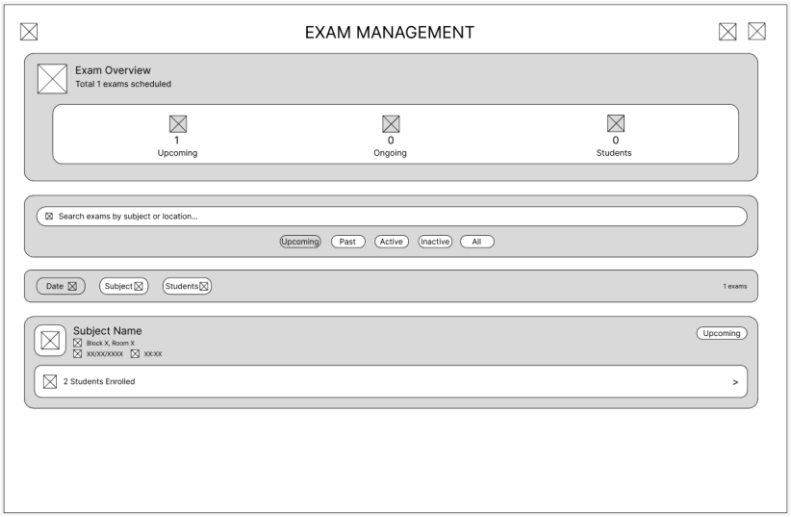

Figure 4.19 Exam management screen

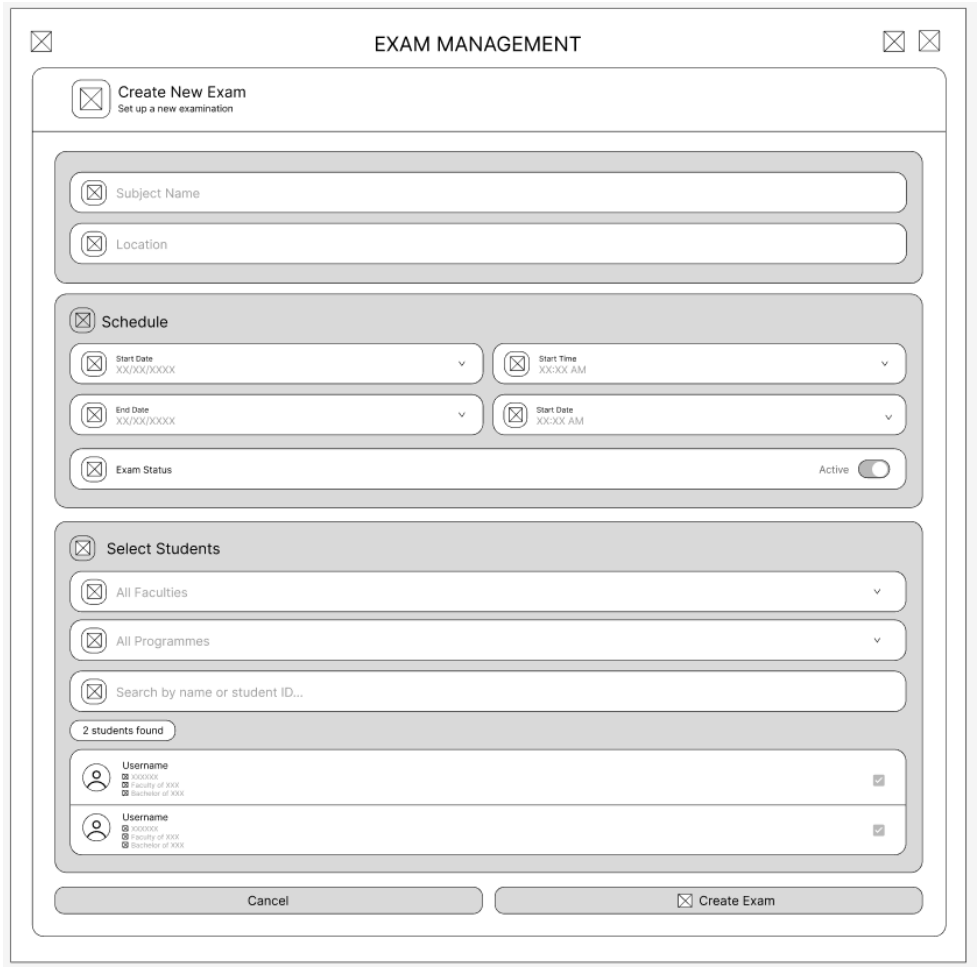

Figure 4.20 Create exam screen

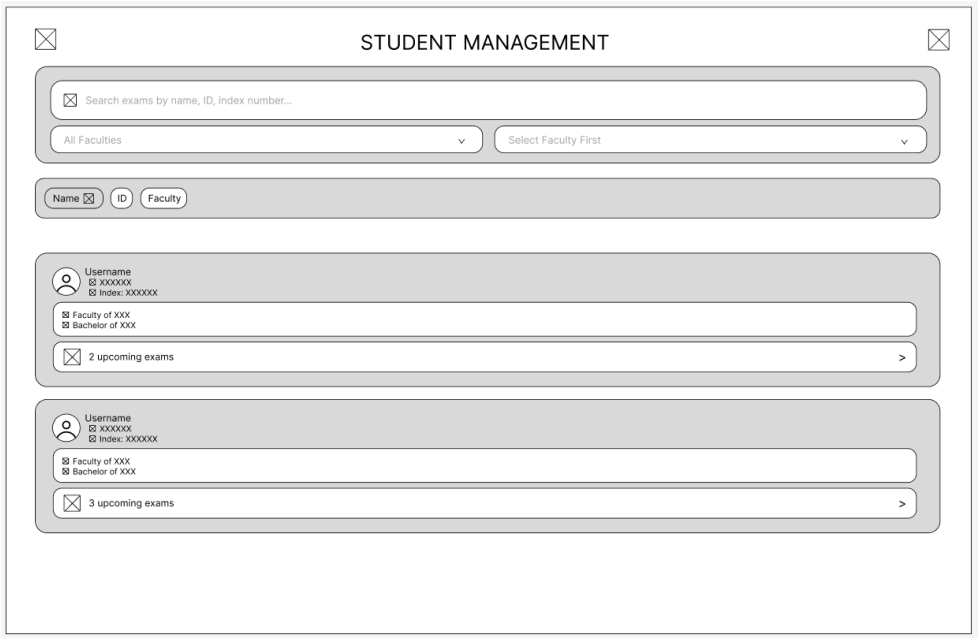

Figure 4.21 Student management screen

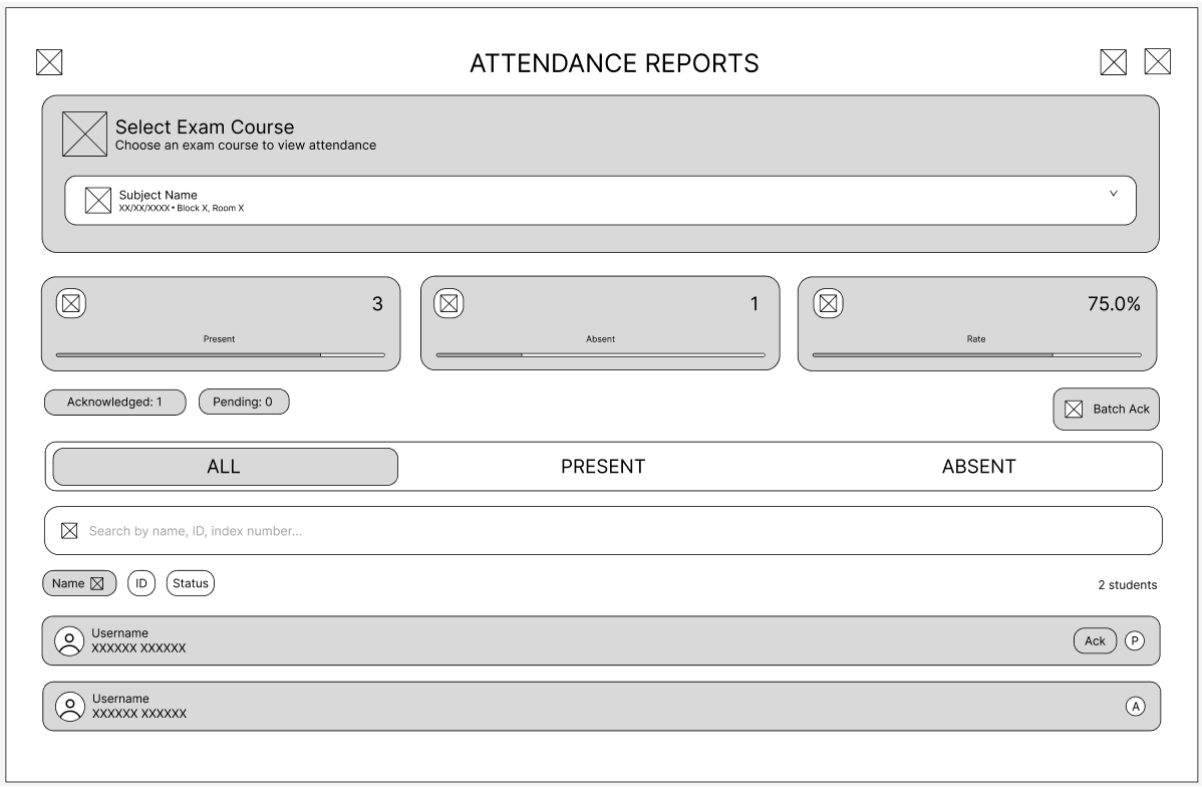

Figure 4.22 Attendance management screen

Chapter 5
System Implementation

5.1 Data Preparation Implementation

This section presents the implementation of the data preparation pipeline using Python, OpenCV, NumPy, and Pandas for data preprocessing, splitting, augmentation, padding and normalisation, and export.

5.1.1 Data Loading

Table 5.1 shows the number of samples loaded from the custom, IAM, and Multi-digit MNIST datasets. Such hybrid data dramatically enlarges the recognised character types and increases variability of handwriting, which generalises to real-world exam attendance slips.

Table 5.1 Dataset loading results

Dataset Source	Samples Loaded
Custom (UTAR slips)	322
IAM	5,389
Multi-digit MNIST	3,000
Total	8,711

5.1.2 Data Preprocessing

After loading the datasets, each image from the custom attendance slips and IAM datasets was preprocessed using the OpenCV and NumPy pipeline shown in Figure 5.1. Section 4.2.2 outlines the detailed steps and the entire code is included in Appendix A.1 Data Preprocessing Pipeline. Sample images before and after processing are illustrated in Figure 5.2 and Figure 5.3.

```
def preprocess(self, img_path):    # Full implementation in Appendix A.1
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    img = cv2.GaussianBlur(img, (3, 3), 0)
    img = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8)).apply(img)
    _, binary = cv2.threshold(img, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
    binary = cv2.morphologyEx(binary, cv2.MORPH_OPEN, np.ones((2,2)))
    binary = cv2.morphologyEx(binary, cv2.MORPH_CLOSE, np.ones((2,2)))
    binary = self._remove_small_components(binary, min_area=15)
    binary = self.add_margin(binary)
    return binary
```

Figure 5.1 Code snippet for image preprocessing

Figure 5.2 Custom dataset sample before and after preprocessing

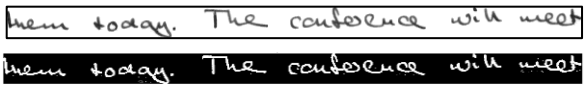

Figure 5.3 IAM sample before and after preprocessing

Unlike the custom and IAM images, the Multi-digit MNIST dataset already came with clean, standardised digits with consistent contrast and minimal noise. Therefore, extensive preprocessing including CLAHE, OTSU and morphological operations was unnecessary. Only margin padding and 32-pixel height resizing were applied, as shown in Figure 5.4. Sample images before and after processing are illustrated in Figure 5.5.

```
def preprocess_multi_digit(self, img_path):
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

    img = self.add_margin(img)
    h, w = img.shape
    new_w = max(1, int(w * (32 / h)))
    return cv2.resize(img, (new_w, 32))
```

Figure 5.4 Code snippet for Multi-digit MNIST preprocessing

Figure 5.5 Multi-digit MNIST sample before and after preprocessing.

5.1.3 Data Splitting

To guarantee the domain relevance, 100 custom images were set aside as an independent test set. The remaining 222 custom samples were combined with 5,389 IAM and 3,000 Multi-digit MNIST images to create 8,611 images for training and validation. As presented in Figure 5.6, the train_test_split() function from scikit-learn with stratification was then used to split the dataset into an 80:20 ratio, ensuring proportional distribution of all three datasets between the training and validation sets. This resulted in 6,888 training images and 1,723 validation images, with the independent test set containing 100 custom images, as illustrated in Figure 5.7.

```
from sklearn.model_selection import train_test_split
X_train_val = X_iam + X_multi + X_custom_trainval
sources = ['iam']*len(X_iam) + ['multi']*len(X_multi) + ['custom']*len(X_custom_trainval)

X_train, X_val, y_train, y_val = train_test_split(
    X_train_val, y_train_val, test_size=0.2, random_state=42, stratify=sources
)
```

Figure 5.6 Code snippet for data splitting

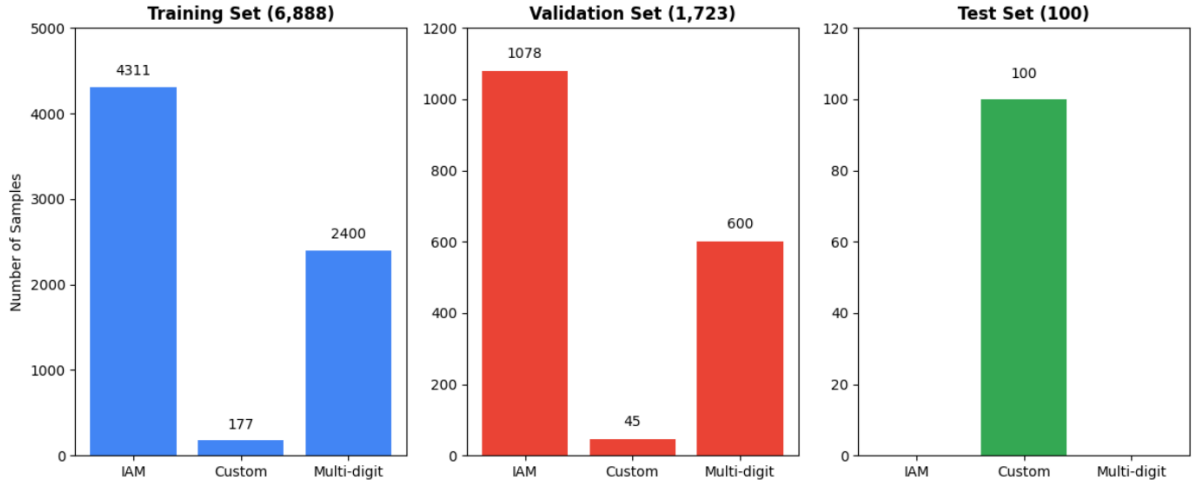

Figure 5.7 Dataset distribution across splits

5.1.4 Data Augmentation, Padding, and Normalisation

To enhance model robustness against handwriting variability, the random_augment() method applied eight stochastic transformations to each image, as detailed in Section 4.2.4. Figure 5.8 shows an example of the random rotation augmentation, while the complete implementation of all eight augmentations is provided in Appendix A.2 Data Augmentation. Figure 5.9 shows representative examples of the visual effects of these augmentations.

```
def random_augment(self, image):
    # Example: random rotation (±3°, 30% probability)
    if random.random() < 0.3:
        M = cv2.getRotationMatrix2D((w//2, h//2), random.uniform(-3,3), 1)
        image = cv2.warpAffine(image, M, (w,h), borderValue=255)

    # Full implementation with all 8 augmentations in Appendix A.2
    return image
```

Figure 5.8 Code snippet for random augmentation

Figure 5.9 Examples of original and augmented images

These transformations increased dataset diversity without compromising semantics. The original and augmented versions were retained, which effectively doubled the dataset size of 8,711 to 17,422 samples. The final composition consisted of 13,776 training, 3,446 validation and 200 test images as illustrated in Figure 5.10.

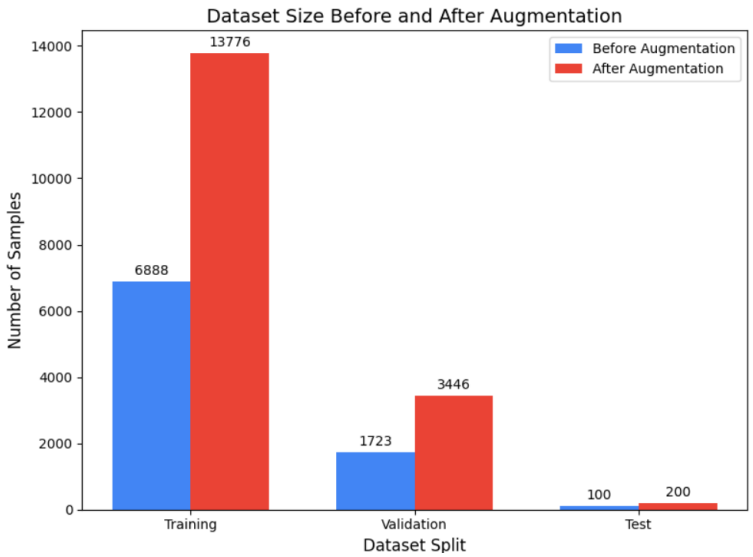

Figure 5.10 Dataset size before and after augmentation

Following augmentation, all the images were padded or cropped to a width of 640 pixels with the pad_image() method, as the model requires the same input size for batch learning. Images with a width less than 640 pixels were padded with white pixels on the right, and images with a width greater than 640 pixels were cropped on the left. The normalise_image() method then normalised pixel values to [0, 1] by dividing by 255 using NumPy, and a channel dimension was added to produce a shape of (32, 640, 1). This padding and normalisation are demonstrated in Figure 5.11.

```
def pad_image(self, image, target_width=640):
    h, w = image.shape
    if w < target_width:
        padded = np.pad(image, ((0,0), (0, target_width-w)), constant_values=255)
    else:
        padded = image[:, :target_width]
```

```
    return padded

def normalise_image(self, image):
    normalised = image.astype(np.float32) / 255.0
    return np.expand_dims(normalised, axis=-1)
```

Figure 5.11 Code snippet for padding and normalisation

5.1.5 Data Export

Lastly, the dataset was saved as .pkl file using Python's pickle module, as shown in Figure 5.12. The file stored the training, validation and test data (X train, X val and X test), the training, validation and test labels (y train, y val and y test), the source identifiers, the character mapper for encoding and decoding the text, as well as the configuration parameters.

```
def save_dataset(dataset_dict: Dict, save_path: str):
    os.makedirs(os.path.dirname(save_path) if os.path.dirname(save_path) else '.', exist_ok=True)
    with open(save_path, 'wb') as f:
        pickle.dump(dataset_dict, f)
```

Figure 5.12 Code snippet for saving the dataset

5.1.6 Final Composition

The final data composition after the data preparation is shown in Table 5.2. This compromise allowed for generalisation from large-scale public data, while offering realistic assessment on domain-specific data.

Table 5.2 Final dataset composition

Dataset Source	Train	Val	Test	Total
Custom (UTAR slips)	354	90	200	**644**
IAM (Handwriting)	8,622	2,156	0	**10,778**
Multi-digit MNIST	4,800	1,200	0	**6,000**
Total	**13,776**	**3,446**	**200**	**17,422**
Shape	**(13776, 32, 640, 1)**	**(3446, 32, 640, 1)**	**(200, 32, 640, 1)**	

5.2 Model Implementation

This section presents the implementation of the CNN-BiLSTM-CTC HTR model using TensorFlow 2.8 and Keras, including model architecture construction, training setup, baseline training, hyperparameter optimisation, retraining, and final training performance.

5.2.1 Model Architecture and Training Setup

This model was constructed with the Keras Functional API of TensorFlow, as illustrated in Figure 5.13. The CNN feature extractor consisted of Conv2D layers with 3x3 kernel and ReLU activation, followed by BatchNormalization and MaxPooling2D, and dropout was used as a regularisation measure. The feature maps were then reshaped and fed into two Bidirectional LSTM layers with 128 units each, and lastly a TimeDistributed Dense layer with softmax generated probability distributions of characters. Appendix A.3 Model Architecture gives the full architecture with four convolutional blocks.

```
def _build_base_model(self):
    inp = Input(shape=self.input_shape)
    # CNN feature extraction (4 blocks) + Reshape + BiLSTM (see Appendix A.3)
    x = Conv2D(32, (3,3), padding='same')(inp)
    x = Reshape((x.shape[1], x.shape[2] * x.shape[3]))(x)
    x = Bidirectional(LSTM(128, return_sequences=True))(x)
    out = TimeDistributed(Dense(self.num_classes, activation='softmax'))(x)
    return Model(inp, out)
```

Figure 5.13 Code snippet for building the base CNN-BiLSTM-CTC model

For training, the preprocessed data was loaded from a pickle file and labels were converted to integer sequences with a custom character mapper, shown in Figure 5.14. The input length was fixed at a constant 320 time steps with NumPy's np.ones, the same as the output sequence length. Label sequences were then padded to equal length with the blank label index for batch training using NumPy's np.pad.

```
def prepare_training_data(data, char_mapper, blank_label, time_steps=320):
    X_train, X_val = data['X_train'].astype(np.float32), data['X_val'].astype(np.float32)
    y_train = [char_mapper.encode_label(l) for l in data['y_train']]
    y_val = [char_mapper.encode_label(l) for l in data['y_val']]

    input_len = np.ones((len(X_train),1), dtype=np.int32) * time_steps
    val_input_len = np.ones((len(X_val),1), dtype=np.int32) * time_steps

    label_len = np.array([[len(l)] for l in y_train], dtype=np.int32)
    val_label_len = np.array([[len(l)] for l in y_val], dtype=np.int32)

    max_len = max(len(l) for l in y_train + y_val)
    pad = lambda seq: np.pad(seq, (0, max_len - len(seq)), constant_values=blank_label)

    return (X_train, np.array([pad(l) for l in y_train]), input_len, label_len,
            X_val, np.array([pad(l) for l in y_val]), val_input_len, val_label_len)
```

Figure 5.14 Code snippet for CTC training data preparation

To efficiently feed data into the model during training, a custom TrainSequence class inherited from Keras' Sequence class was implemented as a data generator, as shown in Figure 5.15. Image data was translated to float32 to allow it to be used by a GPU. NumPy's np.random.shuffle randomised sample order after each epoch.

```
class TrainSequence(Sequence):
    def __init__(self, X, y, in_len, lab_len, batch_size):
        self.X = X.astype(np.float32)
        self.y, self.in_len, self.lab_len = y, in_len, lab_len
        self.batch_size = batch_size
        self.indices = np.arange(len(X))

    def __len__(self):
        return int(np.ceil(len(self.X) / self.batch_size))

    def __getitem__(self, idx):
        idxs = self.indices[idx*self.batch_size:(idx+1)*self.batch_size]
        return ([self.X[idxs], self.y[idxs], self.in_len[idxs], self.lab_len[idxs]],
                np.zeros(len(idxs)))

    def on_epoch_end(self):
        np.random.shuffle(self.indices)
```

Figure 5.15 Code snippet for custom TrainSequence

Four callbacks were implemented to monitor and control the training process, as illustrated in Figure 5.16. EarlyStopping monitored validation CER with a patience of 7 epochs while restoring the best model weights. Meanwhile, ReduceLROnPlateau cut the learning rate by half whenever the validation CER plateaued for 4 epochs, down to a minimum of 1e-6. ModelCheckpoint stored the optimal weights according to the minimum validation CER.

```
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau, ModelCheckpoint
callbacks = [
    MetricsCallback(base_model, X_val, y_val_padded, val_label_lengths, char_list, blank_label, history),
    EarlyStopping(monitor='val_cer', mode='min', patience=7, restore_best_weights=True),
    ReduceLROnPlateau(monitor='val_cer', mode='min', factor=0.5, patience=4, min_lr=1e-6),
    ModelCheckpoint('best_model.h5', monitor='val_cer', mode='min', save_best_only=True)
]
```

Figure 5.16 Code snippet for training callbacks

Another custom MetricsCallback was also created to give more information about the model performance as demonstrated in Figure 5.17. This callback used TensorFlow's ctc_decode for greedy decoding and the jiwer library to compute CER and WER. It further calculated Exact Match Accuracy through a comparison that compared the predicted outputs with the actual ground truth labels. These were calculated on both the training and validation sets after each training epoch to enable the learning progress to be monitored in real-time.

```
class MetricsCallback(Callback):
    def on_epoch_end(self, epoch, logs=None):
        # Decode predictions using CTC greedy decoding
        pred_texts = self._decode(self._base_model.predict(self.X_val))
        true_texts = self._labels_to_text(self.y_val, self.val_lengths)

        # Compute metrics using jiwer
        logs['val_cer'] = cer(true_texts, pred_texts)
        logs['val_wer'] = wer(true_texts, pred_texts)
        logs['val_acc'] = sum(p == t for p, t in zip(pred_texts, true_texts)) / len(true_texts)
```

Figure 5.17 Code snippet for custom MetricsCallback

5.2.2 Baseline Model Training

The baseline model was built with the configuration, as detailed in Section 4.3.3. CTC loss was implemented by tf.keras.backend.ctc_batch_cost. The model was then trained using the fit() method with the custom TrainSequence generator for 30 epochs at a batch size of 16, as shown in Figure 5.18.

```
# Implement CTC Loss
ctc_loss = tf.keras.backend.ctc_batch_cost(
    labels, base_model.output, input_length, label_length
)

# Create CTC model
ctc_model = tf.keras.Model(
    inputs=[base_model.input, labels, input_length, label_length],
    outputs=ctc_loss
)

# Compile the model with Adam optimiser (Learning rate = 0.001)
ctc_model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    loss=lambda y_true, y_pred: y_pred
)

# Train the model using fit() with custom TrainSequence generator
history = ctc_model.fit(
    train_generator,
    epochs=30,
    batch_size=16,
    validation_data=val_generator,
    callbacks=callbacks
)
```

Figure 5.18 Code snippet for baseline model training

5.2.3 Hyperparameter Optimisation

Optuna was used to do hyperparameter optimisation over 20 trials, as illustrated in Figure 5.19. The search space was defined using the suggest_categorical, suggest_float and suggest_int

methods, based on the search space in Section 4.3.5. The TPE sampler balanced exploration and exploitation during the optimisation process. The complete optimisation code is provided in Appendix A.4 Hyperparameter Optimisation.

```
def objective(self, trial):
    learning_rate = trial.suggest_float('learning_rate', 1e-5, 1e-2, log=True)
    batch_size = trial.suggest_categorical('batch_size', [8, 16])
    optimizer_name = trial.suggest_categorical('optimizer', ['adam', 'nadam'])
    dropout_cnn = trial.suggest_float('dropout_cnn', 0.3, 0.7)
    lstm_units = trial.suggest_int('lstm_units', 128, 256)

    base_model = self._build_model(trial)
    ctc_model = self._create_ctc_model(base_model)
    optimizer = Adam(learning_rate) if optimizer_name == 'adam' else Nadam(learning_rate)
    ctc_model.compile(optimizer, loss=lambda y_true, y_pred: y_pred)

    callbacks = [EarlyStopping(patience=7), ReduceLROnPlateau(patience=4), OptunaMetricsCallback(trial)]
    ctc_model.fit(train_seq, val_data, epochs=10, batch_size=batch_size, callbacks=callbacks)
    return trial.user_attrs['best_val_cer']
    # Full implementation in Appendix A.4
```

Figure 5.19 Code snippet for Optuna objective function

An Optuna study was created with a median pruner from optuna.pruners to early-stop unpromising trials, reducing computational waste. The validation CER at every epoch was captured by a custom OptunaMetricsCallback, which was utilized by the pruner to end trials with poor performance. The optimum configuration was then stored in a JSON file using Python json.dump() function as in Figure 5.20.

```
class OptunaMetricsCallback(Callback):
    def __init__(self, trial):
        self.trial = trial
        self.best_cer = float('inf')

    def on_epoch_end(self, epoch, logs=None):
        val_cer = logs.get('val_cer')
        if val_cer:
            self.best_cer = min(self.best_cer, val_cer)
            self.trial.report(val_cer, epoch)
            if self.trial.should_prune():
                raise optuna.TrialPruned()

# Create study with median pruner
study = optuna.create_study(direction='minimize', pruner=optuna.pruners.MedianPruner())

# Run optimisation
study.optimize(objective, n_trials=20)

# Save best configuration
with open('best_config.json', 'w') as f:
    json.dump(study.best_params, f, indent=4)
```

Figure 5.20 Code snippet for Optuna study creation

After 20 optimisation trials, the best hyperparameter configuration emerged and is presented in Table 5.3. The optimised model is significantly larger than the baseline by having about 5.01 million trainable parameters as opposed to 1.98 million. This increase is primarily due to larger LSTM units and increased CNN filters. It also employed lower dropout rates and weaker L2 regularisation, while the optimiser was changed to Nadam with a smaller batch size.

Table 5.3 Baseline vs Optimised model configuration

Parameter	Baseline	Optimised
CNN filters (block 1-4)	32, 64, 128, 256	64, 64, 256, 256
CNN kernel size	3 × 3	5 × 5 (first), 3 × 3 (second)
CNN dropout	0.7	0.53
LSTM units	128	256
LSTM layers	2	2
LSTM dropout	0.6	0.41
L2 regularisation	0.001	1.03e-5
Optimiser	Adam	Nadam
Learning rate	0.001	0.001
Batch size	16	8
Total parameters	1,981,377	5,010,689
Trainable parameters	1,980,417	5,009,409

5.2.4 Optimised Model Retraining

Following 20 trials with Optuna, the best hyperparameters were read from best_config.json with Python's json.load() method. The optimised model was constructed using build_from_config(), which sets CNN filters, kernel sizes, dropout, LSTM units, L2 regularisation and so on. It was trained again for 30 epochs, then saved for deployment through the API, as illustrated in Figure 5.21.

```
# Load best hyperparameters
with open('best_config.json', 'r') as f:
    best = json.load(f)

# Build and compile optimised model
ctc_model = self.build_from_config(best)
optimizer = Nadam(best['learning_rate']) if best['optimizer'] == 'nadam' else Adam(best['learning_rate'])
ctc_model.compile(optimizer, loss=lambda y_true, y_pred: y_pred)

# Retrain and save
ctc_model.fit(train_seq, val_data, epochs=30, batch_size=best['batch_size'], callbacks=callbacks)
ctc_model.save('optimised_model.h5')
```

Figure 5.21 Code snippet for building optimised model

5.2.5 Training Performance

Both the baseline and optimised model were trained for 30 epochs. The training and validation loss curves are shown in Figure 5.22. The optimised model achieved substantially faster initial convergence. After the first epoch, its validation loss dropped to 55.79, while the baseline remained at 95.83. This rapid initial convergence was attributed to the Nadam optimiser, larger model capacity of 5.01 million parameters, and reduced regularisation, which enabled more aggressive learning in the early training phase. The optimised model had consistently lower validation loss than the baseline during training. The optimised model's final validation loss was 12.26, while the baseline's was 14.86.

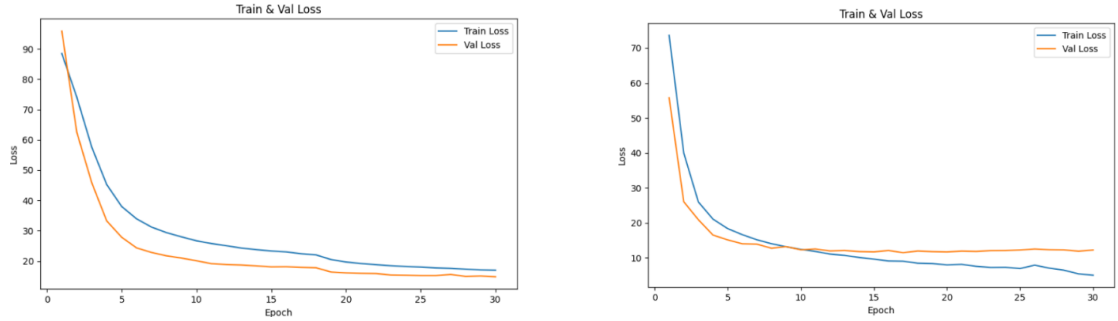

Figure 5.22 Training and validation loss curves: baseline vs optimised

Figure 5.23 presents the training and validation CER curves for both models. The optimised model demonstrated significantly faster initial convergence. After the first epoch, its validation CER dropped to 79.96%, while the baseline remained at 100%. The final validation CER of the optimised model was 8.55% at epoch 30, compared to 10.54% for the baseline.

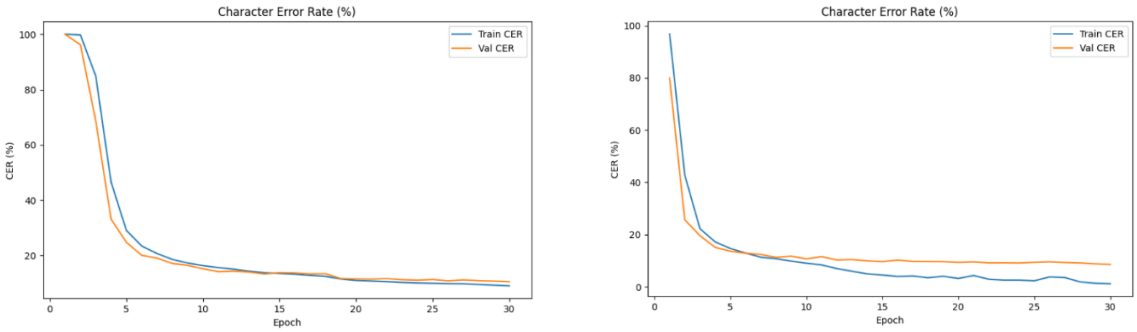

Figure 5.23 Training and validation CER curves: baseline vs optimised

The training and validation WER curves are plotted in Figure 5.24. Similar to the CER curves, the optimised model achieved faster initial convergence and consistently lower

validation WER. The final validation WER of the optimised model was 22.42% at epoch 30, compared to 27.22% for the baseline.

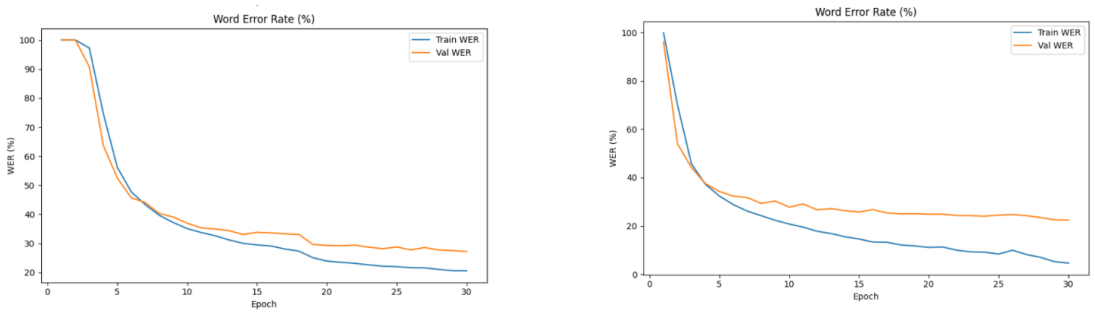

Figure 5.24 Training and validation WER curves: baseline vs optimised

Figure 5.25 presents the training and validation Exact Match Accuracy curves for both models. The optimised model achieved higher validation accuracy throughout most of the training process. The final validation accuracy of the optimised model was 44.72% at epoch 30, whereas the baseline model's was 40.57%.

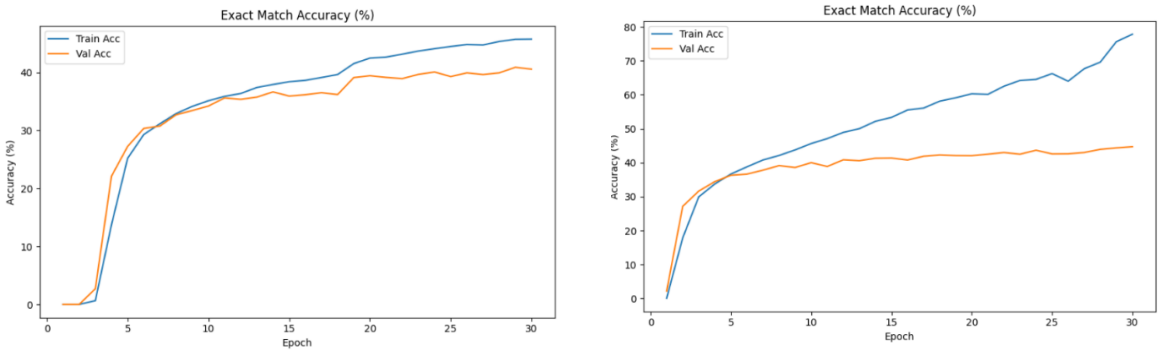

Figure 5.25 Training and validation Exact Match Accuracy curves: baseline vs optimised

Although the optimised model showed improved performance on all the validation metrics, it had greater training-validation gaps. The CER gap between training and validation was 7.39% for the optimised model and 1.48% for the baseline, suggesting that the optimised model, being bigger with 5.01 million parameters compared to 1.98 million and less regularised, was more susceptible to overfitting. However, it performed better than the baseline on all validation metrics throughout training. As mentioned in Table 5.4, the training metrics of the two models in the best epoch are listed.

Table 5.4 Training summary at best epoch

Metric	Baseline (Epoch 30)	Optimised (Epoch 30)
Validation Loss	14.86	12.26
Validation CER	10.54%	8.55%
Validation WER	27.22%	22.42%
Validation Exact Match	40.57%	44.72%

5.3 API Implementation

FastAPI was used to implement the API to serve the trained HTR model remotely. As depicted in Figure 5.26, the application presented a "/predict" endpoint to POST requests with an image file using UploadFile type, and a "/health" endpoint to monitor the health of the application.

```
from fastapi import FastAPI, UploadFile, File
app = FastAPI()

@app.post("/predict")
async def predict(file: UploadFile = File(...)):
    img_np = np.array(Image.open(io.BytesIO(await file.read())).convert("RGB"))
    results = {}
    for field, crop in dynamic_crop_fields(img_np).items():
        processed = preprocess_image(crop)
        cleaned = post_processor.process_field(field, decode_preds(base_model.predict(processed)))
        results[field] = cleaned
    return results

@app.get("/health")
async def health_check():
        return {"status": "healthy"}
```

Figure 5.26 Code snippet for FastAPI endpoint implementation

When an image was received, the API applied dynamic cropping with the EasyOCR library, as illustrated by Figure 5.27. The reader.readtext() method identified all text blocks and the get_stable_crop() function cropped handwritten fields depending on the spatial proximity of the printed label anchors.

```
import easyocr, cv2, numpy as np

reader = easyocr.Reader(['en'], gpu=False)

def dynamic_crop_fields(img_np):
    # Resize for faster OCR
    h, w = img_np.shape[:2]
    scale = 1600.0 / w
    img_small = cv2.resize(img_np, (0,0), fx=scale, fy=scale)

    # Detect text blocks
    results = reader.readtext(img_small, text_threshold=0.15, paragraph=False)

    # Convert to block format with absolute coordinates
    all_blocks = []
    for (bbox, text, prob) in results:
        pts = (np.array(bbox) / scale).astype(int)
        all_blocks.append({'text': text.lower().strip(), 'center': np.mean(pts, axis=0)})
   # Crop each field by anchor keywords
    crop_dict = {}
    for field_name, keyword in [('SeatNo','seat'), ('IndexWords','words'),
                                ('IndexFigures','figures'), ('Course','course')]:
        crop = get_stable_crop(img_np, all_blocks, keyword, w, h)
        if crop:
            crop_dict[field_name] = crop
    return crop_dict
```

Figure 5.27 Code snippet for cropping with EasyOCR

Figure 5.28 illustrates each cropped field was then preprocessed with OpenCV and NumPy functions to perform cv2.cvtColor(), cv2.GaussianBlur(), cv2.createCLAHE(), cv2.threshold() with OTSU flag, cv2.morphologyEx(), and cv2.resize(). Pixel values were normalised to [0, 1] using NumPy's division operator, and a channel dimension was added with np.expand_dims().

```
import cv2
import numpy as np

def preprocess_image(img_np, target_height=32, target_width=640):
    gray = cv2.cvtColor(img_np, cv2.COLOR_RGB2GRAY) if len(img_np.shape) == 3 else img_np

    blurred = cv2.GaussianBlur(gray, (5, 5), 0)
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
    enhanced = clahe.apply(blurred)

    _, binary = cv2.threshold(enhanced, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)

    kernel = np.ones((2, 2), np.uint8)
    binary = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel)
    binary = cv2.morphologyEx(binary, cv2.MORPH_CLOSE, kernel)

    h, w = binary.shape
    new_w = int(w * (target_height / h))
    resized = cv2.resize(binary, (min(new_w, target_width), target_height))
```

```
    padded = np.ones((target_height, target_width), dtype=np.uint8) * 255
    padded[:, :resized.shape[1]] = resized
    return np.expand_dims(padded.astype(np.float32) / 255.0, axis=(0, -1))
```

Figure 5.28 Code snippet for preprocessing

The transformed image was then fed into the TensorFlow model's predict() function. The output of the model was decoded using Tensorflow's ctc_decode function with greedy decoding, as illustrated in Figure 5.29. A ctc_decode function was used to convert the probability matrix into character indices as well as remove the blank tokens and repeated characters to create the final text string.

```
from tensorflow.keras import backend as K
def decode_preds(preds):
    input_len = np.ones(preds.shape[0]) * preds.shape[1]
    decoded, _ = K.ctc_decode(preds, input_length=input_len, greedy=True)
    indices = decoded[0].numpy()[0]

    # Remove blank tokens and -1 padding
    seq = [int(s) for s in indices if s != -1 and s != blank_label]
    return ''.join([char_list[c] for c in seq if 0 <= c < len(char_list)])

# Inference and decoding
preds = base_model.predict(processed, verbose=0)
raw_text = decode_preds(preds)
```

Figure 5.29 Code snippet for CTC decoding using TensorFlow

After decoding, post-processing was implemented by the AttendancePostProcessor class as shown in Figure 5.30. The process_field() method gave a mapping of all the fields to their cleaning functions and applied error corrections through fuzzy matching and character mapping.

```
class AttendancePostProcessor:
    def process_field(self, field_name, text):
        if field_name == 'SeatNo':
            return self.clean_seat_number(text)
        elif field_name == 'IndexWords':
            return self.clean_index_words(text)
        elif field_name == 'IndexFigures':
            return self.clean_index_figures(text)
        elif field_name == 'Course':
            return self.clean_course(text)
        return text

# Apply post-processing
post_processor = AttendancePostProcessor(config)
cleaned_text = post_processor.process_field(field, raw_text)
```

Figure 5.30 Code snippet for post-processing with AttendancePostProcessor

In addition, error handling was implemented using FastAPI's HTTPException, raising status codes 422 for missing files, 400 for invalid formats, and 500 for processing failures. The API was finally deployed on Hugging Face Spaces, where the platform automatically installed dependencies and started the Uvicorn server on host 0.0.0.0 and port 7860, as shown in Figure 5.31.

```
import uvicorn

# Start the server
if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=7860)
```

Figure 5.31 Code snippet for starting the Uvicorn server

After deployment, FastAPI automatically generated interactive API documentation, as shown in Figure 5.32. The "/predict" endpoint accepted a POST request with an image file and returned a JSON response containing SeatNo, IndexWords, IndexFigures, and Course.

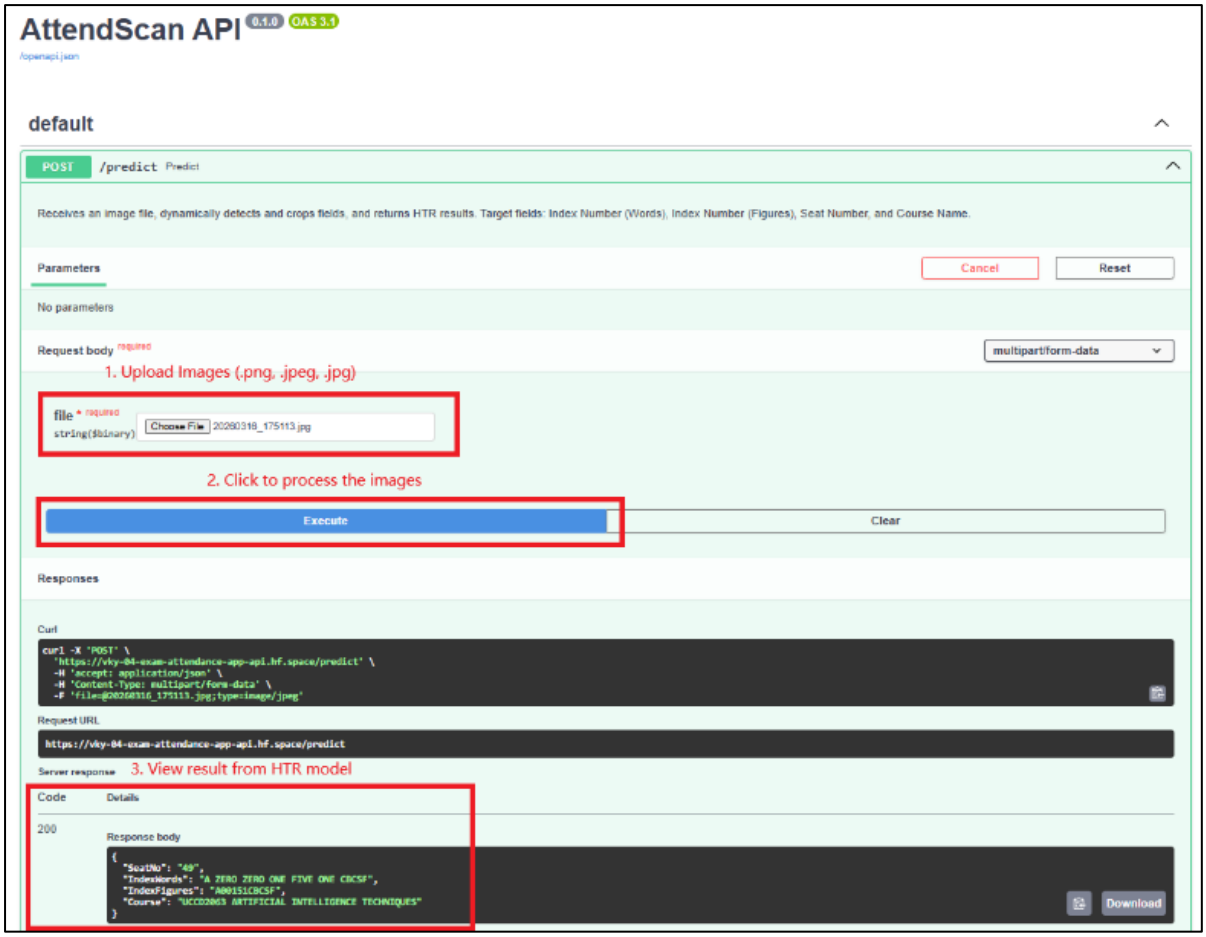

Figure 5.32 FastAPI Swagger UI showing API response

5.4 Database Implementation

This section presents the implementation of the hybrid database architecture using Firebase Firestore for cloud storage and SharedPreferences for local caching.

5.4.1 Cloud Database: Firebase Firestore

The Firebase Firestore database was composed of three collections as presented in Figure 5.33, Figure 5.34, and Figure 5.35: “users" containing student and administrator profiles, “exams” containing examination details, and “attendance” containing attendance records.

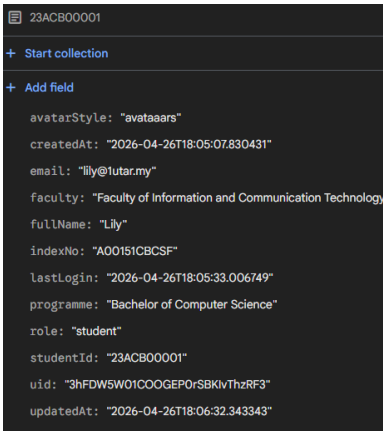

Figure 5.33 Firestore “users” collection

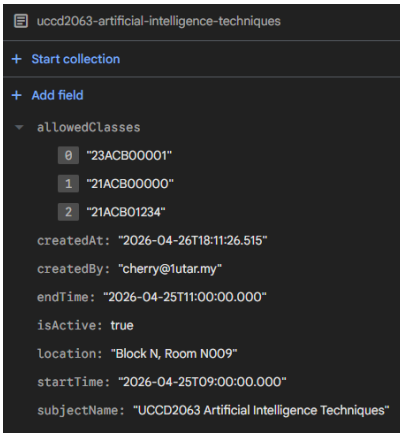

Figure 5.34 Firestore “exams” collection

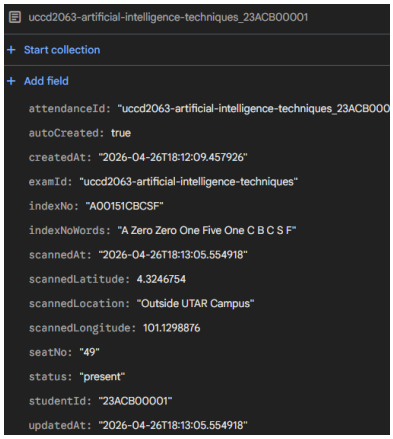

Figure 5.35 Firestore “attendance” collection

The FirebaseService class was developed based on Firebase SDK in Flutter that could support both Android application and the website platform. The methods of Firestore were collection().doc().set() to create a document, collection().where().get() to query, and batch() to run a transaction, as shown in Figure 5.36.

```
# Example 1: Create document
await FirebaseFirestore.instance
    .collection('users').doc('23ACB00001')
    .set({'email': 'student@utar.edu.my', 'role': 'student'});

# Example 2: Query with filter
await FirebaseFirestore.instance
    .collection('exams').where('isActive', isEqualTo: true).get();

# Example 3: Batch write
WriteBatch batch = FirebaseFirestore.instance.batch();
batch.update(examRef, {'isActive': false});
batch.update(attendanceRef, {'status': 'present'});
await batch.commit();
```

Figure 5.36 Code snippet for Firestore operations

5.4.2 Local Database: SharedPreferences

The LocalStorageService class was implemented using the shared_preferences plugin for Flutter on the Android application. The saveLoginStatus() method stored user session data using setString() and setBool(), while cacheExams() serialised data to JSON using jsonEncode() before storage. Offline operations were managed through a sync_queue using getStringList() and setStringList(), as shown in Figure 5.37.

```
class LocalStorageService {
  // Save user login session to local storage
  Future<void> saveLoginSession(Map<String, dynamic> userData) async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.setString('user_data', jsonEncode(userData));
    await prefs.setBool('is_logged_in', true);
  }

  // Cache exam list with timestamp
  Future<void> cacheExams(List<dynamic> exams) async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.setString('cached_exams', jsonEncode(exams));
    await prefs.setString('last_sync_time', DateTime.now().toIso8601String());
  }

  // Add pending operation to offline sync queue
  Future<void> addToSyncQueue(Map<String, dynamic> operation) async {
    final prefs = await SharedPreferences.getInstance();
    final queue = prefs.getStringList('sync_queue') ?? [];
    queue.add(jsonEncode(operation));
    await prefs.setStringList('sync_queue', queue);
  }
}
```

Figure 5.37 Code snippet for SharedPreferences operations

5.4.3 Connectivity and Synchronisation

The ConnectivityService class was implemented using the connectivity_plus plugin to monitor network state through the onConnectivityChanged listener. Once connectivity was restored, the service triggered the SyncManager to flush the sync_queue, as shown in Figure 5.38

```
class ConnectivityService {
  void monitorConnectivity() {
    Connectivity().onConnectivityChanged.listen((result) {
      if (result != ConnectivityResult.none) {
        _syncPendingOperations();  // Auto-sync when online
      }
    });
  }
}
```

Figure 5.38 Code snippet for connectivity monitoring

The SyncManager iterated through the sync_queue and applied each operation using its stored type and payload. Operations were deserialised from JSON and dispatched to the corresponding Firebase service method. Successfully committed operations were removed from the queue, while failures triggered a retry in the next sync cycle, as illustrated in Figure 5.39.

```
class SyncManager {
  Future<void> processSyncQueue() async {
    final queue = _storage.getSyncQueue();
    for (final op in queue) {
      try {
        await _executeOperation(op['type'], op['data']);
        _storage.removeFromQueue(op);
      } catch (e) {
        // Keep for retry
      }
    }
  }
}
```

Figure 5.39 Code snippets for processing pending operations

5.5 Mobile Application and Web Platform Implementation

The mobile application and web platform were developed in Flutter 3.0 and Dart, allowing code to be shared between Android and web browser. The mobile version was bundled into an APK and the web version was hosted using Firebase Hosting.

5.5.1 Code Implementation

This section presents the key code implementations for API integration and Firebase authentication.

API Integration (Dio)

The mobile application relied on Dio package to communicate with the HTR API deployed on Hugging Face Spaces. Figure 5.40 illustrates the API endpoint that was defined as a fixed string and the image was transferred using multipart form data.

```
final String apiUrl = "https://vky-04-exam-attendance-app-api.hf.space/predict"

Future<void> _processImage(String path) async {
  FormData formData = FormData.fromMap({
    "file": await MultipartFile.fromFile(path, filename: "capture.jpg"),
  });
  var response = await Dio().post(apiUrl, data: formData);
  if (response.statusCode == 200) {
    await _processHTRResult(response.data);
  }
}
```

Figure 5.40 Code snippet for Dio API call

Upon receiving the API response, the extracted fields were extracted and a confirmation dialogue is shown to the student. The processing of the HTR response is shown in Figure 5.41.

```
// Process the HTR API response and show confirmation dialogue
Future<void> _processHTRResult(Map<String, dynamic> HTRData) async {
  // Extract the four recognised fields from the API response
  final scannedIndexWords = HTRData['IndexWords']?.toString().trim() ?? '';
  final scannedFullIndex = HTRData['IndexFigures']?.toString().trim() ?? '';
  final scannedSeatNo = HTRData['SeatNo']?.toString().trim() ?? '';
  final scannedCourse = HTRData['Course']?.toString().trim() ?? '';

  // Prepare data for the confirmation dialogue
  final scanData = {
    'subjectName': scannedCourse,
    'indexWords': scannedIndexWords,
    'fullIndex': scannedFullIndex,
    'seatNo': scannedSeatNo,
  };

  // Display the confirmation dialogue for user review
  await _showScanDetailsDialog(scanData);
}
```

Figure 5.41 Code snippet for processing HTR API response

Firebase Authentication

Firebase Authentication provided user registration and login. The implementation of the login method that authenticated the user and retrieved their role from Firestore “users” collection is shown in Figure 5.42.

```
Future<Map<String, dynamic>> loginUser({
  required String email,
  required String password,
}) async {
  // Authenticate with Firebase
  UserCredential userCredential = await _auth.signInWithEmailAndPassword(
    email: email, password: password,
  );
```

```
  // Retrieve user from Firestore
  final userQuery = await _firestore
      .collection('users')
      .where('email', isEqualTo: email)
      .limit(1)
      .get();

  if (userQuery.docs.isEmpty) {
    await _auth.signOut();
    return {'success': false, 'message': 'No user record found.'};
  }

  final userData = userQuery.docs.first.data();

  // Update last login timestamp
  await _firestore.collection('users').doc(userQuery.docs.first.id).update({
    'lastLogin': DateTime.now().toIso8601String(),
  });

  return {'success': true, 'role': userData['role'], 'uid': userCredential.user!.uid};
}
```

Figure 5.42 Code snippet for Firebase authentication

5.5.2 System Operation (Mobile Application)

Login and Registration Pages

Figure 5.43 shows the login and registration forms when first launching the application. Student is required to input their email address and password associated with their account to login. New users are required to create a new account using a valid email address. After registering, the account is set up in Firebase Authentication, and the student details are stored in the Firestore “users” collection. In case of forgotten passwords, the student can enter their email address to receive a reset link.

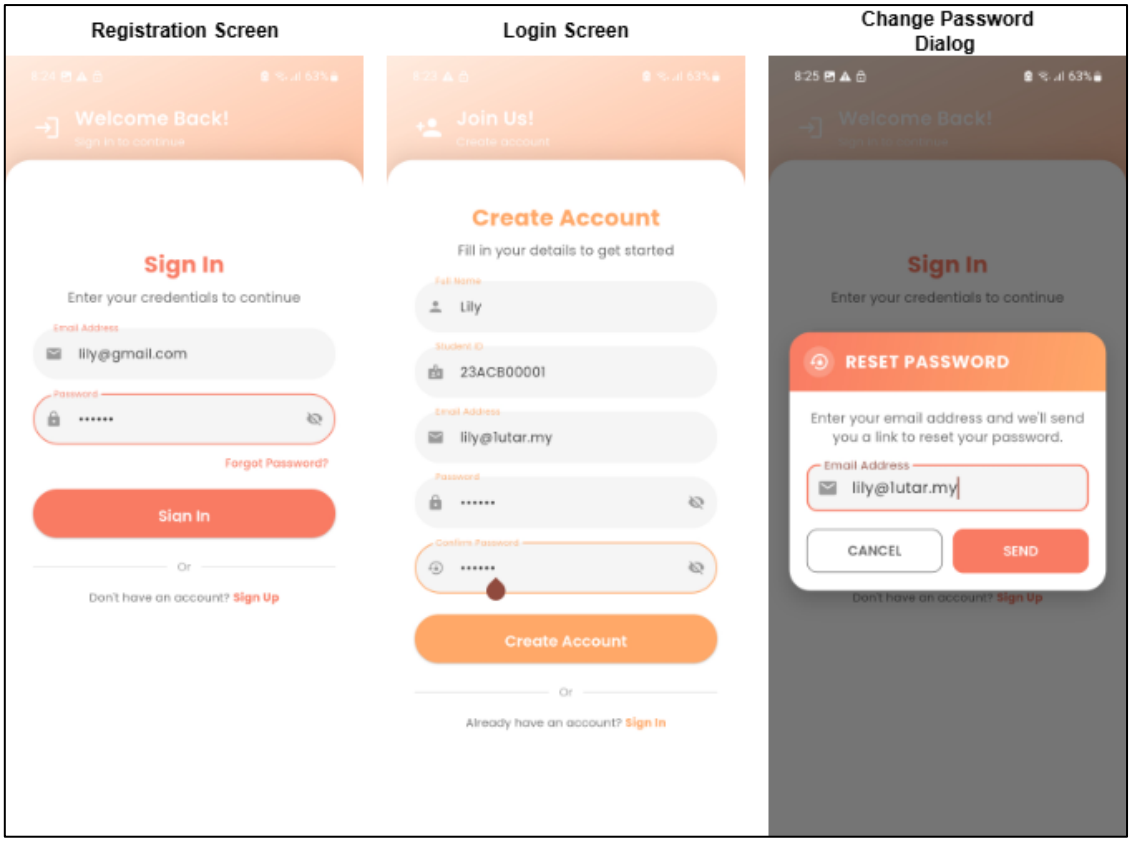

Figure 5.43 Login, registration, and password reset screens

Home Page

After successful login, the home screen displays upcoming exams and an attendance overview, as shown in Figure 5.44. Exam data is retrieved from Firestore, which queried the "exams" collection, and cached copies are used when the device is offline. A menu button located in the top left corner allows students to switch between dark mode and light mode or log out.

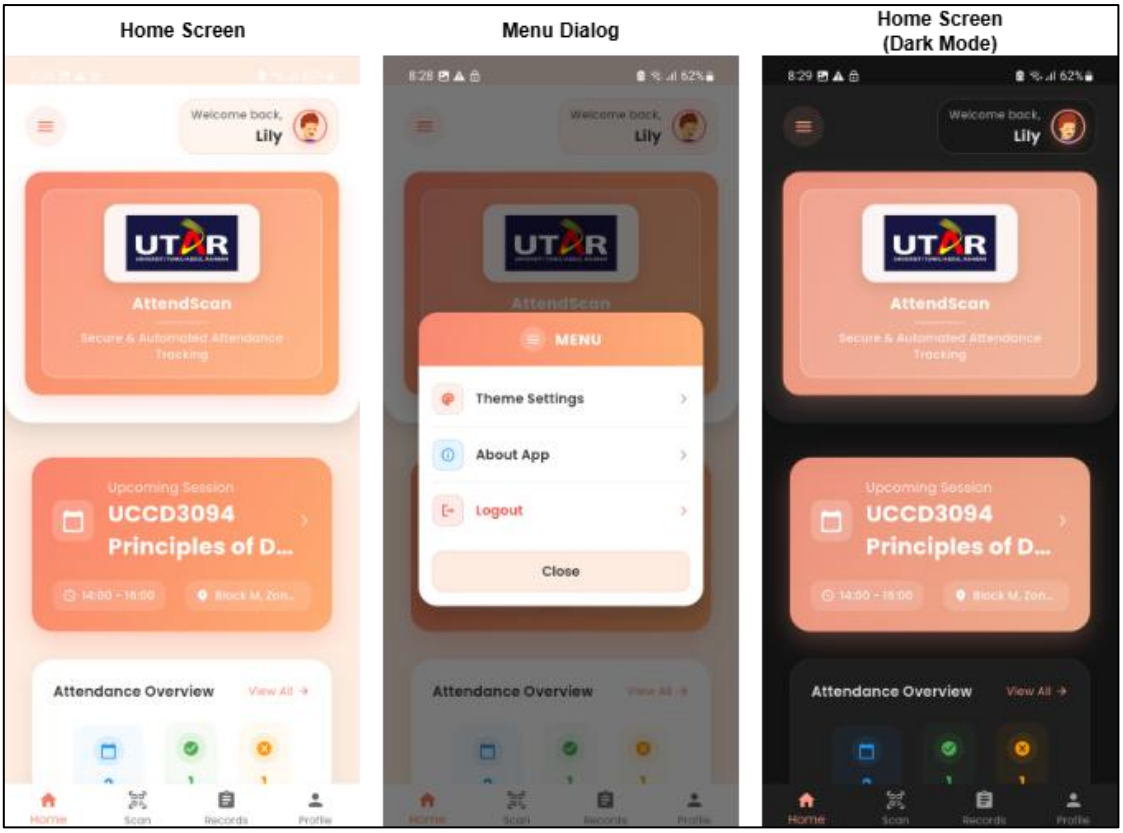

Figure 5.44 Home screen with menu

Scan page (Attendance Scanning and Validation)

The workflow of attendance marking is illustrated in Figure 5.45. Students can capture a photo of their exam attendance slip using the device camera or select an existing image from the gallery. The image is sent to the HTR API deployed on Hugging Face Spaces for handwriting recognition. A loading indicator is shown as the system works on the image and extracts the handwritten fields. The system then presents the confirmation dialogue to request students to review the recognised information, make any necessary changes, confirm the information, and lastly submit the attendance.

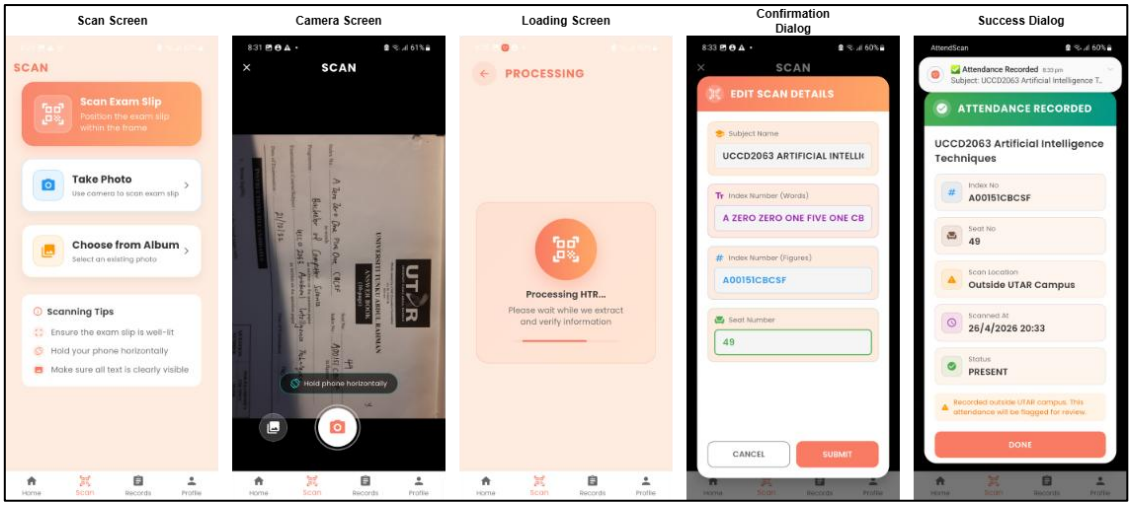

Figure 5.45 Attendance scanning workflow screens

Three validations are performed upon submission: matching subject name with an active exam, matching the index number in figures with the index number in words, and matching the scanned index number with the registered one. A failure in any of these cancels the submission and displays an error dialogue, as illustrated in Figure 5.46.

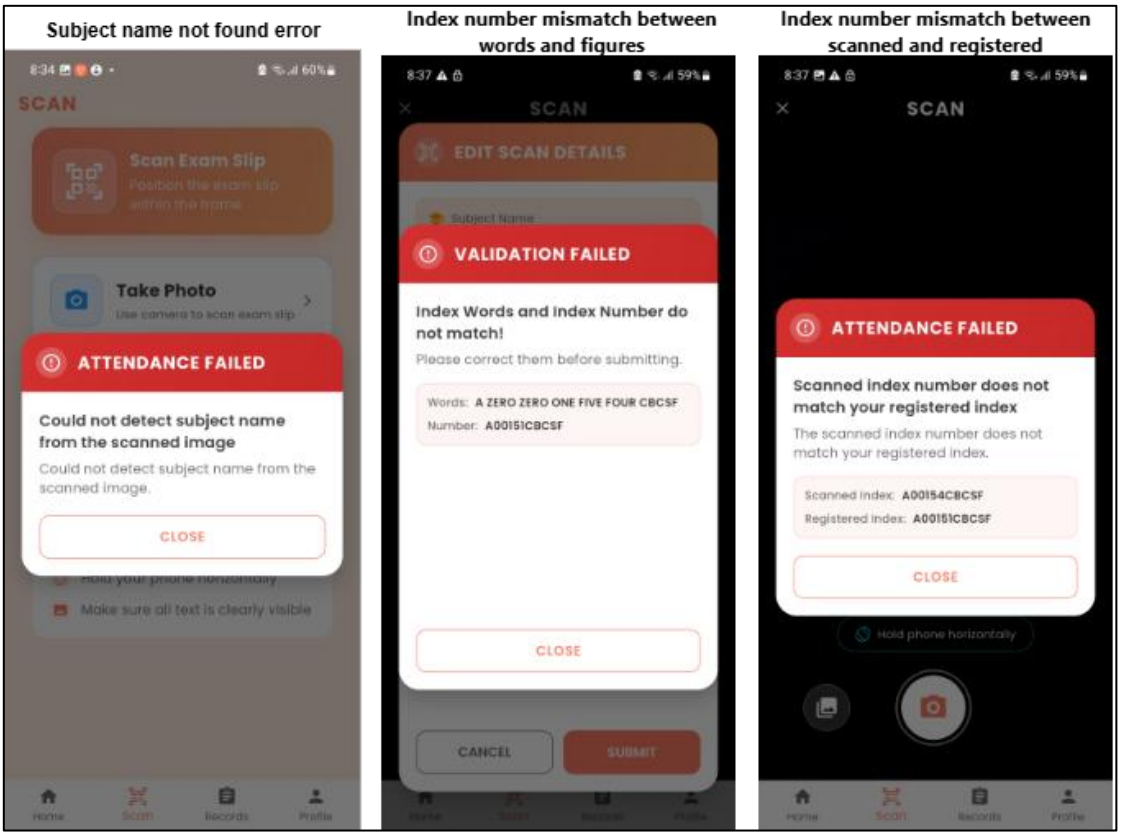

Figure 5.46 Validation error dialogue

If validation passes, a success dialogue appears with the submitted information, and a notification indicates that the attendance record has been uploaded successfully to Firestore "attendance" collection, as shown in Figure 5.45.

Record Page

Figure 5.47 shows the attendance records page. The students will be able to view an overview of the attendance on top of the page which includes the number of all exams, upcoming exams, present exams and absent exams as well as the rate of attendance. Below the overview, detailed attendance records are listed. By clicking on any record in the list, a dialogue is shown containing the detailed information of that attendance record.

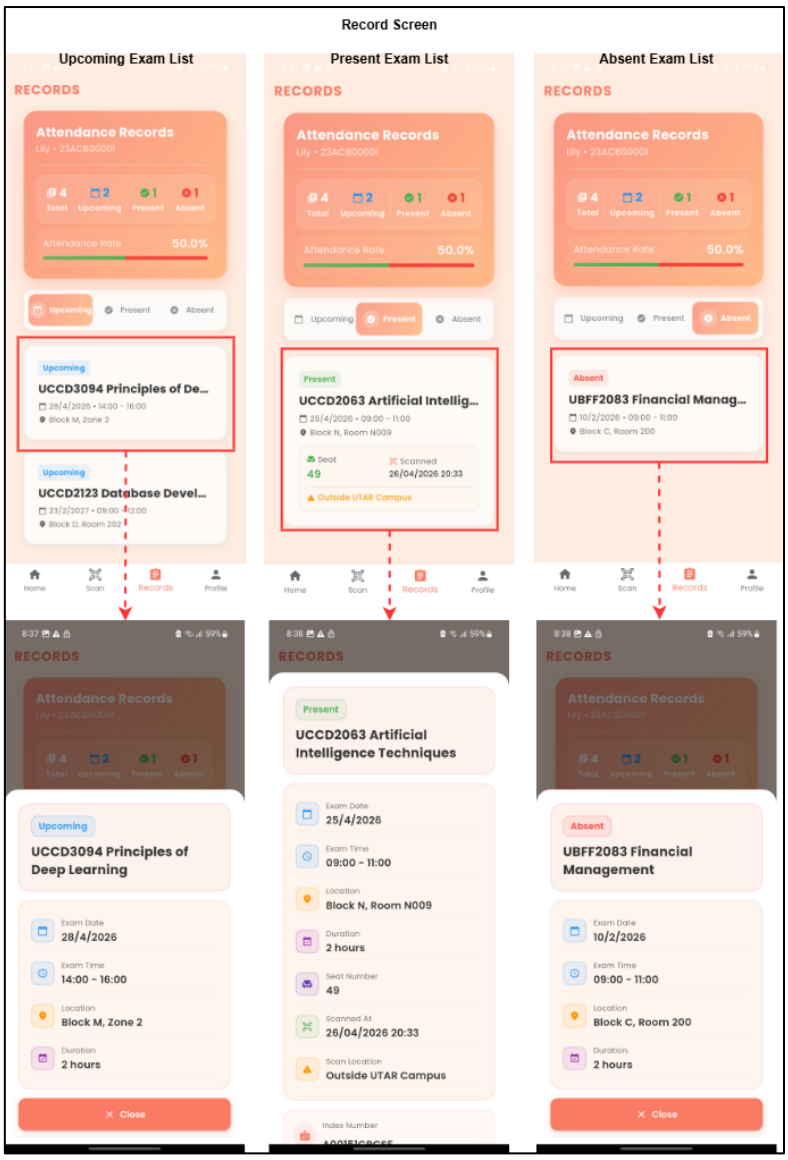

Figure 5.47 Attendance records and details screens

Profile Page

The profile page is illustrated in Figure 5.48. Students can access to their personal details such as username, registered email, student ID, index number, faculty, programme, and date of account creation. This page allows the user to update profile details or password. Updates were saved to the Firestore “users” collection.

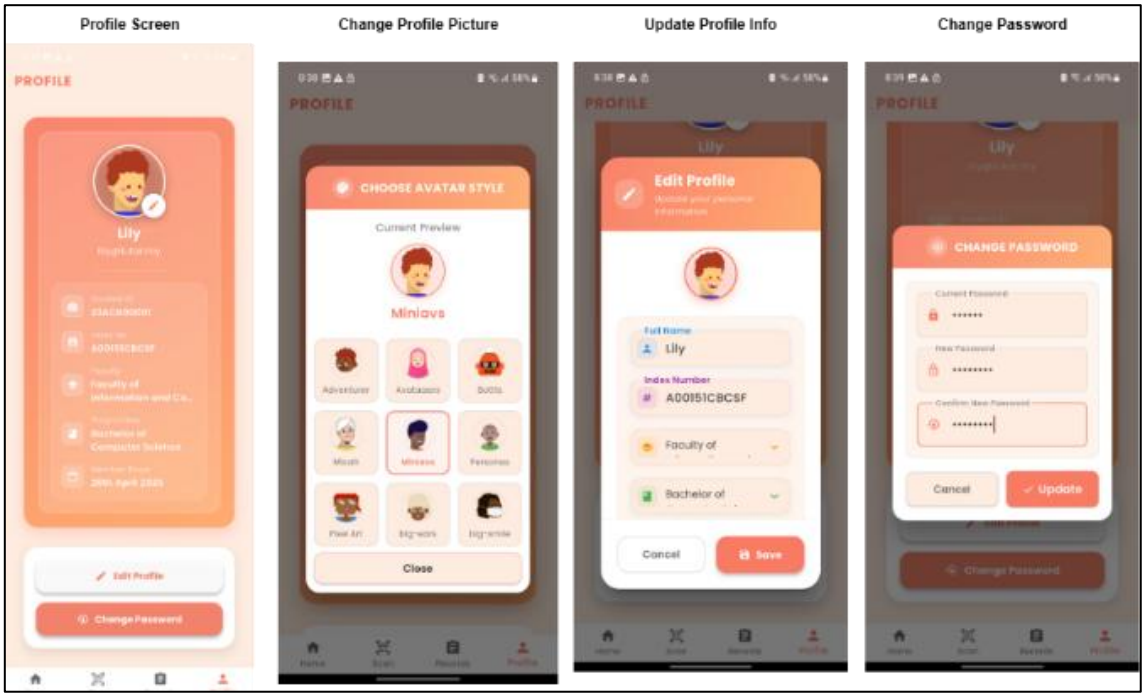

Figure 5.48 Profile screen with edit and password reset

Offline Support

The application caches exam schedules and user profiles locally using SharedPreferences. This allows students to view the cached data even when the device is offline and all the pending changes are stored in a sync queue that will be automatically synced with the server once the device is connected to network. Figure 5.49 presents the profile being updated offline and will be synced when the device is online.

Figure 5.49 Offline profile modification with sync queue

5.5.3 System Operation (Web Platform)

Login and Registration Pages

The web registration and login pages are shown in Figure 5.50 and Figure 5.51. Administrators can register a new account by entering their email and password or use an existing account to log in. User accounts are managed by Firebase Authentication and data stored in Firestore "users" collection.

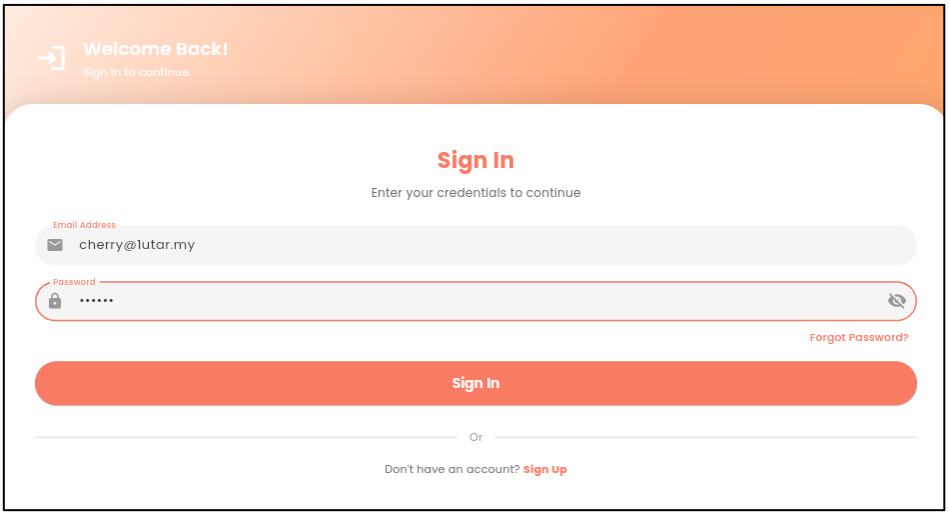

Figure 5.50 Login screen (for admin)

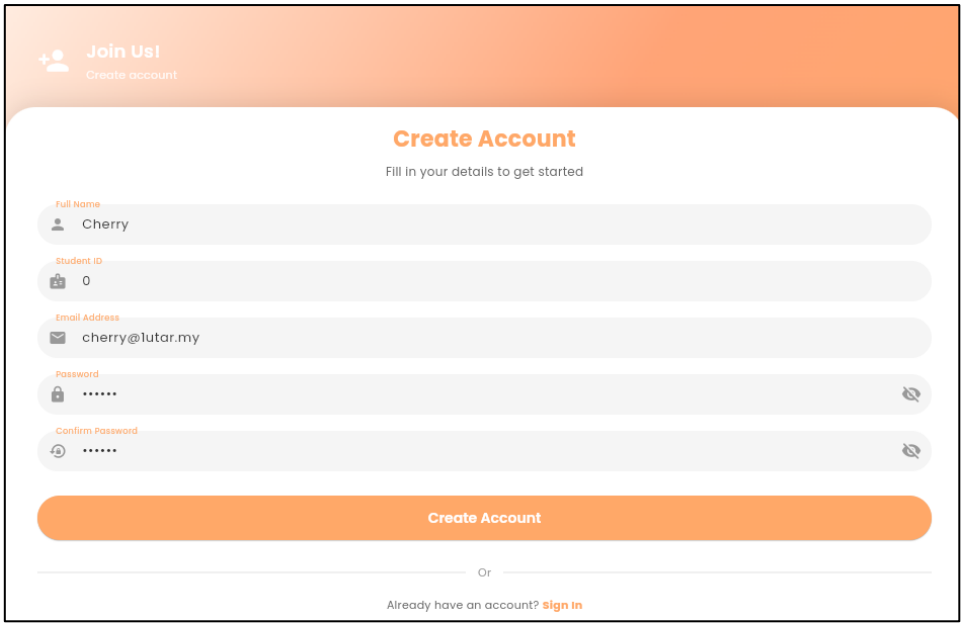

Figure 5.51 Registration screen (for admin)

Dashboard Page

The dashboard is shown in Figure 5.52. Upon logging in, administrators are presented with an overview of the total number of exams, active exams, students and today's exams. There are three quick action buttons that lead directly to the exam management page, the student management page and the attendance management page.

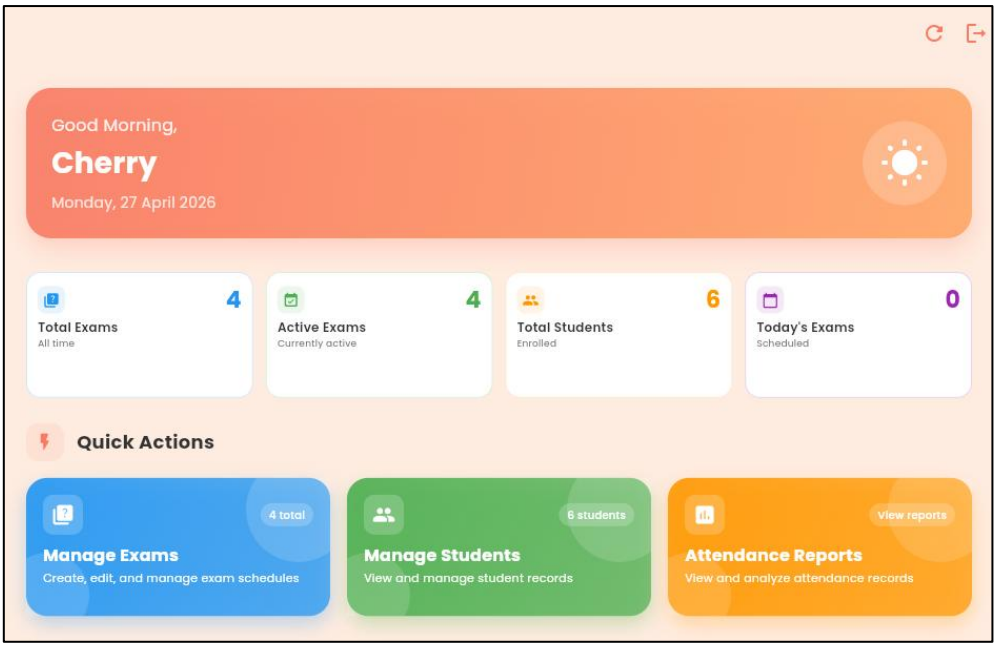

Figure 5.52 Dashboard screen

Exam Management Page

The exam management page is illustrated in Figure 5.53. The top section includes an exam overview showing the number of upcoming exams, current exams, and total enrolled students. A search bar allows administrators to find exams by subject name. Filter options can be used to display upcoming, past, active, or inactive exams. Sorting is available by date, subject, or number of assigned students. Below these controls, a list of all created exams is shown. To make a new exam, the administrator can press the plus icon in the upper right part of the page.

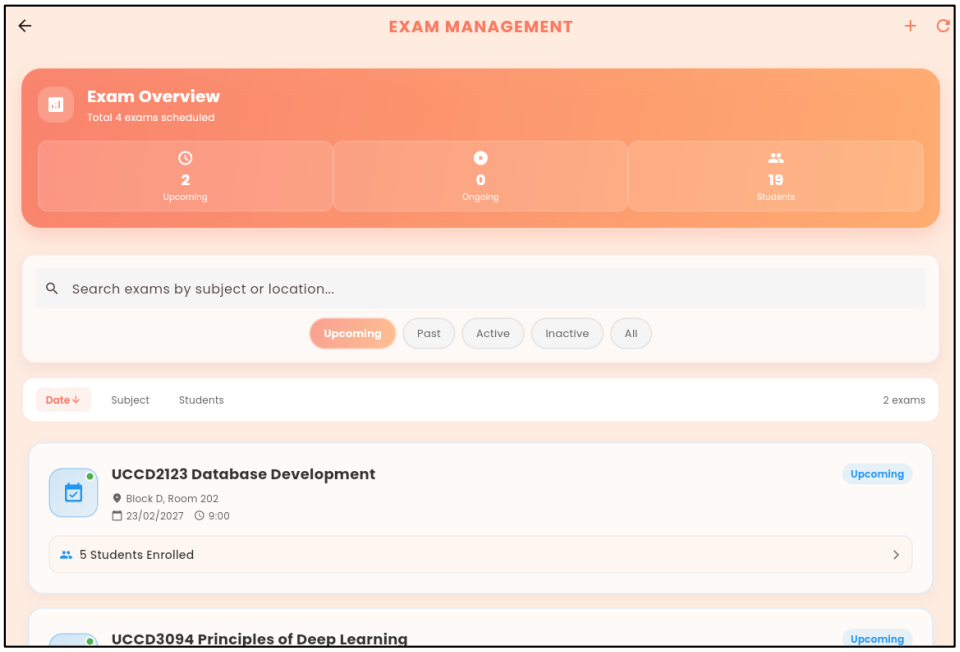

Figure 5.53 Exam management screen

Figure 5.54 and Figure 5.55 show the create exam dialogue. Administrators can enter the subject name, location, and schedule the start and end time. For student selection, they can choose students by faculty and programme, with a search bar to find students by name or student ID. Individual students can be ticked for enrolment. Once created, the exam is uploaded to the Firestore “exams” collection and immediately becomes visible to eligible students in their mobile application.

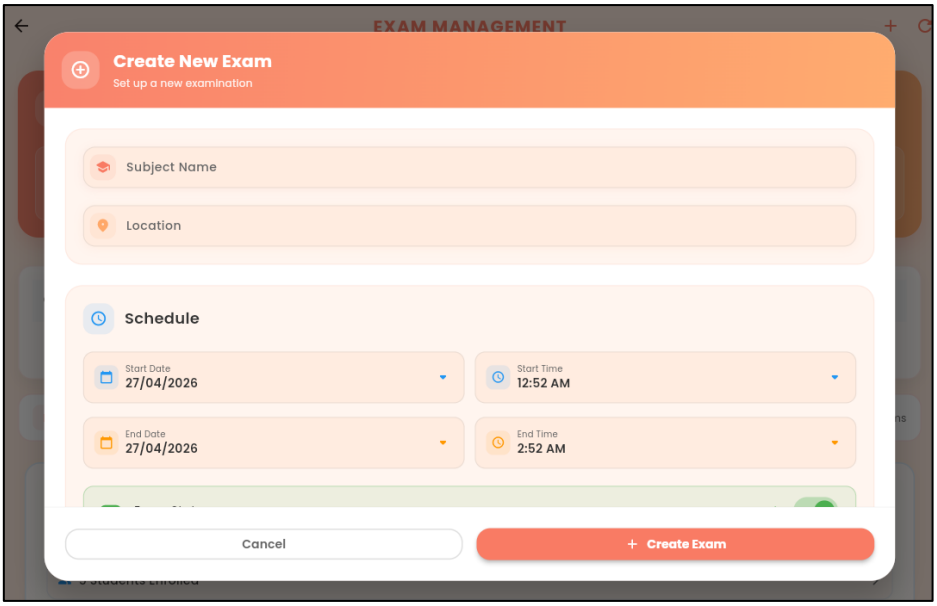

Figure 5.54 Create exam dialogue (a)

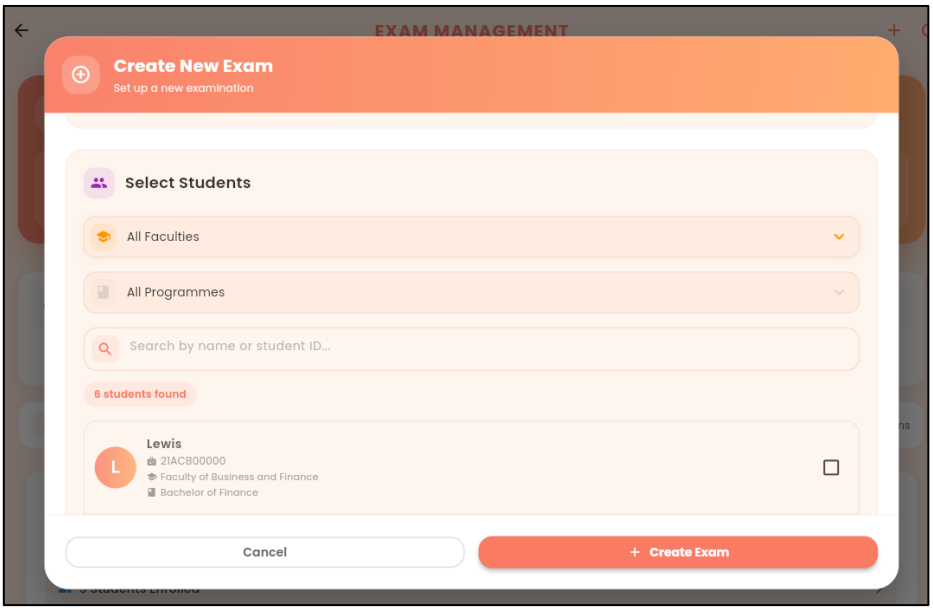

Figure 5.55 Create exam dialogue (b)

Student Management Page

Figure 5.56 illustrates the student management page. A search bar allows searching by name, student ID, or index number. Dropdown menus provide filtering by faculty and programme. Sorting options are available by name, ID, and faculty. Under these controls, a list of all the registered students is shown. Administrator can click on any student to see their detailed profile.

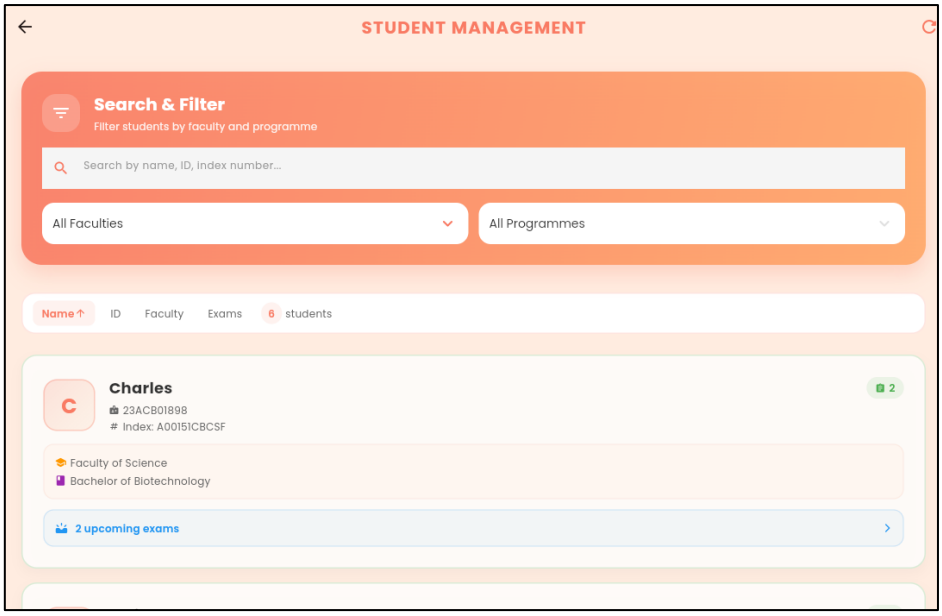

Figure 5.56 Student management screen

Figure 5.57 shows the student details dialogue, containing student information, enrolled exams and full attendance records. This enables administrators to check a student's attendance record across all exams.

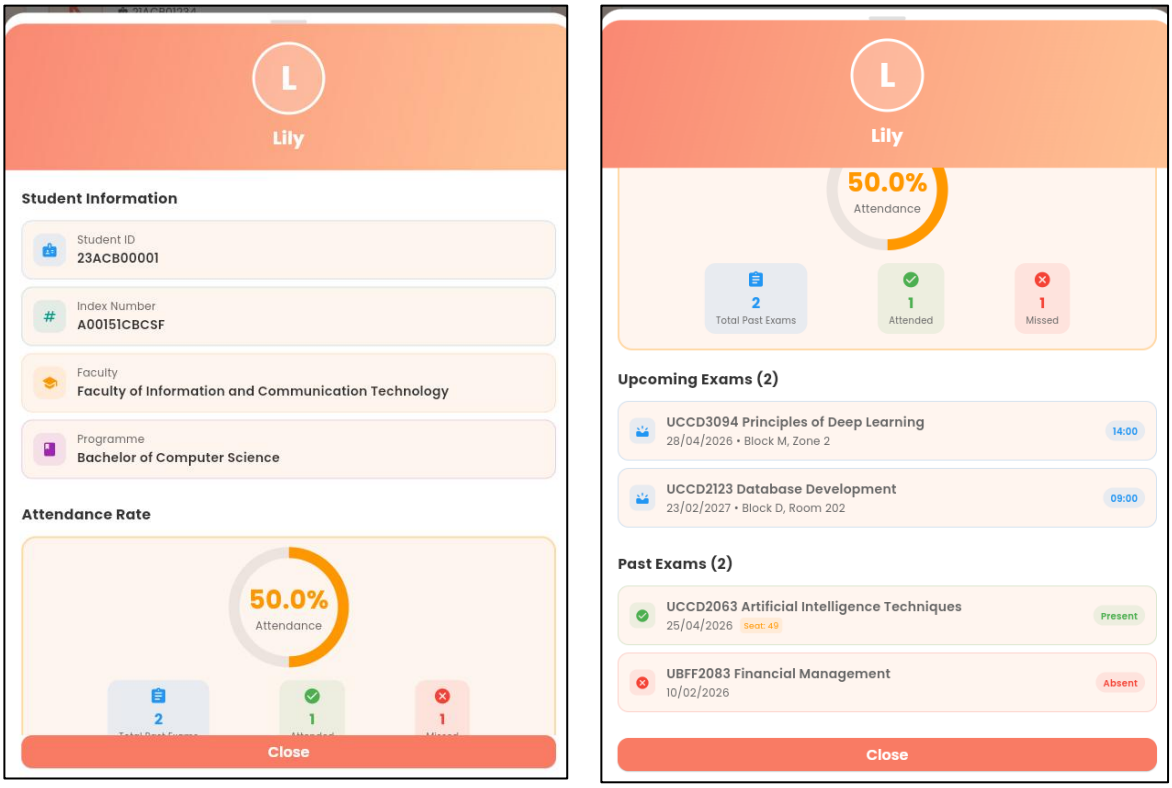

Figure 5.57 Student details dialogue

Attendance Management Page

The attendance management page is shown in Figure 5.58. Administrators select an exam from a dropdown menu and then view attendance. Tabs, search, and sorting functions help locate specific students. Each row displays a "P" for present or an "A" for absent status, along with an acknowledgement indicator and an optional off-campus indicator. An export button generates a CSV report as shown in Figure 5.59, and a batch acknowledgement button is also available.

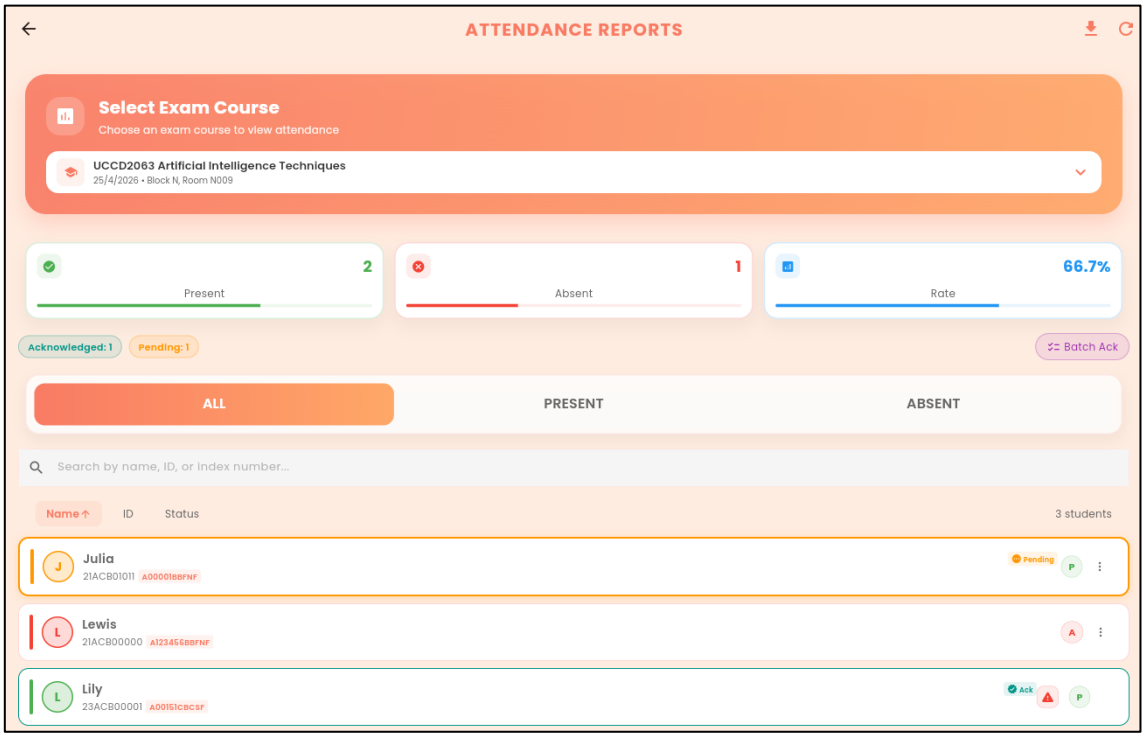

Figure 5.58 Attendance management screen

ATTENDANCE RECORDS							
Generated on: 2026-04-27 01:03:29							
UCCD2063 Artificial Intelligence Techniques	25/4/2026	09:00 - 11:00	Block N	Room N009			
Student ID	Full Name	Faculty	Programme	Index No	Status	Seat No	Scanned At
23ACB00001	Lily	Faculty of Information and Com	Bachelor of Computer Science	A00151CBCSF	PRESENT	49	26/04/2026 20:33:21
21ACB00000	Lewis	Faculty of Business and Finance	Bachelor of Finance	A123456BBFNF	ABSENT	-	-
21ACB01011	Julia	Faculty of Science	Bachelor of Mathematics	A00001BBFNF	PRESENT	22	27/04/2026 00:23:52
SUMMARY							
Total Students	Present	Absent	Attendance Rate				
3	2	1	66.67%				

Figure 5.59 Generated CSV report example

Figure 5.60 shows the detailed attendance view for a student. Clicking a student row opens a panel with warnings about off-campus check-in. The administrator reviews the evidence before deciding whether to verify the attendance, ensuring only legitimate records are counted.

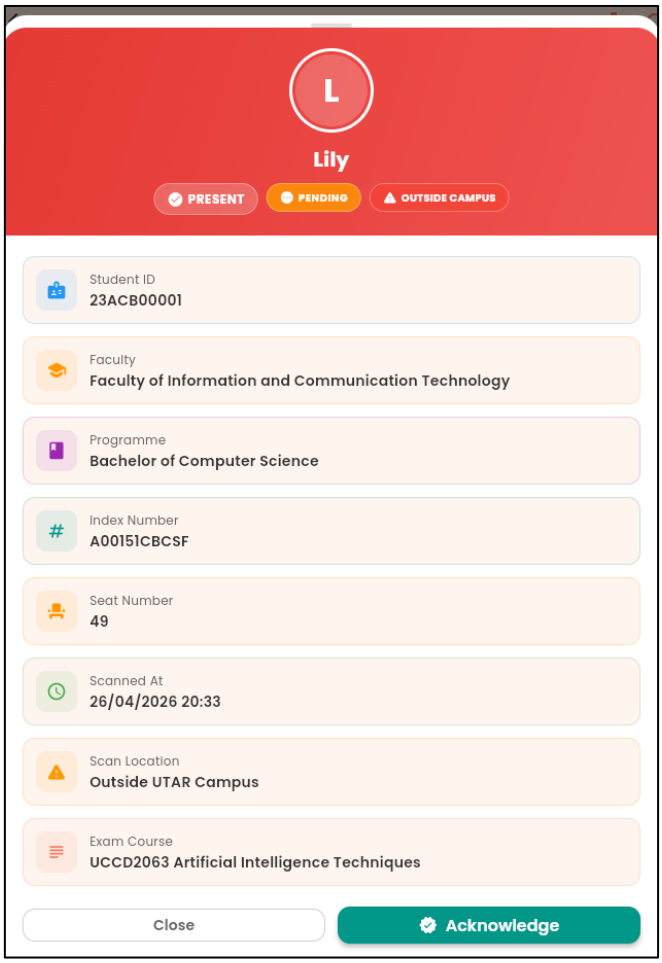

Figure 5.60 Student attendance details with warnings

5.6 Implementation Issues and Challenges

This section outlines the key issues and challenges encountered while implementing the system with the corresponding solutions adopted to overcome them.

5.6.1 Limited Custom Dataset and Domain Specificity

The custom dataset contained only 322 cropped images from UTAR exam attendance slips, which was insufficient for training a deep learning model. Adding too many public samples would cause the model to lose domain specificity, as UTAR slips have unique formats for index

numbers and course codes. To overcome this, data augmentation was applied as the main solution. The original dataset was duplicated and transformed using eight stochastic augmentations. This effectively doubled the dataset size while preserving domain characteristics.

5.6.2 Colour Variation Across Slips

UTAR exam attendance slips have a variety of background colours including white, blue, green and pink. When the same preprocessing pipeline was used in all slips, different results were obtained. For instance, a blue slip processed in the same way as a pink slip generated a binary image with a lot of noise, so extracting the text was difficult. This problem was overcome with OTSU thresholding, which dynamically calculates the threshold value for the image histogram. This method is effective regardless of the background colour, eliminating the need to set up different preprocessing stages for different slip colours.

5.6.3 Dynamic Field Cropping with EasyOCR

When a student captures an attendance slip and sends it to the API, the system needs to extract the handwritten fields from the full image. The initial approach used fixed pixel coordinates to crop the four fields from attendance slips. However, this required students to align their phone camera perfectly with the slip, which was inefficient and error-prone. To address this, EasyOCR was used to detect all text blocks on the slip, then crop each field based on spatial proximity to printed label anchors. For example, the seat number field was cropped near the “Seat No” printed label. Fallback logic was also added for cases where an anchor was not detected, ensuring the API still returned reasonable results.

Chapter 6
System Evaluation

6.1 System Testing and Performance Metrics

To ensure the reliability of the HTR model and the overall system, the evaluation was conducted across three testing components which were model evaluation, API testing, and mobile application testing, with testing strategies defined in Section 3.4.

In addition, a set of performance metrics and targets were defined to assess the system. As defined in Section 4.3.2, CER measures character-level accuracy, which is critical for index numbers where a single wrong character invalidates the entire entry. The 10% target ensures 9 out of 10 characters are correct. WER measures word-level accuracy for longer fields such as subject names. Exact Match Accuracy measures completely correct sequences. API response time was targeted at under 30 seconds to avoid long waiting during exam check-in. Table 6.1 summarises the metrics and targets.

Table 6.1 Performance metrics and targets

Metric	Description	Target
Character Error Rate (CER)	Percentage of character-level errors in recognition	≤ 10%
Word Error Rate (WER)	Percentage of word-level errors in recognition	≤ 35%
Exact Match Accuracy	Percentage of completely correct sequences	≥ 45%
API Response Time	Time taken for the API to process a request	< 30 seconds

6.2 Testing Setup and Result

This section describes the testing setup and presents the evaluation results. The findings are organised by model evaluation, API testing, and mobile application testing.

6.2.1 Model Evaluation

The trained HTR models were evaluated on an independent test set of 200 custom UTAR attendance slips, which consists of real UTAR attendance slips with diverse handwriting styles and mixed alphanumeric characters. Greedy decoding was then used to decode the predicted outputs and choose the most likely character at every time step to form the final text sequence.

1) Performance Comparison

The evaluation metrics CER, WER, and Exact Match Accuracy were computed by comparing the decoded predictions against the ground truth labels using the jiwer library. The comparative performance of the baseline model and the optimised model obtained through Optuna hyperparameter optimisation is reported in Table 6.2 and visualised in Figure 6.1.

Table 6.2 Baseline vs Optimised model performance on test set

Metric	Baseline	Optimised	Improvement
CER	11.31%	7.17%	+4.14%
WER	43.68%	31.97%	+11.71%
Exact Match Accuracy	40.00%	49.50%	+9.50%

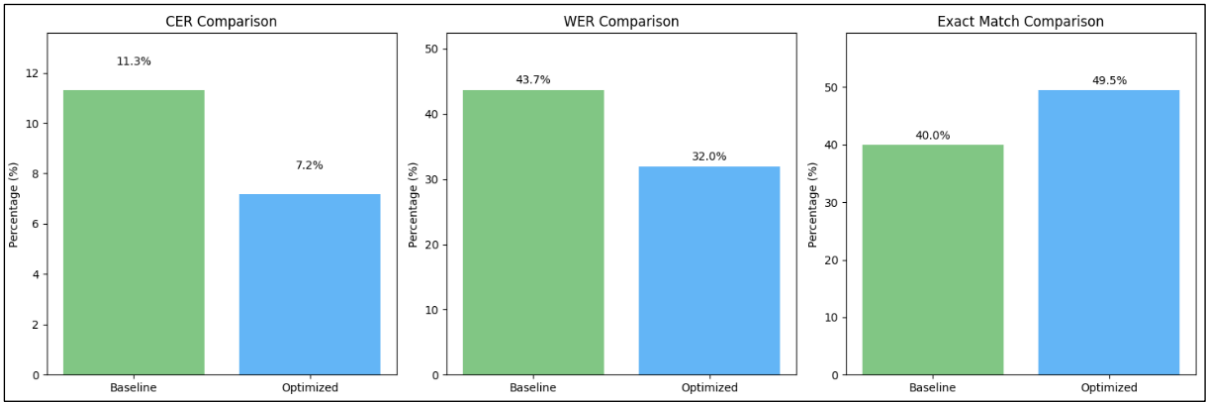

Figure 6.1 Performance comparison bar chart

The optimised model achieved a test CER of 7.17%, below the 10% target; a test WER of 31.97%, below the 35% target; an Exact Match Accuracy of 49.50%, above the 45% target. These results demonstrate that the optimised model successfully achieves all predefined performance targets. Also, the improvement over the baseline can be credited to Optuna which automatically searched the best hyperparameters over 20 trials. Compared to manual tuning, Optuna balanced exploration and exploitation of the search space, reducing CER by 4.14% and WER by 11.71%, and improving Exact Match Accuracy by 9.50%. This confirms that systematic hyperparameter optimisation enhances model performance for the HTR task.

2) Sample Predictions

Figure 6.2 shows sample predictions from the optimised model on the test set. The critical fields of attendance marking such as index numbers, seat numbers and subject names were always recognised correctly. Among the 10 sample predictions provided, only three had minor errors, each with one or two wrong characters. The remaining seven were completely correct, indicating that the model performs well on real attendance slips.

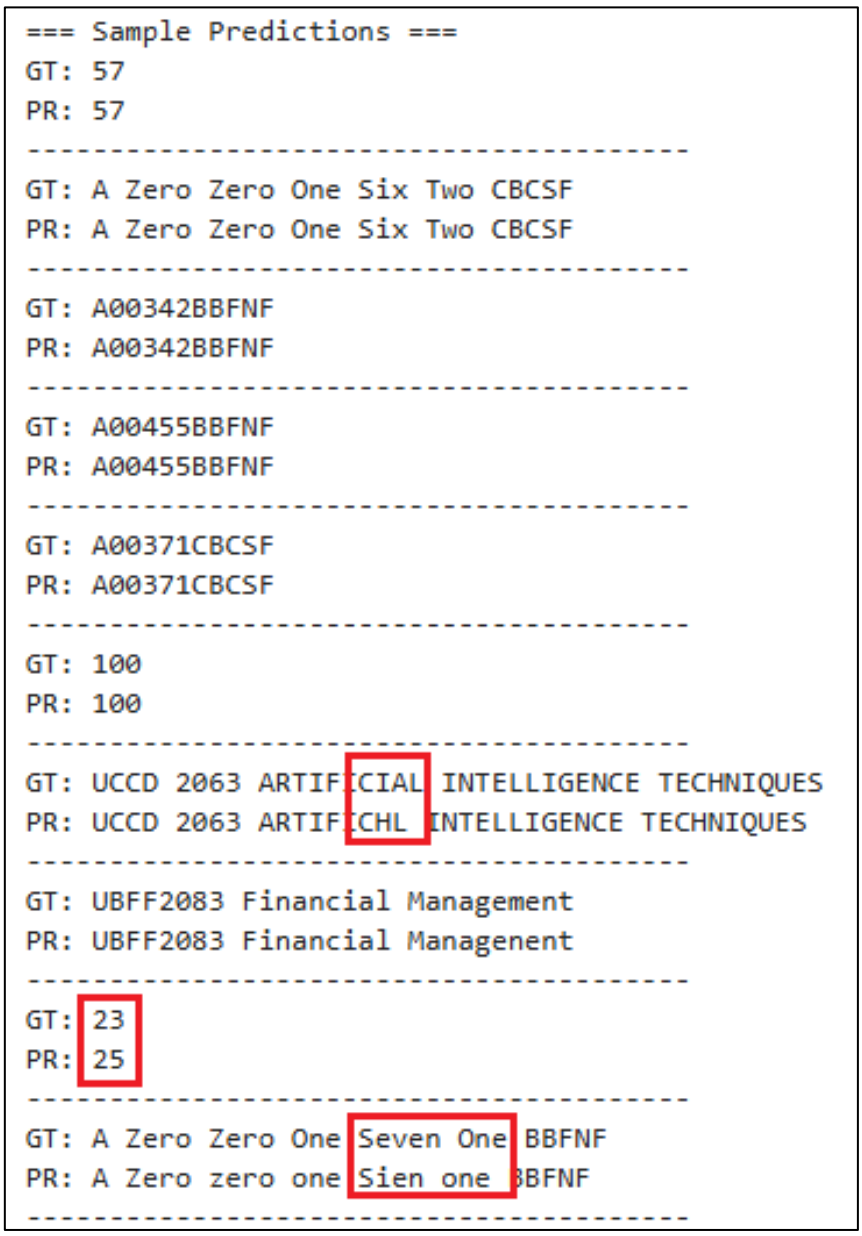

Figure 6.2 Sample predictions from optimised model

3) Confusion Matrix and Error Analysis

The confusion matrix of the optimised model on the test set is depicted in Figure 6.3. Rows show true character labels, columns show predicted labels, and the diagonal cells show correct classifications. Most of the counts are concentrated on the diagonal, indicating that the model correctly recognised most characters.

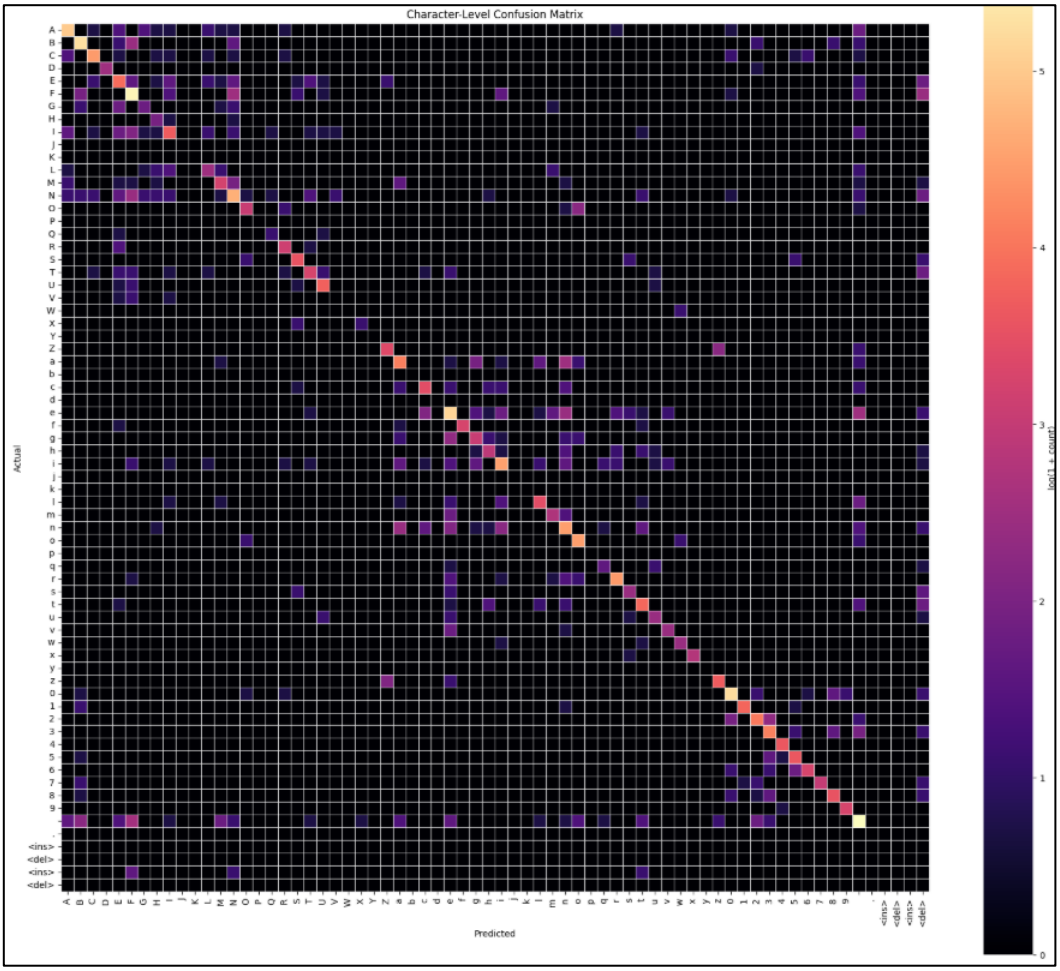

Figure 6.3 Confusion matrix of the optimised model

However, analysis of the off-diagonal cells revealed common misclassification pairs, as listed in Table 6.3. The most frequent errors occurred between visually similar characters such as “a” and “n”, “e” and “n”, and “B” and “F”. Space characters were also sometimes misclassified as “F” or “B”.

Table 6.3 Top 15 most confused character pairs

Rank	True Character	Misclassified As	Frequency
1	(space)	F	12
2	a	n	11
3	e	(space)	11
4	F	N	11
5	n	a	10
6	e	n	10
7	B	F	10
8	N	F	10
9	F	(deletion)	10
10	g	e	9
11	2	3	9
12	(space)	B	8
13	Z	z	8
14	O	o	8
15	n	i	8

Regardless of these minor errors, the evaluation results show that the proposed model manages to overcome the two key challenges, which are handwriting variability and limited character scope. The low CER of 7.17% indicates that the CNN-BiLSTM-CTC model can effectively work with a variety of handwrite styles, and the hybrid dataset allows recognizing mixed-case letters, digits and spaces needed in UTAR attendance slips.

6.2.2 API Testing

The API was subjected to a series of tests to ensure reliable communication between the mobile application and the HTR model by verifying error handling and measuring response time.

1）API Error Handling

To ensure that invalid requests are properly handled by the API, FastAPI's automatic interactive documentation, Swagger UI was used to test the API. The API test cases and results are summarised in Table 6.4.

Table 6.4 API test cases

Test Case ID	Test Case Name	Input	Expected Output	Status
TC-API-01	Valid image upload	JPG image of attendance slip	JSON with SeatNo, IndexWords, IndexFigures, Course	PASS
TC-API-02	Missing file field	POST request without file	HTTP 422 error (Field required)	PASS
TC-API-03	Invalid image format	Text file instead of image	HTTP 400 error or empty fields	PASS
TC-API-04	Empty image	Blank white image	Empty string fields for all outputs	PASS

All API test cases passed. The API successfully handled edge cases such as missing files and invalid image formats by returning appropriate HTTP status codes. Example responses are shown in Figure 6.4, Figure 6.5, and Figure 6.6.

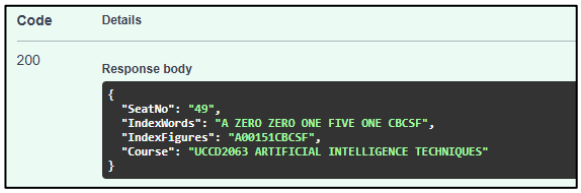

Figure 6.4 JSON response for valid image upload

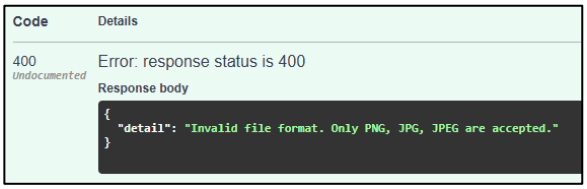

Figure 6.5 HTTP 400 error for invalid file format

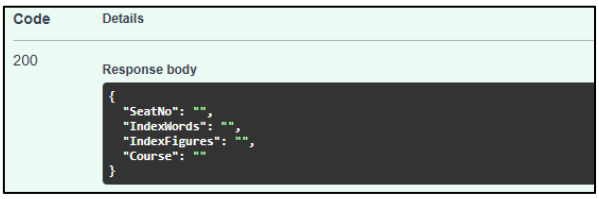

Figure 6.6 Empty field response for blank image

2）API Response Time

The API response time was measured over multiple requests using curl, as shown in Figure 6.7. Each request took between 18 and 21 seconds to complete, with an average of approximately 19.32 seconds. All responses were completed within the 30-second target, confirming that the API meets the performance requirement.

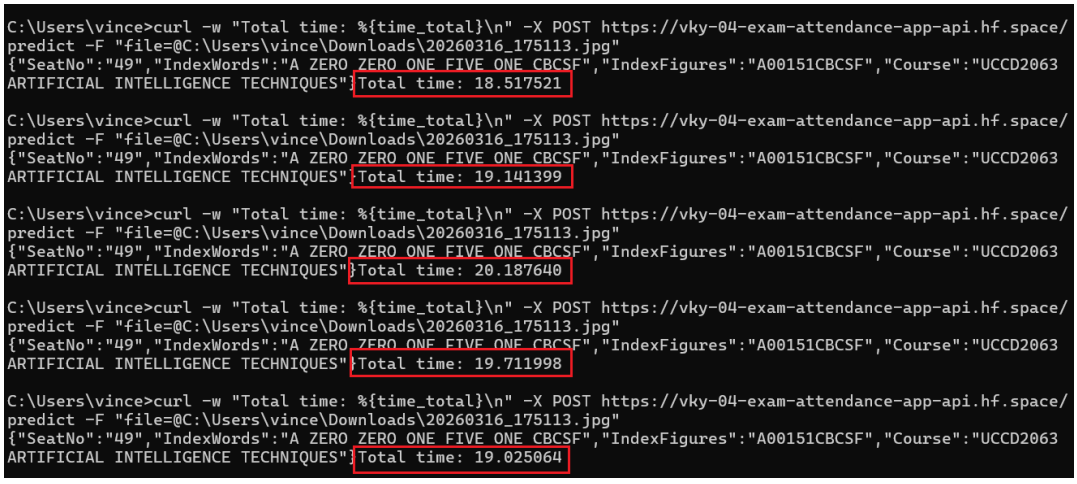

Figure 6.7 API response time measurement with curl

6.2.3 Mobile Application Testing

The mobile application was tested using compatibility testing and functional testing to ensure reliable operation across different devices and scenarios.

1) Compatibility Testing

Various Android devices with different hardware configurations and operating system versions were tested with mobile application, including Google Pixel 4 (Android 11), Samsung Galaxy A52 (Android 13), and Xiaomi Mi 9T (Android 10). Wi-Fi, 4G, and offline network conditions were also tested to ensure the application functions correctly in different situations of network conditions. The compatibility tests were all successful, indicating that the application can be utilized by students with a wide range of Android devices without any hardware or system compatibility problems.

2) Functional Testing

Each test case was designed with a main flow representing the expected successful path and an alternative flow representing error handling or edge cases. A total of 4 test cases were executed, consisting of user authentication, attendance submission in various conditions and validation schemes, offline functionality, and profile management. All test cases were passed, as shown in Table 6.5, Table 6.6, Table 6.7, Table 6.8 and Table 6.9, which demonstrates the stability of the application both under normal and failure conditions such as invalid form entries, duplicate form submissions and network failures.

Table 6.5 Test case: Student login

Test Case Name: Login			ID: TC-APP-01	
			Pre-conditions: User is registered.	
Main Flow	Test Steps		Expected Result	Status
	1	Enter registered email and password.	System accepts the input values	PASS
	2	Click “Sign In”	System displays home screen	PASS

Table 6.6 Test case: Account registration

Test Case Name: Account registration			ID: TC-APP-02	
			Pre-conditions: User is not registered.	
Main Flow	Test Steps		Expected Result	Status
	1	Navigate to registration page	System displays registration screen	PASS
	2	Enter full name, student ID, email, and password	System accepts the input values	PASS

	3	Click “Create Account”	System displays "Account registered successfully" message and redirects to login screen	PASS
Alternative Flow	2a	Enter an existing student ID	System shows “Student ID already registered”	PASS
	2b	Enter an invalid email format	System shows “Enter a valid email”	PASS
	2c	Enter an existing email	System shows “Email already registered”	PASS
	2d	Enter passwords that are less than 6 characters	System shows “Min 6 characters”	PASS
	2e	Enter mismatched passwords	System shows “Passwords do not match”	PASS

Table 6.7 Test case: Attendance submission

Test Case Name: Attendance submission			ID: TC-APP-03	
			Pre-conditions: Camera permission granted. GPS enabled.	
Main Flow	Test Steps		Expected Result	Status
	1	Navigate to scan page	System displays scan page	PASS
	2	Click “Take Photo”	System displays camera interface	PASS
	3	Capture photo of attendance slip	System displays confirmation dialog after processing	PASS
	4	Confirm and Submit attendance	System displays success dialog with notification and no warning.	PASS
Alternative Flow	4a	Submit attendance with mismatch Index number (words vs figures)	System shows validation error dialog. Blocks submission.	PASS
	4b	Submit attendance with mismatch Index number (scanned vs registered)	System shows validation error dialog. Blocks submission.	PASS
	4c	Submit attendance with mismatch Subject name	System shows exam not found error dialog. Blocks submission.	PASS
	4d	Submit attendance for a not enrolled subject	System shows "You are not enrolled in this exam" error dialog. Blocks submission.	PASS
	4e	Submit attendance for an already marked subject	System shows "You have already marked attendance for this exam" message. No duplicate record created.	PASS
	4f	Submit attendance at outside campus	System displays success dialog with notification. Shows "Outside UTAR Campus" warning flag for administrative review.	PASS

Table 6.8 Test case: Offline app launch

Test Case Name: Offline app launch			ID: TC-APP-04	
			Pre-conditions: Internet disconnected. User previously logged in. Cached exam schedule and profile data exist.	
Main Flow	Test Steps		Expected Result	Status
	1	Disable internet connection	System shows device is offline	PASS
	2	Open application	System displays home screen using cached data	PASS
	3	View exam schedule	System displays cached exam schedule	PASS

	4	View profile	System displays cached profile information	PASS

Table 6.9 Test case: Offline profile edit

Test Case Name: Offline profile edit			ID: TC-APP-05 Pre-conditions: Internet disconnected. User logged in.	
Main Flow	Test Steps		Expected Result	Status
	1	Disable internet connection	System shows device is offline	PASS
	2	Go to Profile page	System displays profile page with cached data	PASS
	3	Edit name or index number	System accepts the changes	PASS
	4	Save changes	System stores changes in sync_queue. System shows no error message.	PASS
	5	Re-enable internet connection	System processes sync_queue automatically. System syncs changes to Firestore.	PASS

6.3 Project Challenges

Several project-level challenges were encountered during system development, affecting the system's design, deployment performance, and testing validation.

6.3.1 Hardware and Infrastructure Constraints

To deploy the model, the API was hosted on Hugging Face Spaces on the free tier which is limited to approximately 512MB RAM, leading to a response time of 18 to 21 seconds per request. Additionally, the free tier has cold start times of about 1 minute, and only supports a single container instance, so concurrent requests would be processed sequentially. This would create significant delays in case more than one student submitted attendance at a time during an exam.

6.3.2 Offline Attendance Scanning Limitation

Although the system supports offline caching for exam schedules and user profiles, the attendance scanning feature requires an active internet connection. This is because the HTR model contains 5.01 million parameters, making it too large to be deployed on mobile devices. Older or low-end Android phones may experience out-of-memory errors or excessively slow inference times if the model were embedded locally. Therefore, the model was deployed as a cloud API, which inevitably requires a stable network connection. In offline cases, the students

cannot scan attendance slips by themselves, requiring administrators to manually mark attendance as a fallback.

6.3.3 Limited Availability of User Feedback

Since the system was designed for university examination settings, real user testing had to be arranged in collaboration with students and administrators, which was difficult to schedule outside of exam periods. Although testing was conducted successfully in a controlled setting, the system's performance under real examination conditions could not be fully tested to identify more potential problems such as unexpected user behaviours or network fluctuations that may occur when the system is actually used.

6.4 Objectives Evaluation

The successful completion of this project marks the realisation of all three core objectives. By transitioning from framework design to a deployed mobile solution, this work demonstrates the practical viability of deep learning for automated university administration.

The first objective was to propose a CNN-BiLSTM-CTC-based HTR framework capable of recognising full-line handwritten fields from attendance slips, addressing the core challenges of handwriting variability and limited character scope. This was successfully achieved through a model architecture comprising four convolutional layers for feature extraction, two bidirectional LSTM layers for sequence modelling, and a CTC loss function for alignment-free recognition. To ensure domain-specific recognition, the model learned from a hybrid collection of public handwriting data and real UTAR attendance slips.

The second objective was to implement a trainable HTR model with API deployment, which was fully achieved. The model was built using TensorFlow and trained on a hybrid dataset. After hyperparameter optimisation with Optuna, the final optimised model contained 5.01 million parameters. To make the model usable in practice, it was deployed as a RESTful API with FastAPI on Hugging Face Spaces, where images could be uploaded, and the extracted fields were returned in the form of a JSON. This API was then combined with Android mobile application to be used by students. This demonstrates that the proposed model is both workable and feasible for real-world deployment.

The third objective was to evaluate the HTR model accuracy and practical usability, and this objective was achieved. Quantitatively, the model had a test CER of 7.17%, a WER of 31.97% and an Exact Match Accuracy of 49.50%, all meeting predefined targets. Qualitatively, sample predictions indicated that 7 among 10 test samples were entirely right, with the other three containing only minor errors. These findings validate that the HTR model performs domain-specific accuracy on UTAR attendance slips. Moreover, the HTR model has been successfully implemented as an API with reasonable response time and even compiled with an Android app, which proves that it can be used in practice not only in the training environment.

Chapter 7
Conclusion and Recommendation

7.1 Conclusion

This project successfully developed an automated attendance marking system for university examinations using deep learning-based HTR technology. The project overcame two challenges, namely handwriting variability and limited character scope. Variability in handwriting was addressed using a CNN-BiLSTM-CTC model, with CNNs capturing spatial features, BiLSTM capturing temporal features and CTC allowing alignment-free recognition. The problem of limited character scope was tackled through the creation of a hybrid dataset, which combined publicly available handwriting datasets and real UTAR exam attendance slips, enabling recognition of a broad range of alphanumeric characters.

Through data preparation, model training, and hyperparameter optimisation using Optuna across 20 trials, the optimised model was evaluated on the test set of 200 unseen UTAR attendance slips. It achieved a test CER of 7.17%, compared to 11.31% for the baseline, along with a test WER of 31.97% and an Exact Match Accuracy of 49.50%. These results meet all predefined performance targets and confirm that the proposed model is both accurate and domain-specific for UTAR examination attendance.

To make the HTR model practically usable, an API, database, mobile application, and web platform were developed. The API was deployed in Hugging Face Spaces with FastAPI, the database was created using Firebase Firestore and SharedPreferences, both mobile and website applications were built with Flutter. This integration led to a full-fledged system for students to scan attendance slips and for administrators to manage attendance records, showcasing the integration of an HTR model into a practical educational environment.

The main contribution of this project is advancing the digitalisation of educational administration, providing a scalable framework that can be extended to other areas such as assignment checking, student registration, and record keeping. More broadly, the project demonstrates that HTR technology can effectively reduce manual workload and improve accuracy in administrative processes.

7.2 Recommendation

This project demonstrated feasibility, yet weaknesses persist in cloud optimisation, edge deployment, and user studies. Future research will focus on addressing these issues through the directions recommended.

7.2.1 Deployment Optimisation

The API was deployed on the Hugging Face Spaces free tier, which had limited memory and did not persist the model between requests under certain configurations [22]. Migrating to a paid tier with persistent memory or an alternative platform such as Google Cloud Run or AWS ECS would improve reliability and support concurrent requests more efficiently. Giannini [23] showed that serverless platforms like Google Cloud Run could achieve container startup latencies as low as 475 milliseconds with proper configuration, and Liao et al. [24] confirmed that such platforms outperform server-based alternatives in latency and throughput.

7.2.2 Model Optimisation for Edge Deployment

In the long term, to completely eliminate network dependency, model compression techniques can be explored. To enable offline attendance scanning, the model size must be reduced so that it can run directly on mobile devices without a cloud API. Frickenstein et al. [25] demonstrated that pruning can significantly reduce model size while retaining accuracy. Some lightweight architectures such as MobileOne can offer efficient alternatives for on-device scenarios [26]. Once the model fits on-device, the dependency on a stable internet connection would be eliminated, allowing students to scan slips even in offline exam halls.

7.2.3 User Study and Platform Extension

To deal with the lack of user feedback, a formal user study involving both students and invigilators would be carried out across a series of real exam sessions. Getting qualitative feedback on usability, satisfaction and error management would assist in identifying issues that are not expected. Moreover, extending the mobile application to iOS would allow a broader range of participants, further enriching the feedback. Flutter natively supports cross platform development for Android and iOS, making this extension feasible [26]. Recent versions have also improved iOS compatibility with Impeller rendering and latest SDK features [27].

REFERENCES

[1] Q. Zhang, F. Liu, and W. Song, “IMTLM-Net: improved multi-task transformer based on localization mechanism network for handwritten English text recognition,” *Complex & Intelligent Systems*, vol. 11, no. 1, Jan. 2025, doi: 10.1007/s40747-024-01713-8.

[2] H. A. Alhamad *et al.*, “Handwritten Recognition Techniques: A Comprehensive review,” *Symmetry*, vol. 16, no. 6, p. 681, Jun. 2024, doi: 10.3390/sym16060681.

[3] G. Retsinas, G. Sfikas, B. Gatos, and C. Nikou, “Best practices for a handwritten text recognition system,” *arXiv preprint arXiv:2404.11339*, Apr. 2024.

[4] Y. Wang, H. Cao, K. Chen, D. Zhao, and T. Yang, “Handwritten English recognition based on CNN and CNN-head transformer,” in *IEEE Xplore*, 2024, pp. 1857–1862, doi: 10.1109/icemce64157.2024.10862686.

[5] S. Chauhan, S. Mahmood, and T. Poongodi, “Handwritten digit recognition using deep neural networks,” in *IEEE Xplore*, 2023, pp. 1–6, doi: 10.1109/iciem59379.2023.10167391.

[6] A. A. Rangari, S. Das, and R. D, “Cursive handwriting recognition using CNN with VGG-16,” in *IEEE Xplore*, 2023, pp. 1–6, doi: 10.1109/iceconf57129.2023.10083561.

[7] C. P. Jetlin, G. Mahalakshmi, D. Divya, and P. Babu, “Handwritten Recognition Using Hybrid CNN-SVM Model for Cerebral Palsy,” *SSRN Electronic Journal*, 2025, doi: 10.2139/ssrn.5086728.

[8] E. Mas-Candela and J. Calvo-Zaragoza, “Exploring recursive neural networks for compact handwritten text recognition models,” *International Journal on Document Analysis and Recognition (IJDAR)*, Jun. 2024, doi: 10.1007/s10032-024-00481-y.

[9] A. Ansari, B. Kaur, M. Rakhra, A. Singh, and D. Singh, “Handwritten text recognition using deep learning algorithms,” in *IEEE Xplore*, 2022, pp. 1–6, doi: 10.1109/aist55798.2022.10065348.

[10] A. Goswami, “IAM_dataset_modified,” *Kaggle*, 2020. [Online]. Available: https://www.kaggle.com/datasets/ashish2001/iam-dataset-modified

[11] AbhinavPrasoon141, "multi_digit_mnist," *Kaggle*, 2023. [Online]. Available: https://www.kaggle.com/datasets/abhinavprasoon141/multi-digit-mnist

[12] M. Lourens, R. Raman, P. Vanitha, R. Singh, G. Manoharan, and M. Tiwari, "Agile Technology and Artificial Intelligent Systems in Business Development," in *IEEE Xplore*, 2022, doi: 10.1109/ic3i56241.2022.10073410.

[13] M. Abadi et al., "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *arXiv preprint arXiv:1603.04467*, Jan. 2016.

[14] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[15] C. R. Harris et al., "Array Programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, doi: 10.1038/s41586-020-2649-2.

[16] W. McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, 2010, vol. 445, doi: 10.25080/majora-92bf1922-00a.

[17] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A Next-generation Hyperparameter Optimization Framework," *arXiv preprint arXiv:1907.10902*, Jul. 2019.

[18] B. K. Pattanayak, A. K. Biswal, S. R. Laha, S. Pattnaik, B. B. Dash, and S. S. Patra, "A Novel Technique for Handwritten Text Recognition Using Easy OCR," in *IEEE Xplore*, 2023, pp. 1115–1119, doi: 10.1109/icssas57918.2023.10331704.

[19] A. P. Morris, V. Maier, and P. D. Green, "From WER and RIL to MER and WIL: improved evaluation measures for connected speech recognition," in *Interspeech*, 2004, doi: 10.21437/interspeech.2004-668.

[20] TensorFlow, "tf.keras.optimizers.Adam," *TensorFlow API Documentation*. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam?version=nightly

[21] R. I. Hastoro, A. P. Widodo, and I. Waspada, "A Hybrid Snapshot Architecture Using Firestore and Google Cloud Storage for Medical Reporting Optimization," in *2025 8th International Conference on Informatics and Computational Sciences (ICICoS)*, Oct. 2025, pp. 352–357, doi: 10.1109/icicos68590.2025.11329614.

[22] Hugging Face, "Disk usage on Spaces," *Hugging Face Hub Documentation*. [Online]. Available: https://huggingface.co/docs/hub/main/spaces-storage

[23] S. Giannini, "New startup CPU boost improves cold starts in Cloud Run, Cloud Functions," *Google Cloud Blog*, Sep. 27, 2022. [Online].

Available: https://cloud.google.com/blog/products/serverless/announcing-startup-cpu-boost-for-cloud-run--cloud-functions

[24] G. Liao, A. Deshpande, and D. J. Abadi, “Flock: A Low-Cost Streaming Query Engine on FaaS Platforms,” *arXiv preprint arXiv:2312.16735*, Dec. 2023.

[25] L. Frickenstein, P. Mori, S. B. Sampath, M. Thoma, N. Fasfous, M. R. Vemparala, A. Frickenstein, C. Unger, C. Passerone, and W. Stechele, “Pruning as a binarization technique,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2024.

[26] P. Yin, J. Lyu, S. Zhang, S. Hu, and S. Zhang, “MobileOne: An improved one millisecond mobile backbone,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023, pp. 7907–7916.

[27] Flutter, "Build for and integrate with multiple platforms," *Flutter Documentation*, Dec. 2025. [Online]. Available: https://docs.flutter.dev/platform-integration.

[28] Flutter, "Flutter 2025 Product Roadmap," *Flutter Documentation*, 2025. [Online]. Available: https://github.com/flutter/flutter/blob/master/docs/roadmap/Roadmap.md.

APPENDIX A: Source Code

A.1 Data Preprocessing Pipeline

The preprocess() function implements a robust pipeline to normalize handwritten text images. It includes grayscale conversion, Gaussian blur, CLAHE contrast enhancement, OTSU binarization, morphological opening and closing, median filtering, small component removal, margin addition, and aspect-ratio-preserving resizing to 32 pixels in height.

```
def preprocess(self, img_path: str) -> Optional[np.ndarray]:
    """Unified preprocessing function for Custom Dataset and IAM Dataset"""
    # 1. Grayscale conversion
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
    if img is None:
        return None

    target_height = self.config.get("target_height", 32)

    # 2. Gaussian blur to reduce noise
    img = cv2.GaussianBlur(img, (3, 3), 0)

    # 3. CLAHE contrast enhancement
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
    img = clahe.apply(img)

    # 4. OTSU binarization
    _, binary = cv2.threshold(img, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)

    # 5. Morphological operations
    kernel = np.ones((2, 2), np.uint8)
    binary = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel)
    binary = cv2.morphologyEx(binary, cv2.MORPH_CLOSE, kernel)

    # 6. Median filter
    binary = cv2.medianBlur(binary, 3)

    # 7. Remove small components
    binary = self._remove_small_components(binary, min_area=15)

    # 8. Add margin
    binary = self.add_margin(binary)

    # 9. Resize to target height (preserving aspect ratio)
    h, w = binary.shape
    new_w = max(1, int(w * (target_height / h)))
    resized = cv2.resize(binary, (new_w, target_height), interpolation=cv2.INTER_LINEAR)

    return resized
```

A.2 Data Augmentation

The random_augment() function applies eight stochastic transformations (Gaussian blur, brightness/contrast adjustment, rotation, shift, scaling, morphological operations, and random noise dots) to improve model robustness against handwriting variability. Each transformation is applied with a calibrated probability.

```
def random_augment(self, image: np.ndarray) -> np.ndarray:
    """Apply random augmentations based on configuration"""
    if image.dtype != np.uint8:
        image = (image * 255).astype(np.uint8)

    h, w = image.shape

    # 1. Gaussian Blur
    if random.random() < self.params.get("blur_prob", 0.3):
        image = cv2.GaussianBlur(image, (3, 3), 0)

    # 2. Brightness Adjustment
    if random.random() < self.params.get("brightness_prob", 0.3):
        brightness_range = self.params.get("brightness_range", (0.7, 1.3))
        factor = random.uniform(brightness_range[0], brightness_range[1])
        image = np.clip(image.astype(np.float32) * factor, 0, 255).astype(np.uint8)

    # 3. Contrast Adjustment
    if random.random() < self.params.get("contrast_prob", 0.3):
        alpha_range = self.params.get("contrast_alpha_range", (0.8, 1.2))
        beta_range = self.params.get("contrast_beta_range", (-20, 20))
        alpha = random.uniform(alpha_range[0], alpha_range[1])
        beta = random.randint(beta_range[0], beta_range[1])
        image = cv2.convertScaleAbs(image, alpha=alpha, beta=beta)

    # 4. Random Rotation
    if random.random() < self.params.get("rotation_prob", 0.3):
        angle_range = self.params.get("rotation_angle_range", (-3, 3))
        angle = random.uniform(angle_range[0], angle_range[1])
        M = cv2.getRotationMatrix2D((w // 2, h // 2), angle, 1)
        image = cv2.warpAffine(image, M, (w, h), borderValue=255)

    # 5. Horizontal Shift
    if random.random() < self.params.get("shift_prob", 0.3):
        shift_range = self.params.get("shift_range", (-10, 10))
        shift = random.randint(shift_range[0], shift_range[1])
        M = np.float32([[1, 0, shift], [0, 1, 0]])
        image = cv2.warpAffine(image, M, (w, h), borderValue=255)

    # 6. Scaling
    if random.random() < self.params.get("scale_prob", 0.2):
        scale_range = self.params.get("scale_range", (1.1, 1.3))
        scale = random.uniform(scale_range[0], scale_range[1])
        new_w = int(w * scale)
        image = cv2.resize(image, (new_w, h), interpolation=cv2.INTER_LINEAR)

    # 7. Erosion or Dilation
    if random.random() < self.params.get("morph_prob", 0.2):
        kernel = np.ones((1, 2), np.uint8)
```

```
        if random.random() < 0.5:
            image = cv2.erode(image, kernel, iterations=1)
        else:
            image = cv2.dilate(image, kernel, iterations=1)

    # 8. Add Random Noise Dots
    if random.random() < self.params.get("noise_prob", 0.1):
        dots_range = self.params.get("noise_dots_range", (5, 20))
        num_dots = random.randint(dots_range[0], dots_range[1])
        for _ in range(num_dots):
            x = random.randint(0, w - 1)
            y = random.randint(0, h - 1)
            image[y, x] = random.choice([0, 255])

    return image
```

A.3 Model Architecture

The _build_base_model() function constructs the CNN-BiLSTM-CTC architecture using Keras Functional API. It consists of four convolutional blocks for feature extraction, followed by two bidirectional LSTM layers for sequence modeling, and a TimeDistributed Dense layer with softmax activation for character probability output.

```
# Build the base CNN-BiLSTM-CTC model using Keras Functional API
    def _build_base_model(self) -> tf.keras.Model:
        input_img = Input(shape=self.input_shape, name='image_input')

        # === CNN Feature Extraction ===
        # Block 1
        x = Conv2D(32, (3, 3), padding='same', kernel_regularizer=regularizers.l2(0.001))(input_img)
        x = BatchNormalization()(x)
        x = ReLU()(x)
        x = MaxPooling2D(pool_size=(2, 2))(x)

        # Block 2
        x = Conv2D(64, (3, 3), padding='same')(x)
        x = BatchNormalization()(x)
        x = ReLU()(x)
        x = MaxPooling2D(pool_size=(2, 1))(x)
        x = Dropout(0.7)(x)

        # Block 3
        x = Conv2D(128, (3, 3), padding='same')(x)
        x = BatchNormalization()(x)
        x = ReLU()(x)
        x = MaxPooling2D(pool_size=(2, 1))(x)

        # Block 4
        x = Conv2D(256, (3, 3), padding='same')(x)
        x = BatchNormalization()(x)
        x = ReLU()(x)
        x = Dropout(0.7)(x)

        # === Reshape for RNN ===
        # Change dimension order: width → time steps
        x = Permute((2, 1, 3))(x)
        shape_after_permute = x.shape
        x = Reshape(target_shape=(shape_after_permute[1], shape_after_permute[2] *
shape_after_permute[3]))(x)

        # === RNN (BiLSTM) ===
        x = Bidirectional(LSTM(128, return_sequences=True, kernel_regularizer=regularizers.l2(0.001)))(x)
        x = Dropout(0.6)(x)
        x = Bidirectional(LSTM(128, return_sequences=True, kernel_regularizer=regularizers.l2(0.001)))(x)
        x = Dropout(0.6)(x)

        # === Output Layer ===
        output = TimeDistributed(Dense(self.num_classes, activation='softmax'), name='char_probs')(x)

        base_model = models.Model(inputs=input_img, outputs=output)
        return base_model
```

A.4 Hyperparameter Optimisation

The objective() function defines the search space for Optuna hyperparameter optimisation. It tunes learning rate, batch size, optimizer type, CNN filters, kernel sizes, CNN/RNN dropout rates, LSTM units, L2 regularization, and early stopping/reduce LR patience parameters.

```
def objective(self, trial: optuna.Trial) -> float:
    """Objective function for Optuna hyperparameter optimisation"""

    # Define search space using Optuna's suggestion methods
    learning_rate = trial.suggest_float('learning_rate', 1e-5, 1e-2, log=True)
    batch_size = trial.suggest_categorical('batch_size', [8, 16])
    optimizer_name = trial.suggest_categorical('optimizer', ['adam', 'nadam'])
    early_stop_patience = trial.suggest_int('early_stop_patience', 5, 12)
    reduce_lr_patience = trial.suggest_int('reduce_lr_patience', 3, 6)

    # Build model
    input_shape = (32, 640, 1)
    base_model = self._build_model(trial, input_shape)

    filters1 = trial.params.get('filters1', 32)
    filters2 = trial.params.get('filters2', 64)
    filters3 = trial.params.get('filters3', 128)
    filters4 = trial.params.get('filters4', 256)
    kernel_size1 = trial.suggest_categorical('kernel_size1', [(3,3), (5,5)])
    kernel_size2 = trial.suggest_categorical('kernel_size2', [(3,3), (5,5)])
    lstm_units = trial.params.get('lstm_units', 128)
    n_lstm_layers = trial.params.get('n_lstm_layers', 2)
    dropout_cnn = trial.params.get('dropout_cnn', 0.5)
    dropout_rnn = trial.params.get('dropout_rnn', 0.5)
    l2_reg = trial.params.get('l2_reg', 0.001)

    # Create CTC model
    ctc_model = self._create_ctc_model(base_model)

    # Select optimizer based on trial suggestion
    if optimizer_name == 'adam':
        optimizer = Adam(learning_rate=learning_rate)
    else:  # nadam
        optimizer = Nadam(learning_rate=learning_rate)

    # Compile model (CTC loss is pre-computed)
    ctc_model.compile(optimizer=optimizer, loss=lambda y_true, y_pred: y_pred)

    # Create history dict and callback
    history_dict = {"val_cer": []}
    callbacks = [
        EarlyStopping(monitor='val_loss', patience=early_stop_patience, restore_best_weights=True),
        ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=reduce_lr_patience, min_lr=1e-7),
        OptunaMetricsCallback(trial, history_dict)
    ]

    # Train model
```

POSTER


FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY
AttendScan Mobile Apps
University Student Final Exam Attendance using Deep Learning
INTRODUCTION
Manual exam attendance tracking is slow and prone to errors. This project develops a mobile application with Handwritten Text Recognition (HTR) that automatically extracts student details from exam attendance slips and marks attendance. A web platform is also available for administrators to oversee exams and records.
Powered by a CNN-BiLSTM-CTC architecture and trained on a hybrid dataset of public handwriting data and real UTAR exam slips, the app accurately recognises diverse handwriting styles and mixed-case alphanumeric text, enabling fast, reliable, and fully digitalised attendance management.
METHODS
1. Data Preparation
• UTAR attendance slips
• IAM + Multi-digit MNIST
• Preprocessing & Augmentation
2. Model Development
• CNN-BiLSTM-CTC arch
• Baseline training
• Optuna hyperparam opt
• Optimised model (5.01M params)
3. API Deployment
• FastAPI on HuggingFace
• /predict endpoint
• JSON response
4. Application Integration
Mobile App (Student)
• Scan & upload slip
• Confirm extracted info
• Offline sync support
Cloud Storage (Firebase)
Web Platform (Admin)
• Manage exams & students
• View attendance records
• Manual verification
RESULTS
MODEL PERFORMANCE
CER 7.17%
WER 31.9%
EXACT MATCH 49.50%
Prediction:
A zero zero five zero nine BBFNF
A zero zero five zero nine BBFNF
UBFF2083 FINANCIAL MANAGEMENT
UBFF2083 Financial Management
PRACTICAL USE:
Step 1 Capture & Upload
Step 2 HTR Processing
Step 3 Confirm & Submit
Step 4 Attendance Marked
SCAN
PROCESSING
ATTENDANCE RECORDED
RECORDS
DISCUSSION
• The CNN-BiLSTM-CTC architecture proves effective for recognising unconstrained handwriting on exam attendance slips.
• Optuna-based tuning reduced character errors by over 4% compared to manual configuration, demonstrating the value of automated hyperparameter optimisation for this task.
• The integration of model, API, and cross-platform applications into a single workflow confirms that deep learning-based attendance automation is practically feasible in examination settings.
CONCLUSIONS
This project delivers a fully automated attendance marking system using deep learning-based HTR. The optimised model achieves a CER of 7.17%, meeting all predefined targets, while students can mark attendance in under one minute with no manual invigilator effort. The integrated mobile and web platform provides a complete solution for university examination management.
VICKY YII SHU CHI
SUPERVISOR: MS TSEU KWAN LEE