Development of A Mobile Chatbot for Restaurant Recommendation

BY

Lew Yun Cheng

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)

FEB 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 sincere gratitude to my main supervisor, Dr. Muhammad Husaini Bin Nadri, for his continuous guidance, expertise, and invaluable support throughout the course of this project. His insightful feedback and patient mentorship have been instrumental in shaping the direction of this work and in helping me overcome the technical and academic challenges encountered during its development.

I would also like to extend my heartfelt thanks to my co-supervisor, Ms. Norliana Binti Muslim, for her constructive advice, encouragement, and dedication throughout the project. Her assistance and thoughtful suggestions greatly contributed to the refinement of the system and the overall quality of this report.

Finally, I am deeply grateful to my family and friends for their unwavering support and motivation, which sustained me throughout this journey.

ABSTRACT

Kuala Lumpur is home to thousands of restaurants, yet diners frequently face decision fatigue when choosing where to eat. Existing platforms rely on keyword search or popularity sorting and rarely account for individual taste history, halal certification, or dish level queries such as "chicken chop" or "ramen". This project aimed to design MakanBot, a mobile chatbot for restaurant recommendation in Kuala Lumpur that delivers personalised, halal aware, and explainable suggestions through natural language interaction. The system was built using a public dataset of 4,439 restaurants and approximately 78,000 reviews, which underwent five preprocessing steps including deduplication, text normalisation, cuisine inference, and halal label synchronisation. Three machine learning components were trained on the cleaned data, namely a TF IDF vectoriser, a content based ensemble classifier combining Support Vector Machine and Random Forest through soft voting, and an item based K Nearest Neighbours collaborative filter from the Surprise library. These models were embedded into a Python FastAPI backend connected to a native Android application, with intent analysis and explainable AI provided by the Google Gemini API and live metadata enriched through the Google Places API. The ensemble classifier achieved an accuracy of 90.63%, an F1 score of 94.32%, and an ROC AUC of 0.94, outperforming both individual models. Functional testing across 32 test cases recorded a 100% pass rate, and a pilot user study with 10 respondents reported 90% favourable satisfaction. The significance of this project lies in integrating hybrid recommendation, large language models, and culturally aware filtering on a mobile platform. Future work will focus on dataset expansion, multilingual support, and cloud deployment.

Area of Study: Machine Learning, Data Science
Keywords: Chatbot, Explainable AI, Halal Filter, Hybrid Recommendation, Sentiment Analysis

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

k	Number of nearest neighbours in the KNN collaborative filter
N	Sample size (number of respondents or data points)
C	Regularisation parameter of the Support Vector Machine
content_prob	Probability score from the content-based ensemble classifier
collab_pred	Normalised rating prediction from the KNN collaborative filter
fav_sim	Cosine similarity between a candidate restaurant and the user's taste vector
final_score	Final hybrid score used to rank candidate restaurants
≥	Greater than or equal to
%	Percentage

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
APK	Android Package Kit
AUC	Area Under the Curve
CSV	Comma-Separated Values
FastAPI	Fast Application Programming Interface
FYP	Final Year Project
GPU	Graphics Processing Unit
HTTPS	HyperText Transfer Protocol Secure
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
JSON	JavaScript Object Notation
KL	Kuala Lumpur
KNN	K-Nearest Neighbours
LLM	Large Language Model
MAE	Mean Absolute Error
ML	Machine Learning
NLP	Natural Language Processing
PNG	Portable Network Graphics
RAM	Random Access Memory
RF	Random Forest
RMSE	Root Mean Square Error
ROC	Receiver Operating Characteristic
SDK	Software Development Kit
SSD	Solid State Drive
SVM	Support Vector Machine
TF-IDF	Term Frequency – Inverse Document Frequency
UAT	User Acceptance Testing
UI	User Interface
URL	Uniform Resource Locator

VADER	Valence Aware Dictionary and sEntiment Reasoner
XAI	Explainable Artificial Intelligence
XML	eXtensible Markup Language

Chapter 1
Introduction

Choosing a restaurant can be a complex decision-making task, especially in urban areas like Kuala Lumpur where dining options are vast and diverse[1]. Diners often rely on online platforms filled with user-generated reviews to explore food quality, ambiance, service standards, and overall experiences. However, these reviews are typically unstructured and voluminous, requiring users to sift through large amounts of textual content to find restaurants that align with their expectations [2]. This manual process can lead to decision fatigue and delays, ultimately diminishing the dining experience.

Traditional recommendation platforms generally rely on filters, keywords, or ratings, but these approaches overlook the emotional meaning contained in written feedback. Moreover, most systems provide static lists that do not reflect user context, such as cuisine preferences or dining mood [3][4]. Traditional web-based recommendation systems often rely on filters, search keywords, and numerical ratings, which lack the depth and emotional nuance conveyed in written feedback. Moreover, these systems tend to provide static lists that are not tailored to the user's context, such as cuisine preference or dining mood [3][4]. While both mobile and web platforms can host recommendation systems, the method of delivery particularly the interaction mode plays a critical role in shaping usability and convenience.

To improve the restaurant discovery experience, conversational agents like chatbots have emerged as a more natural and interactive alternative. Unlike conventional web interfaces, chatbots simulate human-like dialogue, allowing users to express preferences in everyday language and receive dynamic responses in return. Chatbots reduce cognitive load by guiding users through structured conversations and offering curated suggestions based on input and sentiment analysis [5][6]. Their dialog-based nature enables faster and more personalized recommendation delivery, particularly within mobile environments where users prefer quick and flexible interactions [7].

This project focuses on the development of a sentiment-aware restaurant recommendation chatbot tailored to the dining landscape of Kuala Lumpur. By leveraging machine learning models.The system classifies online restaurant reviews into emotional categories like positive, negative, or neutral and combines this insight with user preferences like cuisine type and

location. The goal is to enhance dining satisfaction by offering emotionally aligned and context-relevant recommendations through an intuitive mobile chatbot interface.

1.1 Problem Statement and Motivation

In Kuala Lumpur's diverse and competitive food scene, diners increasingly consult online platforms that host large volumes of customer reviews before choosing where to eat[1]. Although customer reviews contain valuable details about food quality, ambiance, and service, they present several challenges. First, the reviews are often lengthy and unstructured, making it difficult to extract useful recommendations quickly. Second, many recommendation systems depend on ratings or simple filters, which fail to capture the tone and context in user feedback. As a result, they cannot adapt recommendations to match a diner's expectations or mood [8]. Third, the interfaces of existing platforms are largely static and form-based, requiring multiple filters and page navigation. This design is inconvenient for mobile users who prefer quick and flexible decision-making [8].

To address this limitation, this project proposes the development of a mobile chatbot that leverages machine learning and sentiment analysis to classify restaurant reviews into categories such as positive, negative and neutral and generate recommendations accordingly. The chatbot aims to simplify the decision-making process by combining sentiment trends with basic user preferences such as cuisine type and location. Unlike conventional web-based systems that often rely on rigid forms and static interfaces, a mobile chatbot enables conversational interaction that feels more intuitive and accessible, especially for users exploring options on the move. The mobile-first design also reflects current user behavior trends in Malaysia, where instant messaging platforms and chat-based interfaces are widely used for service access.

The motivation behind adopting a chatbot interface instead of a web-based model stems from its ability to deliver guided and flexible interactions without overwhelming the user. Through natural dialogue, the chatbot can interpret preferences, clarify needs, and return relevant recommendations efficiently. It eliminates the need to navigate multiple pages, manually apply filters, or read extensive text blocks. Furthermore, mobile chatbot applications are more engaging and better suited for habitual use, supporting faster discovery in scenarios where time and convenience matter most.

By integrating machine learning-driven sentiment classification with conversational interaction, this project offers a smarter and more emotionally aware recommendation experience tailored to Kuala Lumpur's urban food culture.

1.2 Objectives

The aim of this project is to design a mobile chatbot application that leverages machine learning and sentiment analysis to deliver personalized restaurant recommendations. To achieve the aim, the following objective must be achieved:

a) **To prepare user-generated restaurant review data**

   This includes cleaning, formatting, and organizing the textual data to ensure it is suitable for sentiment classification and preference extraction

b) **To develop suitable machine learning models for sentiment classification and recommendation logic**

   The models should be capable of interpreting user reviews and mapping them to relevant restaurant attributes and user preferences without requiring fixed labels or manual rules.

c) **To create a mobile chatbot interface that integrates with the backend recommendation system**

   This involves linking the machine learning module with the chatbot interface, ensuring smooth communication between frontend and backend components, and enabling users to interact conversationally to receive personalized restaurant suggestions.

1.3 Project Scope and Direction

This project focuses on developing a restaurant recommendation system based on customer review sentiments. The scope is clearly defined in the following aspects:

A) Scope of Dataset

The dataset used in this project will consist of restaurant review data prepared from online platforms that provide English-language reviews. Only English reviews will be considered to ensure consistency in text processing and sentiment analysis. Reviews containing mixed languages, other languages, or incomplete entries will be excluded during data cleaning. The dataset will include user-written reviews, ratings, basic restaurant attributes such as food type and location, and user information.

B) Scope of Location

The project is limited to restaurants located in Kuala Lumpur, Malaysia. All recommendations generated by the system will be based on restaurants within this geographic area. Data that references restaurants outside Kuala Lumpur will not be included. The system is designed to reflect dining options and customer expectations specific to Kuala Lumpur's urban environment.

C) Scope of Study

This study focuses on developing a sentiment-aware restaurant recommendation system tailored for the urban dining context of Kuala Lumpur. The scope includes designing a mobile chatbot application that delivers personalized restaurant suggestions by integrating sentiment analysis and recommendation algorithms.

The development of this project is structured into several key phases, each contributing to the creation of a personalized restaurant recommendation experience. It begins with data collection, where structured restaurant reviews and user preference inputs such as cuisine type and location are gathered. The second phase involves preprocessing this data to prepare it for analysis, including cleaning, formatting, and organizing review content. In the third phase, sentiment classification methods are applied to interpret user feedback, helping uncover emotional patterns in the reviews. These sentiment insights are then incorporated into a recommendation engine designed to suggest suitable restaurants based on both review sentiment and individual user preferences. Next, a conversational chatbot interface is developed and integrated with the backend logic to enable intuitive and interactive recommendations through natural dialogue. The final phase involves testing and evaluation, focusing on system performance, accuracy, and user satisfaction. This sequential development flow ensures the proposed solution delivers a smart and user-friendly dining assistant tailored for the local context.

1.4 Contributions

This project contributes to the field of intelligent recommendation systems by successfully completing three key objectives that collectively achieve its overall aim. First, by collecting and preprocessing restaurant review data along with basic user preference inputs, the system establishes a clean and structured foundation for extracting meaningful insights. This phase

supports a better understanding of user feedback and ensures that the input data accurately reflects local dining characteristics in Kuala Lumpur.

Second, by applying sentiment classification techniques, the system transforms textual reviews into sentiment categories like positive, neutral, or negative. This enables the model to interpret user experiences more deeply, going beyond traditional numeric ratings to capture emotional context. The outcome enhances the personalization of restaurant recommendations by aligning suggestions with sentiment trends found in customer feedback.

Third, the integration of a chatbot interface allows users to receive personalized dining suggestions through conversational interaction. The connection between the system's logic and the chatbot front-end ensures a responsive and intuitive recommendation experience, where users can express preferences naturally and obtain contextually relevant guidance with ease.

Together, these outcomes demonstrate the practical value of combining user preferences and sentiment insights within an interactive mobile environment. The system not only improves restaurant discovery for users but also offers restaurant operators a window into public sentiment. This project showcases how AI-driven review interpretation and conversational design can transform the dining decision process, making it more emotionally aware, personalized, and user-friendly.

1.5 Report Organization

This report is structured into seven comprehensive chapters that document the end to end development lifecycle of MakanBot, a mobile chatbot for restaurant recommendation in Kuala Lumpur. Chapter 1 provides the foundational framework, including the problem statement, project objectives, and project scope. Chapter 2 examines existing studies on restaurant recommendation systems, sentiment analysis techniques, and the integration of Large Language Models (LLMs) in chatbot applications. Chapter 3 outlines the development methodology used to manage the project, including the six phase workflow, dataset preprocessing steps, and the design of the hybrid recommendation engine. Chapter 4 presents the technical blueprint, detailing the system block diagram, component specifications, dataset visualisations, machine learning model construction, and performance evaluation results. Chapter 5 documents the actual development phase, covering the hardware and software setup, environment configuration, user facing screens, backend data flow, recommendation scenarios, and implementation challenges encountered during development. Chapter 6 presents both the functional chatbot testing results across 32 test cases and the pilot user study findings collected

from 10 respondents through a structured Google Form survey. Finally, Chapter 7 summarises the project's achievements, acknowledges its participation in an international conference, and proposes directions for future enhancement.

Chapter 2

Literature Review

2.1 Recommendation System

Table 2.1 illustrates the summary of recent research studies on recommendation systems, highlighting their applications, approaches, problems addressed, and key results. It provides an overview of different methods used across various domains, with a particular focus on restaurant-based recommendation systems.

Table2.1 Previous research on RS

Application	Approach	Problem	Result
Restaurant-based recommender system [9]	Hybrid filtering using sentiment analysis, Wu-Palmer semantic similarity, cosine similarity, and Jaccard similarity	Traditional recommender systems rely only on static attributes (like food quality, price, service) and fail to capture user's dynamic food preferences from reviews; also, fixed questionnaires and star ratings don't reflect real preferences accurately.	The proposed system achieved 92.8% accuracy (Wu-Palmer), 87% accuracy (cosine similarity), and 82% accuracy (Jaccard similarity), outperforming previous methods by extracting more accurate user preferences from reviews.
Restaurant recommendation system [10]	Hybrid filtering combining Content-Based Filtering (CBF) and Collaborative Filtering (CF) using the LightFM package with matrix factorization	Traditional systems relying only on ratings cannot accurately predict user preferences; problems like cold start, data sparsity, and limited personalization arise when only using CBF or CF alone.	Hybrid filtering achieved better performance (higher AUC score) compared to pure collaborative filtering. WRAP loss function performed best among the loss functions tested.
Restaurant Recommendation System using	Popularity-based model, Collaborative	Difficulty in discovering preferred	The system achieved around 70–71% testing accuracy depending on

Machine Learning Algorithms [11]	Filtering (SVD), and Machine Learning algorithms (SVM with RBF Kernel, Linear Kernel, Logistic Regression)	restaurants and dishes in unfamiliar areas, lack of personalized recommendations based on user tastes.	feature sets and training size. Derived features improved performance. RMSE was used for evaluation, and a Flask web application was developed.
Restaurant recommendation system based on sentiment analysis[12]	Uses LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) networks for sentiment analysis. Preprocessing includes tokenization, stemming, stop-word removal, and clustering of food nouns using semantic similarity (Wu-Palmer criterion). Cosine similarity helps match user preferences to restaurant menus.	Existing systems rely on static attributes like price, cuisine, or service quality and fail to consider dynamic user preferences derived from reviews.	Achieved 92.8% accuracy in the Top-5 recommendation scenario by analysing a dataset of 2990 TripAdvisor reviews. Introduced enhanced features like waiting time and menu reachability for more precise results.
Restaurant Recommendation System based on Collaborative Filtering using Machine Learning Algorithms and the Multi-Criteria Method[13]	Utilizes collaborative filtering, Analytical Hierarchy Process (AHP), and machine learning methods (SVD, NMF, Slope One). The system builds user profiles using demographic, preference, and location data, and evaluates	Existing systems tend to lack a multi-criteria approach, often limiting recommendations to single aspects like ratings or location. This system addresses the need for a broader evaluation of user preferences and restaurant attributes.	The proposed system improves the matching process by integrating multiple criteria and user feedback. SVD demonstrated the best overall performance with RMSE = 0.5777 and MAE = 0.6821.

	restaurants based on proximity and historical ratings		
A restaurant recommendation system that personalizes suggestions by analyzing online user reviews and comments[14]	Combines natural language processing (e.g., tokenization, stemming, and noun filtering with WordNet), hierarchical clustering (using the Wu-Palmer method), and sentiment analysis (SentiWordNet dictionary) to extract user preferences and recommend nearby restaurants matching those preferences.	Previous systems mostly relied on static restaurant attributes like price and service quality, ignoring dynamic user preferences extracted from reviews.	The system provided suggestions with a precision of 92.8% in the Top 5 scenario, based on evaluations using TripAdvisor user data from 2018
Collaborative filtering in recommender systems [15]	Proposes the hybrid metaheuristic KDI-KNN algorithm with perturbation-based KNN, integrating techniques like genetic algorithms (GA), particle swarm optimization (PSO), and sine cosine algorithm (SCA). Uses densest imputation and a similarity union function to enhance prediction accuracy.	Traditional collaborative filtering suffers from data sparsity, cold-start issues, and scalability challenges. Existing methods often fail to combine similarities effectively for varied datasets.	Sequential hybrid KDI-KNN with perturbation achieved the best performance. It yielded lower MAE values across multiple datasets, including 0.393 in Restaurant data and 0.7197 in Fund data, showing improved predictions compared to other algorithms.
A restaurant recommendation system that uses	Reviews are processed using Count Vectorizer	Previous systems fail to consider the sentiment in	Multinomial Naive Bayes classifier performs better than KNN for classifying

sentiment analysis for personalized suggestions based on user reviews [16]	and TF-IDF techniques. Positive reviews are identified with classification algorithms like Multinomial Naive Bayes (MNB) and K-Nearest Neighbor (KNN). Then, similar reviews are found using cosine similarity and Pearson Correlation Coefficient to recommend top-ranked restaurants.	user reviews, limiting the ability to provide personalized recommendations. The system addresses this gap by focusing on reviews and user preferences.	reviews. Cosine similarity delivers better results in identifying similar reviews compared to Pearson Correlation. Top 10 restaurants are recommended based on positive review analysis.
Comparing similarity measures to improve collaborative filtering recommender systems [17]	Conducts a theoretical review and experimental comparison of 13 similarity measures applied to standard datasets (MovieLens100k, MovieLens1M, and Jester). Tests include user-based and item-based collaborative filtering with weighted sum prediction.	Collaborative filtering depends heavily on the choice of similarity measure. There is a lack of consensus on the best measure, especially given differences in dataset density and filtering approach.	ITR and IPWR were the best similarity measures for user-based approaches, achieving lower RMSE and MAE scores on MovieLens datasets. AMI was most effective for item-based approaches across all datasets, particularly excelling on Jester (high-density data).
Health recommender systems [18]	Systematic review analysing 73 studies based on PRISMA guidelines. Focuses on recommended items classified into lifestyle, nutrition, general health care	Most health recommender systems lack consistency in reporting standards, user-centered evaluations, and transparency. Furthermore, evaluations and	Proposed five reporting guidelines to unify HRS development. Identified opportunities for integrating personalized, actionable health recommendations into user-friendly systems.

	information, and specific health conditions. Uses hybrid algorithms, collaborative filtering, content-based filtering, and context-based techniques for recommendation generation.	interfaces remain underdeveloped in many systems.	
A survey of reinforcement learning (RL) applied to recommender systems (RSs), offering insights for researchers and developers[19]	Provides an RLRS framework with four components: state representation, policy optimization, reward formulation, and environment building. Analyzes algorithms using RL and deep RL (DRL), categorized into value-based, policy gradient, and actor-critic methods. Discusses emerging trends like multi-agent systems and hierarchical RL.	Conventional recommendation approaches, such as collaborative filtering and content-based filtering struggle with sequential, dynamic interactions and long-term user engagement. Applying RL to RSs has scalability and sample efficiency challenges, especially with large state and action spaces.	The survey highlights significant advancements using DRL to address scalability issues, enabling applications in vast state-action environments. Identifies open challenges in environment simulation, explainability, and custom RL metrics.
Conversational recommender system (CHAT-REC) that leverages large language models (LLMs) like ChatGPT for interactive, explainable, and cross-domain recommendations[20]	Transforms user profiles and historical interactions into prompts for in-context learning. Augments traditional recommender systems by narrowing down candidate sets	Traditional recommender systems lack interactivity, explainability, and adaptability to new items or domains. They often require external retrievers for knowledge and suffer from	Achieved better results in Top-5 recommendations and zero-shot rating predictions compared to traditional methods. "text-davinci-003" showed significant improvement in NDCG (+11%) and RMSE (-15.86%) compared to LightGCN and Item-KNN, respectively.

	through LLMs and explaining recommendations interactively. Handles cross-domain recommendations and cold-start scenarios using external information to enrich LLMs' embeddings.	cold-start challenges.	
Utilizes graph neural networks (GNNs) to enhance recommendations by capturing high-order relationships and structured data[21]	Explores challenges and solutions in graph construction, network design, optimization, and computation. Taxonomy provided for GNNs in recommendation across four aspects: stages, scenarios, objectives, and applications.	Existing recommender systems struggle with dynamic interactions, multi-behaviour modelling, and data sparsity. Challenges in designing GNN-based systems include scalability, efficient computation, and optimal data modelling.	Provides a systematic overview of advancements in GNNs for recommender systems, highlighting unresolved issues like handling dynamic data, addressing graph depth limits, and improving the scalability of training methods
Recommender systems applied in healthcare to assist patients and professionals in making well-informed health-related decisions[22]	Covers various scenarios such as food, drug, health status prediction, physical activity, and healthcare professional recommendations. Utilizes collaborative filtering, content-based, knowledge-based, and hybrid techniques, along with machine learning approaches like classification,	Current systems often address specific diseases or recommendation contexts. Challenges include integrating diverse user profiles, ensuring secure handling of sensitive data, and providing trust, causability, and persuasive recommendations.	Provides a systematic overview of healthcare recommendation scenarios and techniques. Offers insights for future development, focusing on user profile construction, early disease detection, group decision-making, and improved system evaluation.

	clustering, and decision trees.		
Using large language models (LLMs) like GPT-3.5 for zero-shot ranking in recommender systems[23]	Explores techniques like conditional ranking prompts with historical interactions, bootstrapping to reduce position bias, and recency-focused prompting to emphasize recent behaviour. Evaluates prompts for sequential interactions and candidate ranking using natural language instructions.	LLMs struggle with understanding sequential user behaviour, position bias in candidate arrangement, and popularity bias in recommendations.	The paper highlights that LLMs show strong zero-shot ranking abilities, often outperforming traditional methods when adapted with strategies like bootstrapping. It stresses their potential as ranking models, though no fixed numerical "final result" is emphasized.
Exploring self-supervised learning (SSL) techniques to improve recommender systems by leveraging unlabelled data for enhanced representation learning[24]	Proposes a taxonomy of four main paradigms for self-supervised recommendation: contrastive, generative, predictive, and hybrid. Uses pretext tasks like contrastive signal matching, generative input reconstruction, and pseudo-label prediction to address sparsity. Open-source	Deep neural-based recommendation models struggle with sparse user-item interaction data and heavily rely on labelled datasets. The diverse data types and objectives in recommendation further complicate generalization across domains.	Provides a systematic survey of over 60 papers, categorizing methods and outlining their benefits and limitations. Introduces future research directions, emphasizing explainable SSL models, augmentation theory, and robust SSR to address challenges like cold-start, biases, and scalability.

	library "SELFRec" is also introduced for benchmarking SSR models and datasets.		
Leveraging Graph Learning (GL) to enhance recommender systems (RS) by modelling complex relationships between users, items, and attributes[25]	Reviews graph-based techniques such as Random Walks, Graph Neural Networks (GNNs), and Graph Embedding. Focuses on modelling heterogeneous and high-order interactions within data represented as graphs. Categorizes GLRS approaches into methodologies like GCN (Graph Convolutional Networks), GAT (Graph Attention Networks), and Graph Auto-Encoders.	Traditional RS approaches struggle to address issues like cold-start, sparse data, and dynamic user behaviour. Challenges include constructing effective graphs and propagating information across multi-hop neighbors in the graph structure.	Provides a comprehensive review of existing GLRS techniques, categorizing their applications and limitations. Proposes future directions like self-evolving RS, cross-domain graph learning, and explainable causal graph learning for improved recommendations.
Utilizing recommender systems (RS) to enhance tourism management, improve user satisfaction, and promote emerging destinations[26]	Conducted a systematic review using PRISMA methodology, categorizing RS into tourism management, techniques, typologies, and learning systems. Includes Recommendation models such as	Emerging destinations face challenges due to limited datasets for system training. Common RS limitations include cold-start problems, data sparsity, and lack of diversity in recommendations	Provides guidelines to address data scarcity, emphasizing the use of multiple sources (e.g., social media, tourism portals, user feedback). Highlights trends in RS development, such as integration with machine learning and contextual information to improve tourism services and destination branding.

	collaborative filtering, content-based methods, hybrid approaches, and location-based techniques RS. Discusses multi-criteria approaches for integrating contextual and user-specific data		
Hotel recommender system for tourists to match accommodations with personal priorities[27]	Combines ABC algorithm for optimized hotel searching and fuzzy TOPSIS for ranking hotels based on seven criteria from the TripAdvisor dataset (value, rooms, location, cleanliness, service, check-in/front desk, and business service).	Tourists often struggle to prioritize multiple criteria simultaneously when selecting hotels, and existing systems fail to provide fully dynamic, preference-driven recommendations.	Demonstrates the feasibility of aligning hotel recommendations with diverse user-defined priorities. The ABC algorithm is improved to provide more relevant hotel suggestions within the user's preferred range, showing enhanced performance over its standard version.
A recommendation system for food and restaurants to cater to user preferences like ratings, top sales, discounts, and weather conditions.[28]	Combines collaborative filtering and content-based filtering. Content-based filtering uses TF-IDF for feature extraction, while collaborative filtering calculates similarities using Pearson correlation coefficients. Hybrid filtering alternates between the two methods to suggest food and	Current systems in the food industry lack robust personalization features. Challenges include data sparsity, cold-start issues, and failure to consider dynamic user preferences like mood or seasonality.	The proposed hybrid filtering system achieved a success rate of 83.5%. Among tested algorithms, Random Forest demonstrated the best performance, with accuracy = 0.859 and loss = 0.1193. Recommendations effectively align with user-defined priorities and attributes like price, ratings, and location.

	restaurants. Various algorithms in machine learning, for example Decision Trees, Random Forest, and Gradient Boosting, are tested on pre-processed data		
A prompt-based restaurant recommendation system to enhance user experience by personalizing dining suggestions and streamlining operations.[29]	Utilizes NLP techniques such as regular expressions, tokenization, and cosine similarity to process user prompts. Integrated with a chatbot for interactive communication and scheduling features like table reservations, staff allocation, and delivery management using scheduling algorithms (e.g., FCFS, Round Robin, Priority-based).	Traditional restaurant search systems rely on rigid filters (e.g., price, location) that limit personalization and user convenience. Operational inefficiencies, like long wait times and unbalanced workloads, further affect customer satisfaction.	Demonstrates seamless integration of NLP-driven recommendations and chatbot functionalities. Scheduling algorithms improve restaurant operations by organizing reservations, allocating staff, and balancing delivery workloads. Future developments include LLM integration for regional recommendations
Conversational Recommender Systems (CRS)[30]	Multi-turn conversation strategy, preference elicitation, NLP integration, exploration-exploitation balancing	Static RS fail to capture dynamic user preferences; difficult to understand "why" users like items; traditional evaluation methods are insufficient	Identified five main research challenges; provided a road map to improve CRS performance through real-time, interactive conversations

Enhancing the capabilities of recommender systems using artificial intelligence (AI) techniques to address challenges in personalization and prediction.[31]	AI techniques such as fuzzy logic, transfer learning, neural networks (e.g., CNN, RNN), evolutionary algorithms, and reinforcement learning are reviewed. Applications span domains including e-commerce, social media, and e-learning.	Conventional methods face limitations like data sparsity, cold-start problems, and lack of transparency. Complex data sources, user dynamics, and varied item representations add to the challenges.	Provides a systematic review of AI applications, identifying open research questions and future directions. Highlights potential for improved recommendation methods using hybrid AI approaches and methodologies addressing long-term user engagement.
Exploring recommender systems across diverse applications such as books, movies, e-commerce, health, and tourism.[32]	Conducts a systematic review of recent methods categorized into content-based, collaborative, and hybrid filtering techniques. Evaluates datasets, simulation platforms, and performance metrics used in the research.	Challenges include cold-start, sparsity, scalability, user feedback collection, and selecting suitable algorithms for specific applications. Lack of a standardized evaluation framework complicates comparison across systems.	Provides a taxonomy for recommender system techniques and insights into current research gaps. Suggests exploring deep learning, real-time user feedback, and hybrid models as future directions.
General Recommendation Systems[33]	Content-Based Filtering, Collaborative Filtering, Hybrid Systems, cognitive-based personalization	Traditional models face cold-start, sparsity, and gray sheep issues; growing need for dynamic behaviour tracking	Systematized and categorized recommendation techniques; linked academic developments with real-world business applications (e.g., Netflix, Amazon)

Table 2.1 reviews recent studies on restaurant and general recommender systems, showing the range of approaches that have been applied. A clear pattern is the rise of hybrid models that merge content-based filtering, collaborative filtering, and sentiment analysis to capture user preferences more effectively [13][14][15][16]. Earlier systems that focused only on fixed

restaurant features such as price, service, or food quality were less capable of reflecting the evolving and complex nature of user interests, especially those expressed in reviews and social media [13][16][18].

Several studies describe the use of sentiment analysis supported by deep learning models such as LSTM and GRU [16], or machine learning methods including SVM and logistic regression [15], to adjust restaurant recommendations more closely to user behaviour. Other works introduced multi-criteria decision-making models, allowing a wider evaluation of user needs beyond basic ratings [17]. Newer research directions such as conversational recommender systems [35] and the use of large language models (LLMs) like ChatGPT [25] [28] show an effort to make recommendation processes more adaptive and conversational across different application areas.

Although notable developments have been reported, many studies continue to highlight challenges, particularly related to cold-start problems, data sparsity, and the limited ability to capture real-time changes in user interests [19][23] [38]. Recent approaches such as self-supervised learning [29], graph neural networks [26], and reinforcement learning [24] are being explored to manage these difficulties by learning from smaller datasets and better representing complex user-item interactions.

From the findings of this review, this project emphasizes developing a hybrid restaurant recommendation system that integrates collaborative filtering with content-based filtering techniques. This approach is chosen to address challenges such as cold-start and sparsity, while also allowing the system to suggest items based on both user similarity and item attributes. The system will incorporate user demographic information, historical ratings, and item features to generate recommendations. Inspired by recent studies [14][17], this project aims to develop a system that is more adaptive to user preferences by considering multiple sources of information.

2.2 Machine Learning Models

Table 2.2 illustrates various machine learning and deep learning models applied to sentiment classification and restaurant recommendation tasks, which can support the development of a hybrid recommender approach utilizing both collaborative filtering and content-based filtering methods

Table 2.2 Machine Learning Model

References	Approach	Application and Dataset	Performance Metric	Finding
[34]	Random Forest	Sentiment classification and review rating prediction using Yelp reviews (112,412 reviews from Jan–Jun 2020)	MSE = 0.896, MAE = 1.002, R^2 = 0.214, F1 = 0.866	Performs moderately well but less reliable compared to deep learning models
	Simple Embedding + Average Pooling		MSE = 0.909, MAE = 0.821, R^2 = 0.514, F1 = 0.900	Performs better than machine learning models and proves reliable for large datasets.
	Bidirectional LSTM		MSE = 0.917, MAE = 0.913, R^2 = 0.601, F1 = 0.920	Achieves the best performance among all models, highlighting its suitability for sentiment analysis and review prediction tasks.
[35]	Logistic Regression	Classification of restaurant ratings (Zomato dataset: 51,717 rows, 17 features, focused on Bangalore restaurants).	Accuracy = 59.30%	Performs poorly compared to ensemble methods like Random Forest and XGBoost.

	K-Nearest Neighbors (KNN)		Accuracy = 89.81%	Moderate performance but struggles with larger datasets or high-dimensional spaces.
	Random Forest		Accuracy = 94.90%	Strong performance and generalization capabilities, but slightly inferior to XGBoost in terms of accuracy.
	Support Vector Machine		Accuracy = 59.05%	Like Logistic Regression; performs poorly compared to ensemble models
	XGBoost		Accuracy = 98.07%	Achieves the highest accuracy among all models, demonstrating its suitability for predictive tasks in restaurant classification
[3]	SVM	Sentiment classification on Yelp halal restaurant reviews (52,233 reviews, 565 restaurants	Accuracy = 88.47%, Precision = 86.6%, Recall = 88.5%, F1-score = 87.1%	Shows strong classification performance, effective in high-dimensional text data analysis.
	Logistic Regression		Accuracy = 88.56%, Precision = 86.8%, Recall = 88.6%, F1-score = 87.2%	Outperforms other models in text classification tasks; robust and interpretable

	Random Forest (RF)		Accuracy = 82.51%, Precision = 82.9%, Recall = 82.5%, F1-score = 77.5%	Balances complexity and performance but falls short compared to Logistic Regression and SVM
[4]	Naïve Bayes (Gaussian NB, Multinomial NB)	Sentiment analysis on Kaggle's Restaurant Reviews dataset (1000 reviws).	Accuracy = 73% (Gaussian NB), 77.5% (Multinomial NB)	Naïve Bayes is efficient for text classification but assumes independence between features.
	Logistic Regression		Accuracy = 77%	Performs well in binary classification tasks but lacks interaction modeling for recommendations.
	KNN		Accuracy = 69%	Performs worst among tested models but has a role in user-based CF approaches
	SVM		Accuracy = 78%	Works well for sentiment classification but less effective for recommendation tasks
[36]	Decision Tree Classifier	Personalized food recommendation system based on user ratings in an Android restaurant app	Accuracy = 92%	Uses ratings on food quality, time, taste, service, and price to predict meal preferences.

[37]	RF	Prediction model for review helpfulness using 1,483,858 Yelp restaurant reviews.	RMSE = 2.484, MAE = 0.253	Perform less accurate than XGBoost.
	SVM		RMSE = 2.735, MAE = 1.458	Weaker predictive performance compared to ensemble methods.
	Extreme Gradient Boosting (XGBoost)		RMSE = 1.643, MAE = 0.231	Best predictive performance among tested models, robust for large datasets
[38]	SVM	Sentiment analysis on 6,394 Zomato Bangalore restaurant reviews	Accuracy = 92%	Used for sentiment classification of positive, negative, and neutral reviews.
[39]	Naïve Bayes (Multinomial, Bernoulli)	Sentiment classification using restaurant review dataset (1000 reviews).	Accuracy= 76% (Multinomial), 75% (Bernoulli	Performs efficiently but assumes feature independence. Multinomial NB achieves higher accuracy
	Logistic Regression		Accuracy= 73%	Works well for binary classification tasks but lacks interaction modeling for recommendations.

	SVM		Accuracy= 74%	Works well for sentiment classification but less effective for recommendation tasks.
	RF		Accuracy= 72%	Performs better than basic classifiers but less effective than deep learning models
	KNN		Accuracy= 72%	Simple but less effective in text classification. Works best in user-item similarity searches.
	LSTM		Accuracy= 77%	Deep learning model with memory capability, effective for understanding text sentiment.
	CNN		Accuracy= 84%	High performance in text-based sentiment classification
[40]	Logistic Regression	Sentiment analysis on 1001 restaurant reviews sourced from Kaggle.	Accuracy = 76.40%	Performs well for binary classification (positive/negative sentiment)
	SVM		Accuracy = 76.80%	Slightly outperforms Logistic Regression, effective for distinguishing sentiment categories.
[41]	SVM	Sentiment classification on restaurant	Accuracy = 88.47%, F1-score =	SVM is enhanced with Particle Swarm Optimization (PSO) to

		reviews from Jeeran (Arabic dataset).	87.1%, AUC = 0.81	improve classification accuracy.

Table 2.2 provides a summary of machine learning and deep learning methods that have been applied in sentiment analysis and restaurant recommendation research. Among the commonly evaluated models are Random Forest, Logistic Regression, SVM, KNN, XGBoost, LSTM, and CNN [12]. These techniques have been tested on datasets such as Yelp, Zomato, and Kaggle restaurant reviews, and their performance measured using metrics including accuracy, precision, recall, F1-score, RMSE, and MAE.

Based on the findings, Random Forest, SVM, and KNN consistently delivered reliable performance across multiple classification and recommendation scenarios. Although XGBoost and CNN [12] achieved strong accuracy in certain cases, this study focuses on developing a hybrid recommender that integrates collaborative filtering and content-based filtering. Consequently, Random Forest, SVM, and KNN were chosen as the primary models.

For sentiment analysis, customer review texts will be transformed into numerical feature representations using the TF-IDF method. TF-IDF highlights the most meaningful and sentiment-rich words in the reviews, which allows the chosen models to classify them as positive, negative, or neutral. These classifications, along with user ratings and preferences, will serve as inputs to the recommendation logic, helping the system generate personalized restaurant suggestions aligned with both customer opinions and stated preferences.

2.3 Comparison between Selected Models

Table 2.3 illustrates the comparison of Random Forest, SVM, and KNN models based on their strengths and weaknesses for building a hybrid restaurant recommender system.

Table 2.3 Strengths and Weaknesses of Selected Model

Models	Strengths	Weaknesses
Random Forest	1.Handles structured data well 2.Can deal with missing data and noisy inputs 3. Reduces risk of overfitting compared to single decision trees	1. Can become slow with very large datasets 2.Less interpretable compared to simpler models
Support Vector Machine (SVM)	1.Works well with high-dimensional feature spaces 2.Suitable for text classification tasks 3.Effective with small and medium-sized datasets	1.Training time can be long with very large datasets 2.Requires careful tuning of parameters (e.g., kernel choice)
K-Nearest Neighbors (KNN)	1. Very simple to implement 2.No training phase needed 3.Performs well when similar items/users exist in the dataset	1.Can become slow during prediction with large datasets 2.Sensitive to noisy data and irrelevant features

Table 2.3 compares the three selected models, which are Random Forest, SVM, and KNN based on their strengths and weaknesses. Random Forest is suitable for handling structured data and managing missing or noisy inputs, making it helpful for analysing user and item attributes in recommendation tasks. However, it may become slower when applied to large datasets and can be less straightforward to interpret compared to simpler models. SVM is appropriate for working in high-dimensional spaces, such as text classification from restaurant reviews, and performs consistently with small to medium-sized datasets. Nonetheless, it requires careful adjustment of parameters, and training time may increase with larger datasets. KNN provides a simple and direct method for identifying similar users or items, fitting well within collaborative filtering approaches. Its simplicity, however, can lead to slower predictions when handling large datasets, and it is sensitive to noisy or irrelevant features.

Based on this comparison, Random Forest, SVM, and KNN together offer a foundation for building a hybrid restaurant recommender system. Random Forest will support learning from structured data, SVM will assist in sentiment classification using user reviews, and KNN will help in capturing similarities between users or items for collaborative recommendations. Combining these models allows the system to address multiple aspects of recommendation needs, relying on both user interactions and item attributes to generate more tailored suggestions.

2.4 Existing Chatbot

2.4.1 Next-Generation Food Recommendation Chatbot

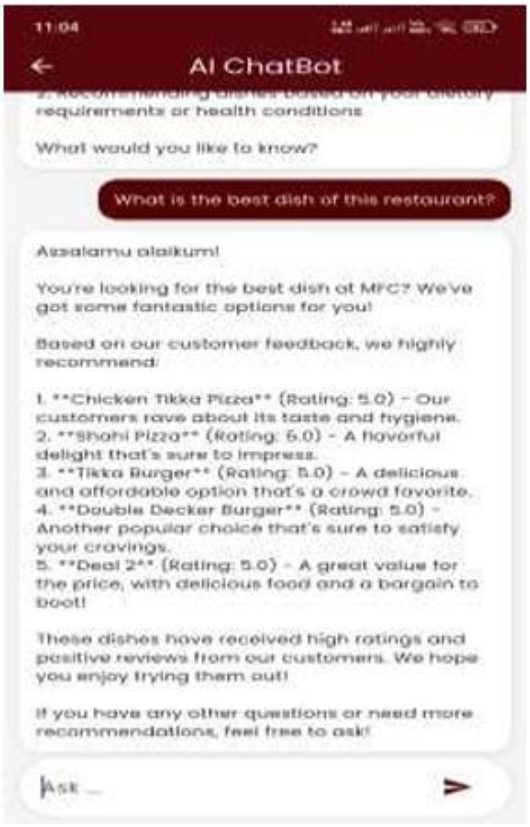


Figure 2.1 Next-Generation Food recommendations Chatbot [5]

Nawaz et al. [5] introduced a chatbot that functions as a real-time, feedback-oriented food recommendation tool by combining technologies such as Groq's high-speed inference engine, Meta's LLaMA 3-8B model, and the Firebase Realtime Database. The system is designed to dynamically interpret user dietary restrictions and preferences, offering customized dish suggestions through ingredient-level filtering, analysis of customer feedback, and recognition of emerging taste patterns. Through natural language queries, users can request options that meet specific needs (e.g., gluten-free, vegan, or diabetic-friendly), and receive instant

recommendations with detailed ingredient transparency. Unlike traditional static recommenders, this chatbot continuously adapts based on live feedback, refining its outputs to mirror user satisfaction levels and trending menu choices. The framework also integrates a Flutter-driven interface that supports conversational interaction, visual elements, and accessibility features, ensuring a smooth experience on both mobile and web platforms [5].

2.4.2 AI Chatbot-Based Restaurant Recommendation App Using Personalization Information

Figure 2.2 AI Chatbot-Based Restaurant Recommendation App Using Personalization Information [6]

Kim et al. [6] introduced an AI-powered chatbot integrated into a restaurant recommendation mobile application, which leverages diverse personalization data to improve recommendation accuracy and overall user experience.The system captures user-specific attributes such as gender, age, interests, occupation, search history, wishlist entries, real-time location, and review history to generate customized restaurant recommendations through interactive chatbot conversations. The app features multiple service modules, including real-time restaurant data retrieval, reservation systems, review writing, and even 3D spatial views of restaurant interiors. Its underlying recommendation mechanism is driven by collaborative filtering, enabling cross-recommendation of restaurants based on similarity between user profiles. The chatbot operates on a two-way dialogue model, allowing users to either inquire about restaurants or make direct reservations via natural language interaction. Unlike conventional static apps, this system emphasizes dynamic personalization by fusing AI chatbot communication with location-aware

and user-specific data, offering a more intelligent and responsive dining assistance platform [6].

2.4.3 SiLaperBot – Conversational Culinary Recommender Chatbot

Figure 2.3 Conversational Culinary Recommender Chatbot [42]

Fadhlullah et al. [42] present a mobile chatbot system called SiLaperBot, designed to recommend culinary venues based on personalized user preferences and contextual time awareness. Built as a Conversational Recommender System (CRS), it captures user input through mixed-mode interaction like text messages and button-based selections and supports dynamic feedback such as critiques and preference adjustments. The chatbot collects at least three initial preferences related to food, cuisine, facilities, or specific restaurants to form a user profile. Recommendations are generated using a graph-based Personalized PageRank algorithm that scores culinary places by linking user preferences to item attributes. SiLaperBot also includes an explanation feature that helps users understand why certain venues are recommended by referencing matched nodes in its knowledge graph. The system operates on the Telegram platform and utilizes Indonesian language processing through components such as Intent Recognizer, Entity Recognizer, Dialog Manager, and Recommendation Services. Although user satisfaction was high based on qualitative feedback, the system achieved only 40% accuracy due to limited data, highlighting the need for expanded knowledge bases and property diversity in future enhancements [42].

2.4.4 Text Rex – Human-to-Chatbot Transition System

Text Rex, originally a fully human-operated SMS restaurant recommendation service by The Infatuation, presents a compelling use case for exploring chatbot automation in high-engagement settings. Ravi and Staddon [43] investigate which types of customer inquiries are most suitable for transition to automated or hybrid chatbot services, often referred to as "humbots." The study reveals that shorter, interest-driven conversations typically involving cuisine preferences or restaurant vibe are more chatbot-amenable. In contrast, complex requests related to future event planning or interpersonal relationships, such as dining with in-laws or colleagues, tend to trigger longer conversations that challenge chatbot responsiveness. Text Rex was valued for its personal touch, frequently being compared to "texting a food-savvy friend," which raises concerns about maintaining trust and warmth in automated versions. The system also demonstrated that user tolerance for chatbots increases when speed and efficiency are prioritized, particularly for urgent or less nuanced requests. The research highlights the potential for conversational classification models to guide real-time chatbot deployment and underlines ethical design principles for transitioning human-centric services into automated frameworks [43].

2.4.5 MelakaEats – AI Chatbot for Personalized Recommendations in Melaka

Figure 2.4 MelakaEats [44]

MelakaEats is an AI-powered chatbot designed to assist users in discovering dining options across Melaka's culturally rich culinary landscape. Built using the GPT-3.5 Turbo language model and developed via Streamlit, the chatbot enables users to interact using natural language

queries related to cuisine types, dietary restrictions, budget, and location. The system incorporates personalization features through real-time conversation, recommending restaurants based on user prompts. Using the Technology Acceptance Model (TAM), the study examined user perceptions through key dimensions including usefulness, ease of use, attitudes toward the system, and the intention to adopt the technology. Responses from 35 participants indicated strong acceptance, with Cronbach's alpha values above 0.75 and mean scores exceeding 4.0 across all TAM categories. This pilot study highlights the effectiveness of MelakaEats in enhancing user experience by offering tailored recommendations and suggests future enhancements using AR/VR and deeper personalization to further enrich the dining journey [44].

2.4.6 NLP-Based Chatbot for Multiple Restaurant Services

Figure 2.5 NLP-Based Chatbot for Multiple Restaurant Services [45]

Garg et al. [45] designed an NLP-driven chatbot aimed at automating key restaurant-related tasks such as reservations, menu inquiries, takeout orders, and answering frequently asked questions. The system interprets user input using predefined intents and entities, enabling it to

guide users through reservation flows, suggest dishes, and provide business information like location and payment methods. A key focus of the chatbot is its ability to simulate conversation by recognizing user intent and selecting relevant responses trained on a structured dataset. Implemented using Python, TensorFlow, and TFLearn within a Jupyter environment, the chatbot employs a simple artificial neural network with two hidden layers to classify user queries and retrieve appropriate answers. While it shows promise in easing receptionist workload and improving customer experience, the system struggles with unseen entities and context-sensitive logic, highlighting a limitation in adaptability. The proposed solution demonstrates a foundational approach to restaurant chatbot development, emphasizing user interaction via platforms like Telegram, Facebook Messenger, and custom restaurant apps [45].

2.4.7 Muk-guide – Restaurant-Menu Curation Chatbot Prototype

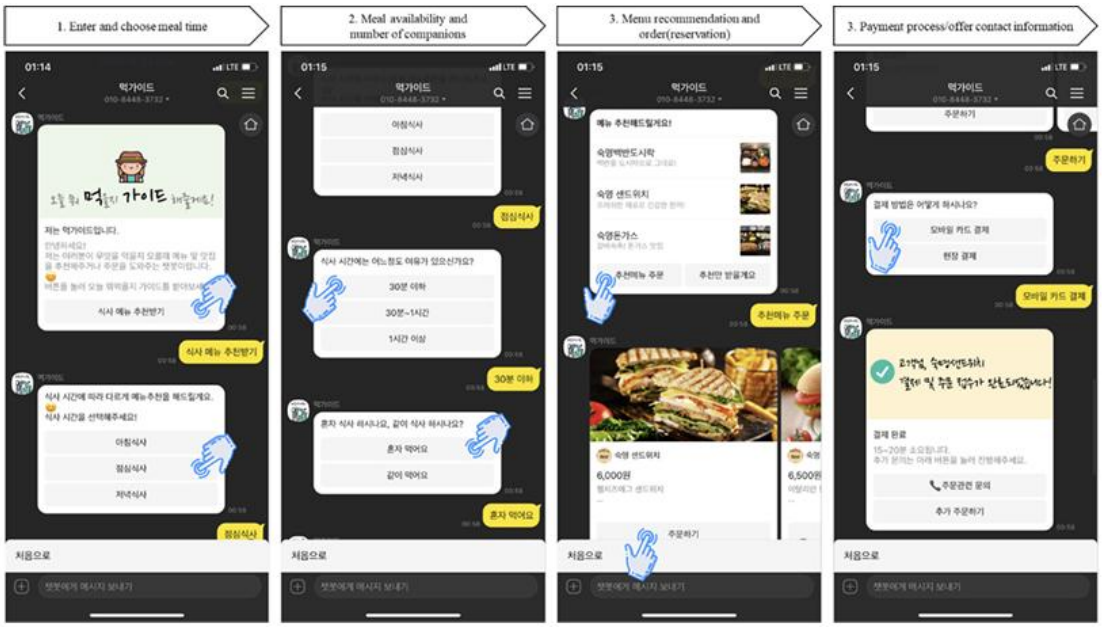


Figure 2.6 Muk-Guide [46]

Yoon and Yu [46] developed the Muk-guide chatbot, a restaurant-menu curation (RMC) system designed to assist users in selecting suitable restaurants or menus based on time, place, and occasion (TPO). Created using Kakao i Open Builder, the chatbot followed a structured scenario-based flow and featured personalized messaging, fallback handling, and a welcoming conversational persona. The system aimed to simplify the dine-out decision-making process by guiding users through contextual menus and facilitating smart choices quickly. To evaluate user acceptance, the study measured experience characteristics using Morville's UX honeycomb model, focusing on facets like usefulness, findability, value, and desirability. A

survey involving 368 participants revealed that positive experiences especially usefulness and findability significantly shaped attitudes toward the chatbot and strongly influenced utilization intention. The study emphasized that well-designed UX elements, even in a simple conversational format, contribute meaningfully to acceptance and long-term engagement. Muk-guide demonstrated the potential of curated chatbot services to reduce decision fatigue and enhance the overall dining experience by offering smart, context-aware recommendations [46].

2.4.8 Ask Vicente – WhatsApp-Based Restaurant Recommendation Chatbot

Romero-Charneco et al. [7] investigated *Ask Vicente*, a restaurant recommendation chatbot operating through WhatsApp, with the aim of identifying factors influencing users' intention to continue its use. Applying the UTAUT2 framework, the researchers conducted a survey with 386 frequent users and found that effort expectancy (EE), hedonic motivation (HM), habit (HT), and price value (PV) were significant predictors of continued adoption. Among these, EE and HM were identified as essential for shaping behavioral intention, emphasizing the need for user-friendly interactions and an enjoyable experience. Conversely, constructs such as performance expectancy, social influence, and facilitating conditions showed no significant impact on ongoing use. The integration of the chatbot into mobile instant messaging (MIM) platforms like WhatsApp enhances both accessibility and convenience, aligning well with the variety-seeking tendencies often present in restaurant choices. Moreover, the findings indicated that demographic factors, including age and gender, did not significantly moderate user intentions, reflecting widespread acceptance across different user groups. Ultimately, *Ask Vicente* demonstrates how chatbot systems can leverage familiarity with messaging platforms and habitual use patterns to encourage customer loyalty within the hospitality sector [7].

2.5 Summarisation of Features in Existing Chatbot

Based on discussions in previous sections, all features in the existing chatbot and chatbot for this project have been summarised in Table 2.4.

Table 2.4 Features Comparison between Existing Chatbot

	Restaurant Recommendation	Sentiment Analysis	API Integration	Halal Restaurant	Restaurant Detail	Reservation	Location Guidance	XAI	Mobile Platform
[5]	✓	✓						✓	
[6]	✓				✓	✓	✓		
[42]	✓	✓		✓	✓	✓		✓	✓
[43]	✓				✓				✓
[44]	✓	✓			✓				
[45]	✓	✓			✓	✓			
[46]	✓				✓	✓			✓
[7]	✓				✓				✓
This Work	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 2.4 compares the features of existing restaurant chatbots and highlights how the proposed chatbot aims to deliver a more complete and user-centered experience. While all prior systems, such as Next Generation Food Recommendation Chatbot [5], AI Chatbot Based Restaurant Recommendation App Using Personalization Information [6], and MelakaEats [44], include basic restaurant recommendation features, most lack advanced capabilities that meet broader user needs. For example, only a few systems such as MelakaEats [44] and NLP Based Chatbot for Multiple Restaurant Services [45] include halal restaurant filtering, which is especially important for users in culturally and religiously diverse regions such as Malaysia. Furthermore, explanation AI (XAI), which improves transparency by clarifying why certain recommendations are made, is only included in a few systems such as [5] and [42] even though it plays an important role in building user trust. Features such as location guidance and

reservation support are only partially implemented in some chatbots including [6], [42], [44], and [45].

In addition, none of the existing chatbots reviewed integrate modern external APIs to enhance system intelligence and data richness. The chatbot proposed in this project addresses these limitations by combining all the key features which include restaurant recommendation, sentiment analysis, halal filtering, explanation AI, location guidance, reservation functionality, and mobile deployment. Moreover, the proposed system integrates two powerful external APIs that are absent in all prior works. The Gemini API is used to power the natural-language understanding layer, enabling the chatbot to interpret dish-level queries (e.g. "chicken chop") and generate human-readable XAI explanations, while the Google Places API is used to enrich each recommendation with real-time restaurant metadata such as photos, addresses, ratings, menu links, and reservation URLs. The integration of these APIs, together with the other essential features, enables the proposed chatbot to provide a dining recommendation experience that emphasizes personalization, transparency, real-world context, and cultural relevance, thereby surpassing existing approaches.

Chapter 3

System Methodology

3.1 Proposed Method/Approach

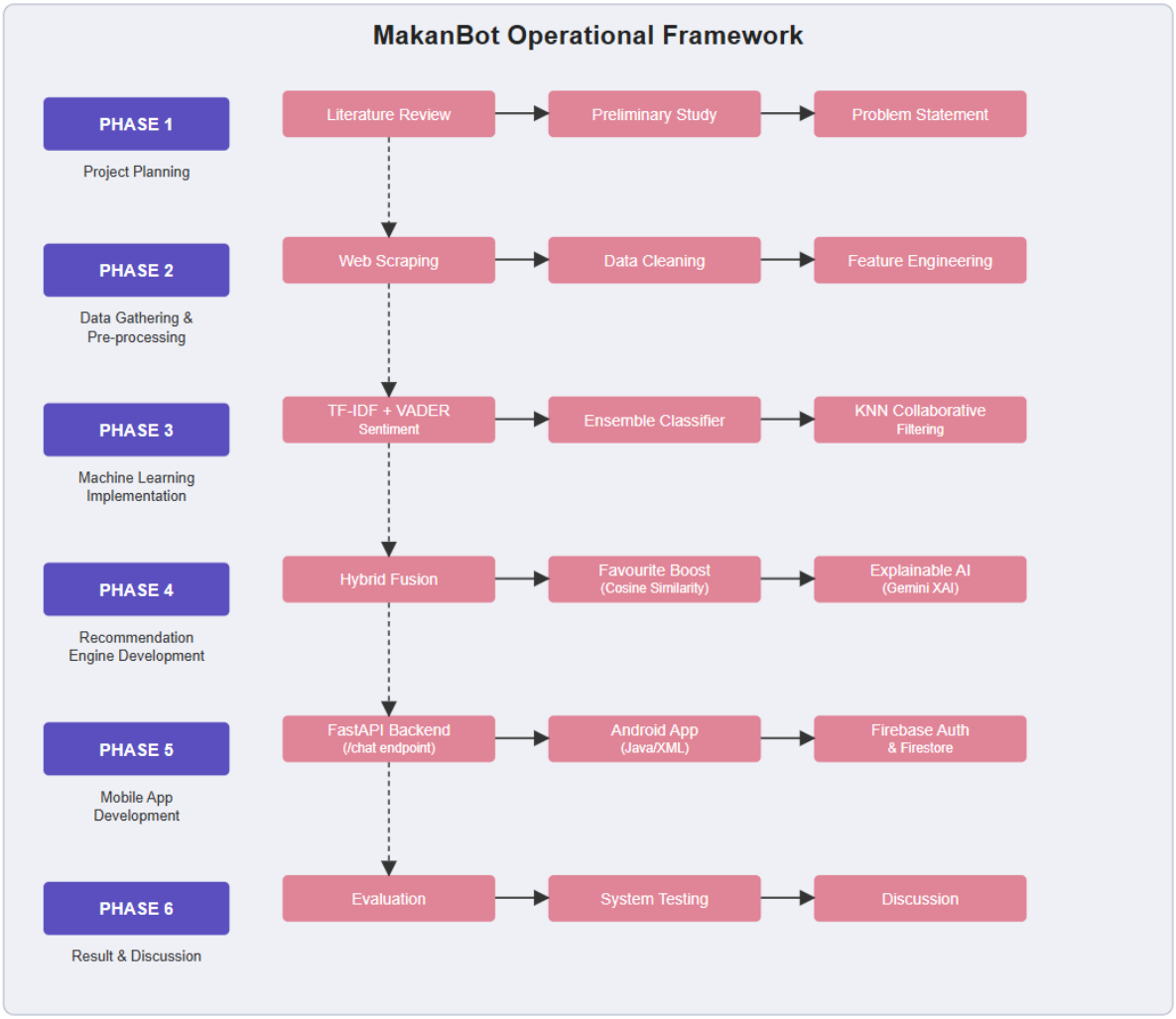


Figure 3.1 Operational Framework

The development of the MakanBot mobile chatbot for restaurant recommendation is organised into a series of structured phases. The project's processes are categorised into separate phases in the development lifecycle. This includes project planning, data gathering and pre-processing, machine learning model development, recommendation engine integration, mobile application development, and finally performance analysis followed by the result and discussion.

Phase 1: Project Planning

Referring to Figure 3.1, before developing the MakanBot system, a preliminary investigation was conducted to identify the most suitable domain, geographic scope, and data source. Kuala Lumpur (KL) was chosen as the target city due to its high restaurant density and culinary diversity (Chinese, Malay, Indian, Japanese, Western, and Cafe/Dessert cuisines). A publicly available dataset hosted on Kaggle, originally compiled by a Master's student from University of Malaya, was selected as the primary data source because it focuses entirely on Kuala Lumpur restaurants and provides both structured restaurant metadata and user review text, which is ideal for building a hybrid recommendation system.. A literature review was also carried out in Chapter 2 to compare existing recommendation approaches (content-based, collaborative filtering, and hybrid) and to identify the research gap namely the lack of mobile chatbot solutions that combine hybrid recommendation with explainable AI for Malaysian restaurants.

Phase 2: Data Gathering and Pre-processing

This project makes use of a publicly available restaurant dataset hosted on Kaggle [47]. The dataset was originally compiled by a Master's student from the University of Malaya (UM) and focuses entirely on restaurants located in Kuala Lumpur, Malaysia, making it highly suitable for the scope of this project. Using a reputable pre-compiled dataset instead of performing manual web scraping offers several advantages: it ensures data reliability and consistency, saves a significant amount of time and effort, and allows the project to focus on machine learning model development and mobile application implementation rather than on data collection logistics.

Although the raw dataset acquired from Kaggle already underwent basic cleaning by the original author, additional pre-processing was performed in this project to transform the data into a format suitable for training the hybrid recommendation models. The workflow begins by loading the CSV files using Pandas and inspecting their contents. The following steps are then applied:

1. **Data loading and missing-value removal**
   the raw CSV files downloaded from Kaggle are loaded with pd.read_csv(), and rows with missing Review, Rest_Match, or Rating fields are dropped using .dropna() to ensure data quality.

2. **Hybrid cuisine inference**

   because the original dataset does not contain a reliable cuisine label for every restaurant, a custom hybrid classifier is applied. A cuisine keyword dictionary is defined for six categories (Japanese, Western, Chinese, Indian, Malay, Dessert). For each restaurant, all reviews are aggregated into a single text block, and a scoring function assigns 10 points for each keyword match in the restaurant name and 1 point for each keyword occurrence in the review text. The cuisine with the highest total score is assigned; if no keyword matches, the restaurant is labelled Other.

3. **Halal status synchronisation and export**

   for consistency, any restaurant classified as Malay cuisine has its HALAL_UPDATED field set to YES.

   The cleaned profile is saved as Master_Restaurants_Profile_Updated.csv for downstream use by the recommender engine.

4. **Feature engineering**

   the review dataset is merged with the cleaned restaurant metadata, and a combined text feature Full_Text_Features is constructed by concatenating *Cuisine* + *Halal* + *Review*. Each review is then scored using VADER's SentimentIntensityAnalyzer, producing a compound polarity score between −1 (very negative) and +1 (very positive). A binary target label Target is created, marking a review as relevant (1) if its Rating is ≥ 4.0, otherwise not relevant (0).

5. **TF-IDF vectorisation and train-test split**

   the combined text feature is vectorised using TfidfVectorizer (top 1000 terms, English stop words removed), and the numerical Sentiment_Score is appended to the TF-IDF matrix via scipy.sparse.hstack. Finally, the data is stratified and split 80/20 into training and testing subsets using train_test_split with random_state=42 for reproducibility.

Phase 3: Machine Learning Implementation

Three complementary machine learning components were chosen to power the recommendation engine. Each was selected based on its ability to capture a different signal of user preference.

a) TF-IDF Vectoriser (Content Feature Extraction)

Term Frequency–Inverse Document Frequency (TF-IDF) is a classical technique used to convert text features (restaurant name + cuisine) into numeric vectors, while down-weighting common words. The TF-IDF weight is calculated as:

$$w_{t,d} = tf_{t,d} \times \log \frac{N}{df_t}$$

Where:

- $tf_{t,d}$ is the frequency of term *t* in document *d*.
- N is the total number of documents in the corpus.
- df_t is the number of documents containing term *t*.

b) Ensemble Content Classifier (Logistic Regression + Random Forest)

An ensemble classifier combines multiple base learners to produce more robust predictions. In this project, the ensemble takes the TF-IDF vector concatenated with the VADER sentiment score as input, and outputs a probability score content_prob ∈ [0, 1] indicating how likely a restaurant is relevant to the user's query. The Random Forest component is governed by Gini impurity:

$$G = 1 - \sum_{i=1}^{C} p_i^2$$

Where *G* is the Gini impurity, p_i is the proportion of samples belonging to class *i*, and *C* is the number of classes.

c) K-Nearest Neighbours (KNN) Collaborative Filtering

KNN from the Surprise library performs user-based collaborative filtering. Given a user *u* and an item *i*, KNN predicts the rating based on similarity to *k* nearest users who have rated *i*. Cosine similarity is used as the similarity metric:

$$\text{sim}(u, v) = \frac{\sum_{i \in I_{uv}} r_{ui} \cdot r_{vi}}{\sqrt{\sum r_{ui}^2} \cdot \sqrt{\sum r_{vi}^2}}$$

The KNN output is normalised to [0, 1] as collab_pred for hybrid fusion.

d) Favourite-Similarity Boost (Cosine Similarity)

To personalise recommendations further, MakanBot builds a taste vector by averaging the TF-IDF vectors of all restaurants the user has saved as favourites. Each candidate's cosine similarity to this vector provides a fav_sim score:

$$\text{fav\_sim} = \cos\left(\vec{v}_{candidate}, \vec{v}_{taste}\right) = \frac{\vec{v}_{candidate} \cdot \vec{v}_{taste}}{|| \vec{v}_{candidate} ||\cdot|| \vec{v}_{taste} ||}$$

e) Performance Metric

To evaluate the effectiveness of the trained models, several standard performance metrics will be adopted. For the content based ensemble classifier, accuracy, precision, recall, and F1 score will be used to measure overall classification performance, while the confusion matrix and Receiver Operating Characteristic (ROC) curve with Area Under the Curve (AUC) value will be used to provide a deeper analysis of how well the model separates relevant from irrelevant reviews. For the K Nearest Neighbours collaborative filter, Root Mean Square Error (RMSE) and Mean Absolute Error (MAE) will be adopted as the regression metrics, since the filter predicts ratings on a continuous 1 to 5 scale rather than producing a binary label. The detailed evaluation results obtained from these metrics are presented in Chapter 4.

f) Hybrid Score Fusion

The three signals are combined into a single final score. Weighting is adaptive if the user has saved favourites, the favourite-similarity score is given weight; otherwise the original content–collaborative split is used:

- With favourites: final_score = 0.5 × content_prob + 0.2 × collab_pred + 0.3 × fav_sim
- Without favourites: final_score = 0.7 × content_prob + 0.3 × collab_pred

To analyse model performance, three metrics are used accuracy, precision, and recall. Accuracy measures overall correctness, precision measures the classifier's ability to identify only relevant restaurants, and recall measures the ability to find all relevant ones in the dataset. Python is the primary programming language used during this phase, owing to its rich

ecosystem of libraries including Scikit-learn, NLTK, Surprise, Pandas, and NumPy, which accelerate the development of recommendation models.

Phase 4: Recommendation Engine and Backend Development

In this phase, the insights and predictions generated by the machine learning models are integrated into a functional recommendation engine. The engine is encapsulated in a class called RestaurantEngine (in recommender.py), and is served to the mobile app via a FastAPI REST backend (main.py).

The backend exposes a /chat endpoint that performs the following pipeline on each user message:

1. **Intent Analysis**
   the user message is sent to Google Gemini (gemini-3.1-flash-lite) to classify intent as *chitchat* or *recommendation*, and to extract cuisine and halal preferences.
2. **Vague-Query Fallback**
   if the user didn't specify a cuisine (e.g., "I'm hungry"), the system randomly selects from the user's saved preferences, or from a default pool if none exist.
3. **Hybrid Recommendation**
   RestaurantEngine.recommend() is invoked to rank candidate restaurants.
4. **Google Places Enrichment**
   the top result is enriched with a photo, address, rating, menu link, and reservation link via the Google Places API.
5. **Explainable AI (XAI)**
   Gemini generates a one-sentence human-readable justification ("Because you love Japanese, and this place is similar to restaurants you've saved before").

Session memory is used to avoid repeating the same restaurant within one conversation, while Firebase Firestore stores user profiles, saved preferences, and favourites persistently.

Phase 5: Mobile Application Development

The mobile client is a native Android application developed in Java using Android Studio. Key components include:

- SplashActivity, LoginActivity, SignupActivity : handle onboarding using Firebase Authentication.
- MainActivity : the core chat screen, uses ChatAdapter and RecyclerView to render user/bot messages.
- PreferencesActivity : lets the user save cuisine preferences.
- FavoritesActivity : lists restaurants the user has saved for personalisation.
- RestaurantDetailActivity : displays the enriched recommendation card with image, rating, halal badge, menu button, reservation button, and Google Maps link.
- ChatDatabase / ChatDao : local Room database that caches chat history for offline viewing.

The app communicates with the FastAPI backend via Retrofit HTTP client using JSON payloads.

Phase 6: Result and Discussion

Lastly, the performance metrics of the recommendation engine are compared against baseline approaches (content-only and collaborative-only) to determine the most effective configuration. By comparing precision, recall, accuracy, and user-perceived relevance, and considering the trade-offs between personalisation and diversity, a comprehensive discussion of the strengths and weaknesses of the hybrid model is provided. This comprehensive review provides future researchers looking to use hybrid recommendation for the F&B domain in Malaysia with vital information and provides a beacon for future research projects.

3.2 System Architecture Diagram

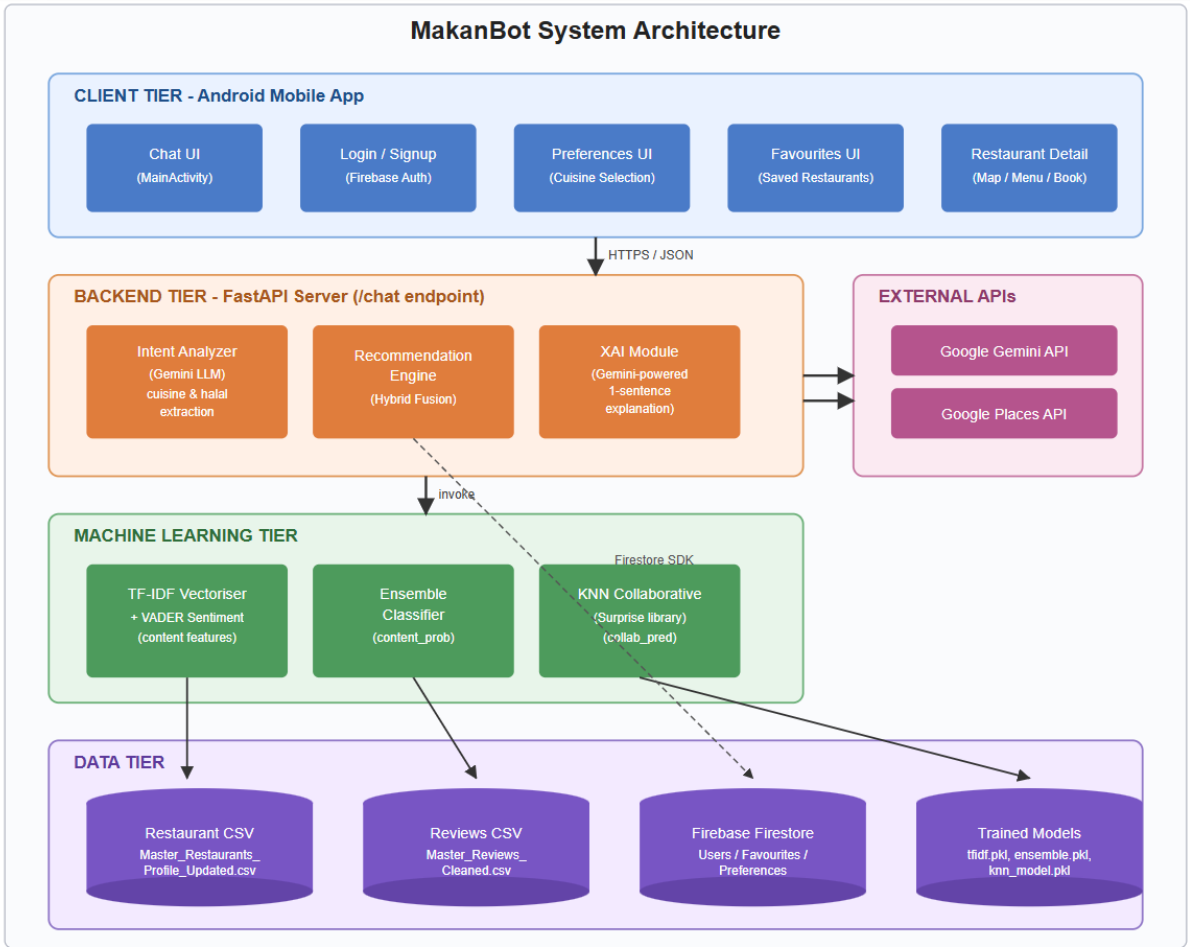


Figure 3.2 System Architecture Diagram

Diagrams of system architecture provide a visual illustration of the numerous components that make up a system and show how those components communicate with one another. In Figure 3.2 above, the proposed MakanBot restaurant recommendation chatbot is illustrated, showing how it operates through each module and database involved. There are four main tiers in the architecture: the client tier (Android mobile app), the backend tier (FastAPI server), the machine learning tier (TF-IDF vectoriser, ensemble classifier, and KNN collaborative model), and the data tier (CSV dataset, Firebase Firestore, and the Google Places API).

The user begins by typing a message such as "Any good Japanese food?" into the chat interface of the mobile app. The message is sent over HTTPS to the FastAPI /chat endpoint along with the user's unique identifier. The Intent Analyzer, powered by Google Gemini, extracts the cuisine and halal filter from the message. The Recommendation Engine then loads the

restaurant dataset, filters candidates by cuisine, scores each using the three ML signals (content, collaborative, favourite-similarity), and returns the top candidate. The winning restaurant is enriched with live data from the Google Places API (image, rating, reservation link), and a human-readable explanation is produced by the XAI module. Finally, the recommendation is sent back to the mobile app, which renders a visually rich restaurant card in the chat.

3.3 Use Case Diagram

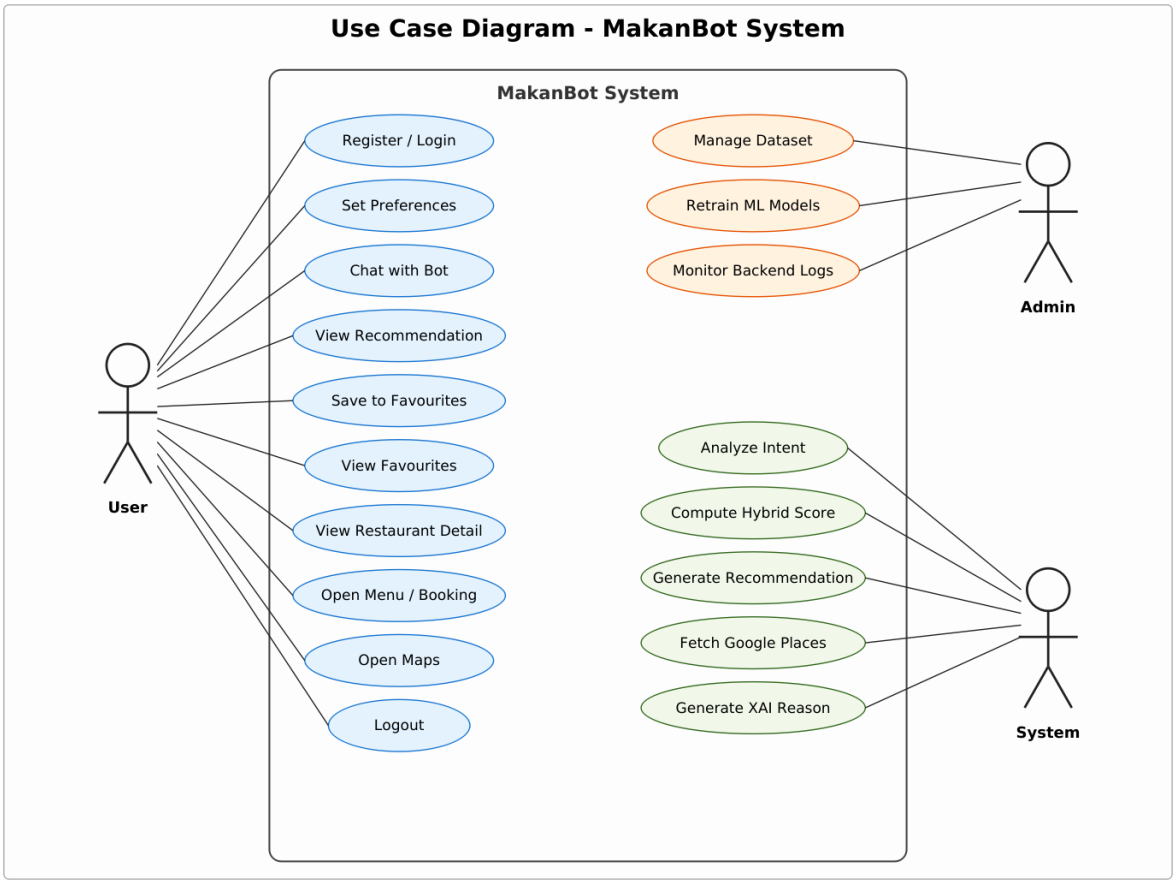


Figure 3.3 Use Case Diagram

From Figure 3.3 above, there are three roles involved in the MakanBot system: user, admin, and system, each performing different tasks. On the user side, the system allows a person to register and login, set their cuisine preferences, and chat with the bot to receive restaurant recommendations. After receiving a recommendation, the user can view restaurant details, access the menu and reservation through external links, save the restaurant to their favourites for future personalisation, and view their list of saved favourites. Additionally, users can logout at any time.

The admin is responsible for managing backend resources, including updating the restaurant dataset when new restaurants are added, retraining the machine learning models periodically, and monitoring backend logs to ensure uptime. Any changes to the underlying dataset and ML models are handled solely by the admin.

The system actor performs the core automated tasks. It analyses user intent using the Gemini LLM, generates hybrid recommendations by fusing the content, collaborative, and favourite-similarity signals, fetches live restaurant metadata from the Google Places API, and generates a human-readable explanation (XAI) for why the restaurant was recommended.

3.4 Activity Diagram

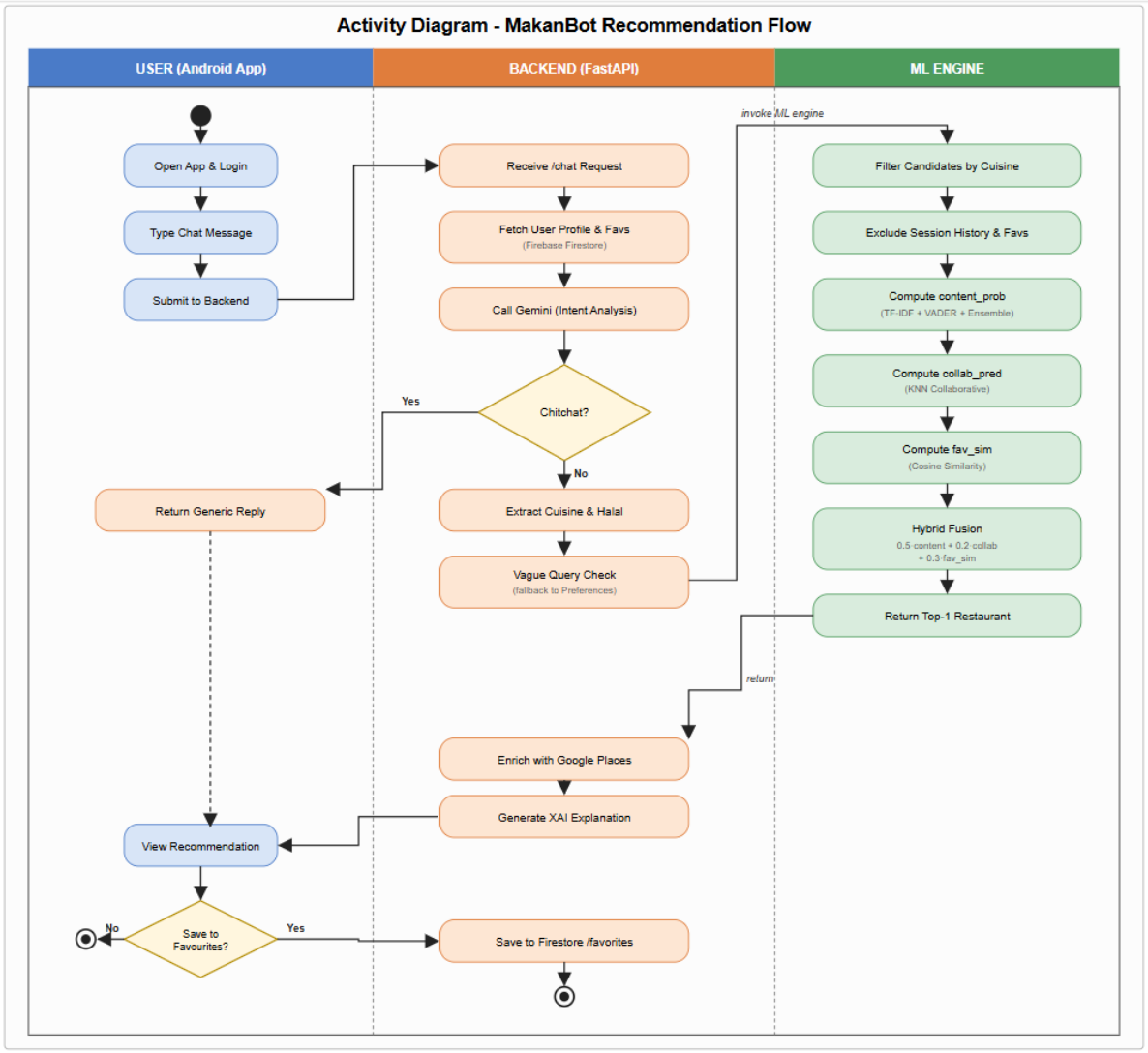


Figure 3.4 Activity Diagram

Figure 3.4 shows the activity diagram for the proposed MakanBot system. There are three swim lanes involved, which are user, backend, and ML engine. At the beginning, the user starts by opening the mobile application and logging in through Firebase Authentication. Once logged in, the user types a natural language message such as "I want Japanese food tonight" into the chat interface and submits it.

The message, together with the user's unique identifier, is sent to the FastAPI backend. The backend first fetches the user profile from Firebase Firestore including saved preferences and favourites. It then forwards the message to Google Gemini for intent analysis. If the intent is classified as chitchat (e.g., "hello", "how are you"), a generic reply is returned immediately. Otherwise, the cuisine and halal filters are extracted and passed to the ML engine.

The ML engine filters the restaurant dataset by cuisine, excludes previously recommended restaurants in the same session, and computes three scores for each candidate: content probability (TF-IDF + sentiment fed into the ensemble classifier), collaborative prediction (from the KNN model), and favourite similarity (cosine similarity to the user's taste vector). These three scores are fused using the adaptive weighting formula, and the top-ranked restaurant is returned to the backend.

The backend then enriches the top recommendation with live metadata from the Google Places API (image, address, rating, menu, reservation link) and generates a one-sentence explanation through Gemini XAI. The final JSON response is sent back to the mobile app, which renders it as a visually rich restaurant card. The user can then view the recommendation, open the menu or reservation link, view the location on Google Maps, or save it to favourites for future personalisation.

3.5 Timeline for FYP1 and FYP2

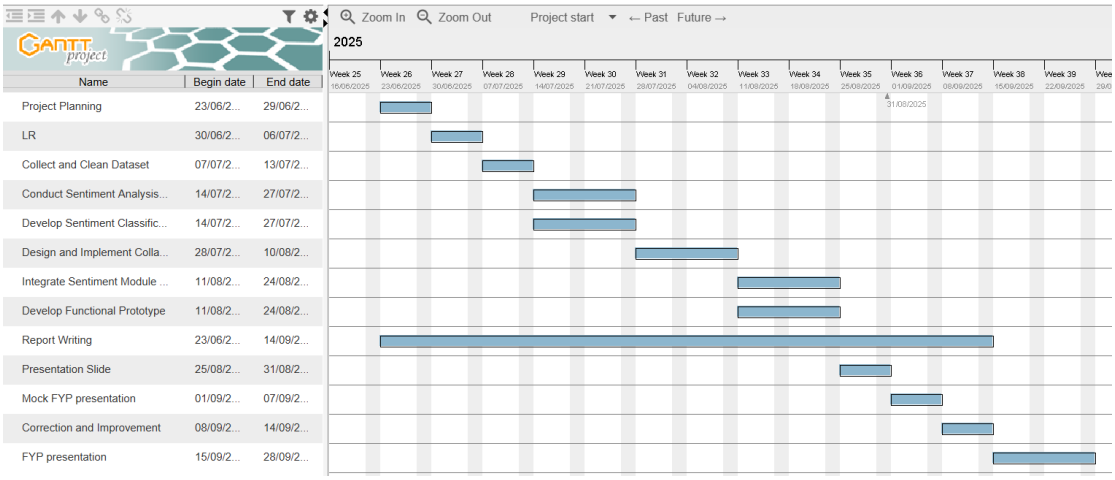


Figure 3.5 Timeline for FYP I

The project timeline was meticulously planned using a Gantt chart to ensure systematic progress throughout the Final Year Project (FYP). The project commenced with Project Planning from 23rd to 29th June 2025, followed by a Literature Review (LR) phase from 30th June to 6th July. Data preparation tasks, including Collecting and Cleaning the Dataset, were completed by 13th July. Subsequent stages, such as Sentiment Analysis and Classification Model Development, as well as System Design and Implementation, were conducted concurrently from mid to late July. Module integration and prototype development spanned from 11th to 24th August. Report writing started early, from 23rd June, and continued through the majority of the project to 14th September. In the final stages, emphasis was placed on Presentation Slide Preparation, Mock Presentations, and Correction and Improvement activities, leading up to the final FYP Presentation, which is scheduled from 15th to 28th September 2025. This structured timeline ensured that each phase was allocated adequate time, facilitating organized development and timely completion of all deliverables.

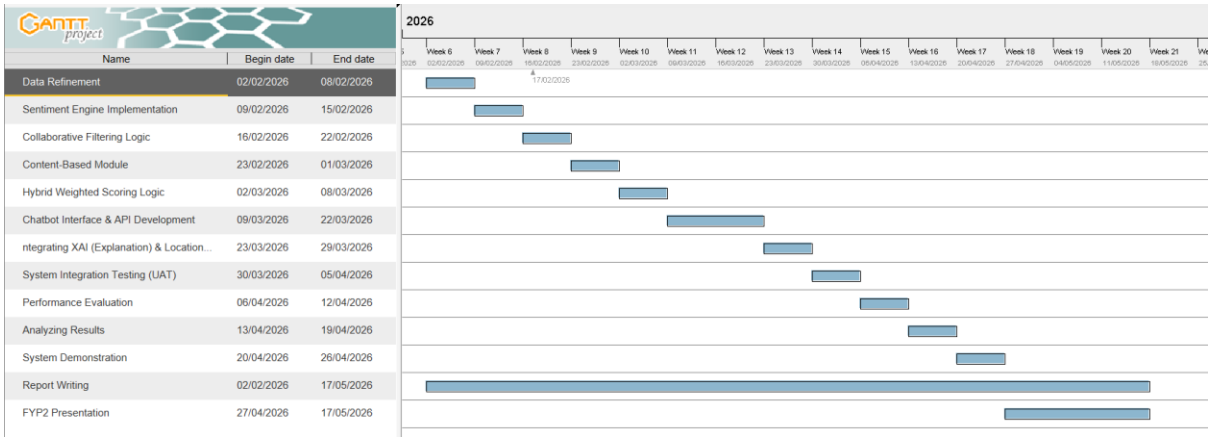


Figure 3.6 Timeline for FYP 2

The FYP2 timeline was systematically executed over a 14-week period, transitioning from the conceptual designs of FYP1 to a fully validated system implementation. Commencing in February 2026, the phase began with rigorous data refinement, followed by the optimization of the Support Vector Machine (SVM) and Random Forest models for the content-based module and the integration of K-Nearest Neighbors (KNN) for collaborative filtering. Throughout March, the focus shifted to developing a TF-IDF based sentiment engine and a hybrid weighted scoring logic to unify the recommendation core, while simultaneously building the Android-based chatbot interface and API connectivity. The inclusion of Explainable AI (XAI) features provided transparency, leading into an April dedicated to performance evaluation using Accuracy and RMSE metrics. This structured progression ensured that the hybrid machine learning system was fully optimized for the final demonstration and viva by Week 14 in May 2026.

Chapter 4

System Design

4.1 System Block Diagram

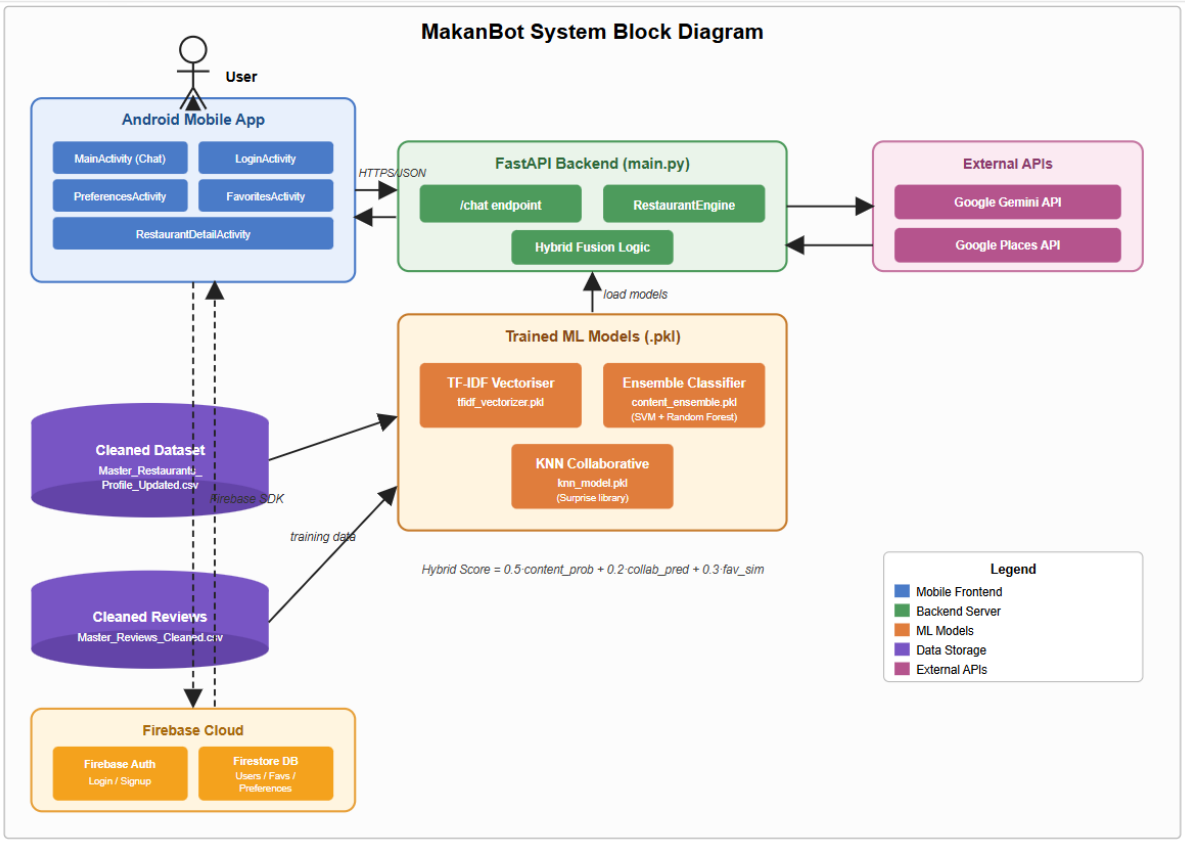


Figure 4.1 System Block Diagram

Figure 4.1 illustrates the system architecture of the MakanBot Mobile Chatbot for Restaurant Recommendation, which integrates machine learning, large language model (LLM) intent analysis, cloud authentication, and a native Android application interface. At the system's core are three trained machine learning components: the TF-IDF Vectoriser, the Ensemble Classifier (SVM + Random Forest with Soft Voting), and the KNN Collaborative Filter. The TF-IDF vectoriser is responsible for converting text features , a concatenation of *Cuisine* + *Halal Status* + *Review* into numerical vectors, while the ensemble classifier uses these features along with VADER sentiment scores to produce a *content probability* (content_prob) score. The KNN collaborative filter, trained using the Surprise library, produces a personalised prediction (collab_pred) for each candidate restaurant.

These pre-trained models are serialised as .pkl files (tfidf_vectorizer.pkl, content_ensemble.pkl, knn_model.pkl) and loaded at runtime by the FastAPI backend, defined in the main.py file, which serves as the system's server-side component. The FastAPI application exposes a /chat endpoint that handles three core responsibilities: first, it forwards the user's message to the Google Gemini API for intent analysis, extracting the target cuisine and halal preference. Second, it invokes the RestaurantEngine class (defined in recommender.py) to perform hybrid recommendation using the three ML signals plus a fourth *favourite-similarity* score computed using cosine similarity. Third, it enriches the chosen restaurant with live metadata (photo, address, rating, menu link, reservation link) from the Google Places API, and generates a human-readable one-sentence explanation through Gemini's XAI module.

The mobile frontend consists of a native Android application developed in Java, comprising several activities: SplashActivity, LoginActivity, SignupActivity, MainActivity (chat interface), PreferencesActivity, FavoritesActivity, and RestaurantDetailActivity. Firebase Authentication handles user login and registration, while Firebase Firestore stores user profiles, saved preferences, and favourite restaurants persistently across devices. The processed restaurant dataset (Master_Restaurants_Profile_Updated.csv) serves as the input item catalogue for the recommendation engine. Together, these components form a complete mobile chatbot system that delivers intelligent, personalised, and explainable restaurant recommendations to users in Kuala Lumpur.

4.2 System Components Specifications

The MakanBot recommendation system is composed of several core components that work together to provide an interactive and intelligent mobile chatbot experience. At the heart of the system is the hybrid machine learning module, which combines content-based filtering, collaborative filtering, and favourite-similarity boosting. The content-based component uses a TF-IDF vectoriser to convert text features into numerical vectors, coupled with an ensemble classifier built from Support Vector Machine (SVM with linear kernel) and Random Forest models using soft voting. The collaborative component uses KNN Basic from the Surprise library with item-based cosine similarity to model user-restaurant interaction patterns. These three models are pre-trained in Jupyter Notebook and saved as .pkl files to be loaded in real-time when a user submits a chat message.

The mobile frontend is constructed as a native Android application containing several key activities. SplashActivity serves as the entry point with app branding, LoginActivity and SignupActivity handle user authentication through Firebase Auth, and MainActivity is the core chat interface where users type natural-language queries and view bot responses rendered through ChatAdapter and RecyclerView. PreferencesActivity lets users select their preferred cuisines from checkboxes, FavoritesActivity displays a list of saved restaurants using FavoritesAdapter, and RestaurantDetailActivity renders a rich restaurant card with image, rating, halal badge, menu link, reservation link, and Google Maps location. A local Room database (ChatDatabase + ChatDao) caches chat history for offline viewing.

The backend is managed through a Python-based FastAPI application contained in the main.py file, with the recommendation engine encapsulated in recommender.py. This server handles user requests, loads the machine learning models at startup, reads the processed restaurant dataset from Master_Restaurants_Profile_Updated.csv, and applies hybrid filtering based on the user's natural-language input and their saved favourites. Upon processing, the system delivers a single top-ranked recommendation by dynamically enriching it with Google Places metadata and generating a natural-language explanation using Google Gemini. This seamless interaction between the mobile client and the FastAPI server ensures that the system provides accurate, responsive, and user-friendly recommendations in real time.

The entire application is deployed as two separate tiers: the Android app runs on the user's mobile device (Android 8.0+), while the FastAPI backend runs on a cloud-hosted server accessed over HTTPS. The use of Python (FastAPI, scikit-learn, Surprise, NLTK) for the backend, combined with Java/XML (Android SDK, Retrofit, Firebase SDK) for the frontend, provides a lightweight yet robust architecture that supports the integration of data processing, model inference, cloud storage, and mobile user engagement. Together, these components form a cohesive system designed to enhance the restaurant discovery experience by leveraging hybrid recommendation, large language models, and mobile-first design.

4.3 Data Visualization

The following section presents a visual analysis of the processed Kuala Lumpur restaurant dataset through various charts, providing insights into the distribution of halal and non-halal restaurants, the most-reviewed restaurants, and the menu items most commonly mentioned in user reviews.

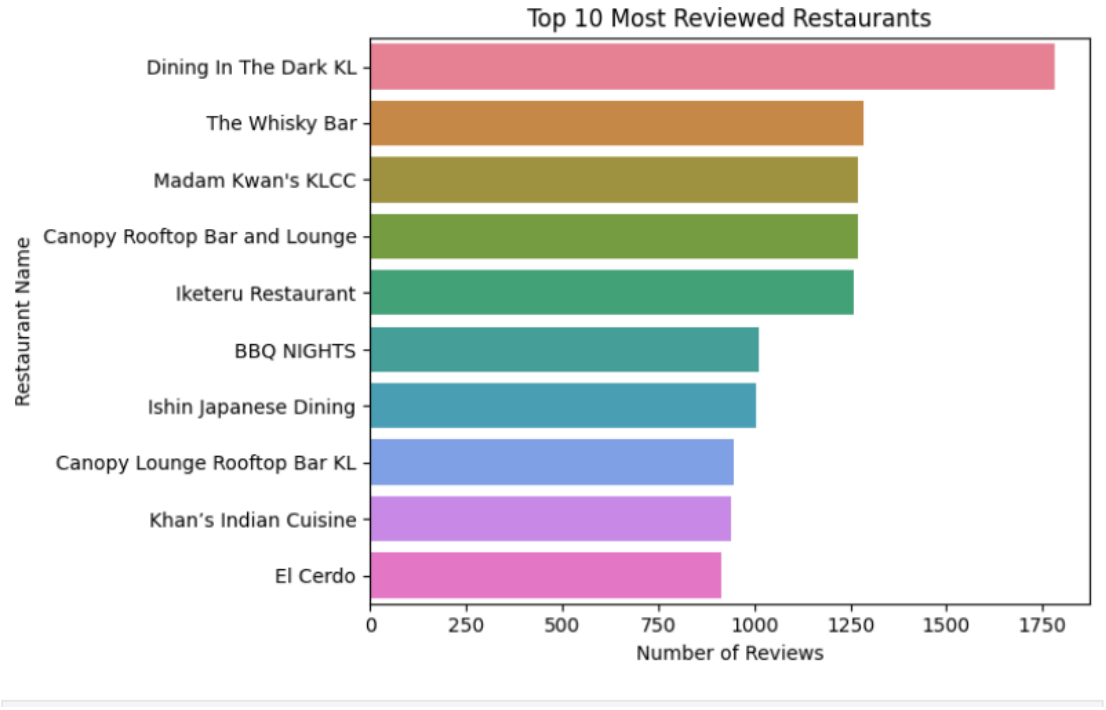


Figure 4.2 Top 10 Most Reviewed Restaurants

Figure 4.2 highlights the restaurants in Kuala Lumpur that have received the highest number of reviews. Leading the list is Dining In The Dark KL, which stands out with nearly 1,800 reviews, significantly more than the rest. Following closely are The Whisky Bar, Madam Kwan's KLCC, and Canopy Rooftop Bar and Lounge, each garnering around 1,200 to 1,300 reviews. The rest of the restaurants, including Iketeru Restaurant, BBQ NIGHTS, and El Cerdo, show a gradual decline in review counts but still maintain strong engagement with over 900 reviews each. This distribution indicates that these top venues are not only popular among diners but also actively engaged with on review platforms, suggesting high visibility and consistent customer interest. Such data is valuable for identifying key players in the local food scene and understanding where consumer attention is most focused.

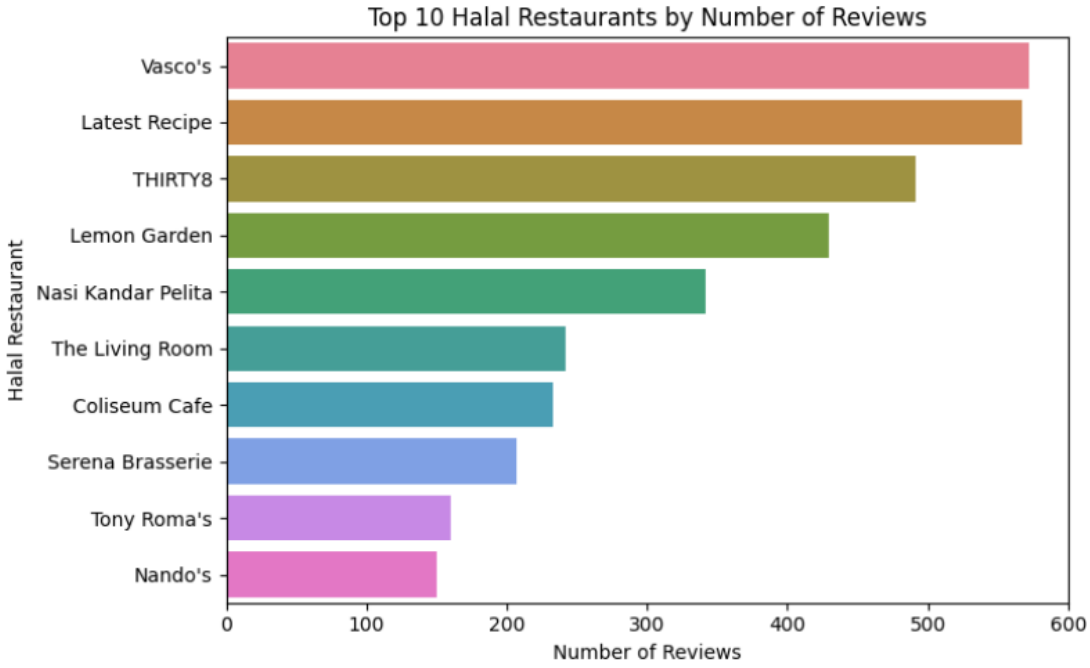


Figure 4.3 Top 10 Halal Restaurants by Number of Reviews

Figure 4.3 illustrates the top 10 halal restaurants based on the number of reviews they have received. Vasco's leads the ranking with nearly 580 reviews, followed closely by Latest Recipe and THIRTY8, each also boasting high review counts, indicating strong customer engagement and popularity. Lemon Garden and Nasi Kandar Pelita occupy the mid-tier range, suggesting a solid reputation among diners. Meanwhile, Coliseum Cafe, The Living Room, Tony Roma's, and Nando's have a moderate number of reviews, while Restoran Rebung Chef Ismail rounds out the list with the lowest count among the top ten. These findings reflect not only the popularity of these establishments but also the likelihood that they have a strong online presence and customer feedback loop. Restaurants with higher review counts may be leveraging digital platforms more effectively or are located in high-traffic areas, attracting more customers and reviews. This insight is valuable for businesses aiming to improve visibility or for researchers analyzing dining preferences in the halal food segment.

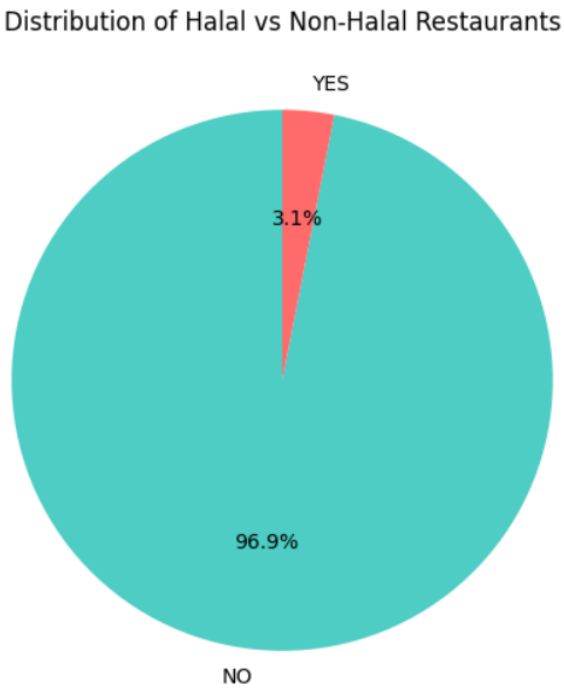


Figure 4.4 Distribution of Halal vs Non-Halal Restaurants in KL

Figure 4.4 illustrates the distribution of Halal and Non-Halal restaurants in Kuala Lumpur, revealing a significant imbalance between the two categories. Only 3.1% of the restaurants in the dataset are Halal-certified, while the vast majority, 96.9%, are not certified as Halal. This indicates that Halal-certified dining options are relatively scarce in comparison to Non-Halal establishments within the city. The disparity may be influenced by factors such as the diversity of cuisines offered, the ownership and target customer base of the restaurants, or the absence of formal Halal certification despite potentially offering Halal-friendly food. This finding highlights the importance of incorporating Halal certification as a filtering criterion in a restaurant recommendation system, particularly for Muslim consumers seeking suitable dining options in Kuala Lumpur.

Figure 4.5 Most Frequent Words in Reviews

Figure 4.5 visualizes the most frequently used words in restaurant reviews from Kuala Lumpur, highlighting the terms that appear most prominently based on their frequency. Larger words, such as *"show"*, *"service"*, *"good"*, *"delicious"*, *"menu"*, *"drink"*, and *"taste"*, indicate aspects that diners commonly emphasize in their feedback. Positive descriptors like *"excellent"*, *"nice"*, *"best"*, and *"great"* suggest that many reviews contain favorable sentiments, while words like *"price"* and *"less"* may reflect discussions on value for money. The presence of terms related to food (*"eat"*, *"ordered"*, *"tasty"*, *"dishes"*), service quality (*"attentive"*, *"friendly"*, *"waiter"*), and dining context (*"table"*, *"time"*, *"KL"*) shows that customers often comment on both the culinary experience and the overall atmosphere. This pattern provides useful insights for understanding customer priorities and areas of interest, which can be leveraged in building a more targeted and effective restaurant recommendation system.

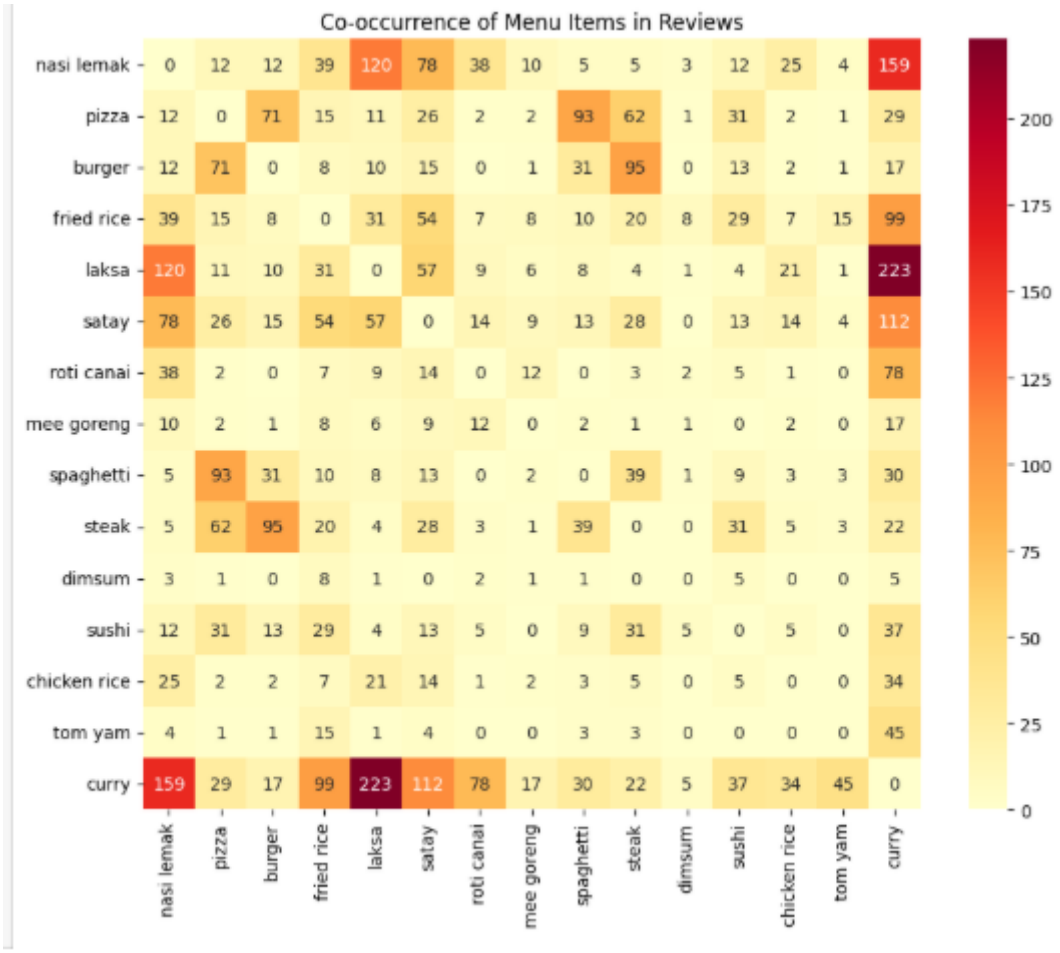


Figure 4.6 Co-occurrence of menu items in Reviews

Figure 4.6 displays the co-occurrence of menu items mentioned together in Kuala Lumpur restaurant reviews, where darker shades indicate higher frequencies of association between dishes. The strongest pairing is between laksa and curry, appearing together 223 times, followed by nasi lemak with curry (159), satay with curry (112), and fried rice with curry (99), suggesting that curry is a central item often linked with multiple local dishes. Western combinations such as burger with steak (95) and pizza with spaghetti (93) also stand out, reflecting common pairings in certain restaurant offerings. These patterns reveal popular food combinations discussed by customers, providing valuable insights for building menu-based recommendation features that suggest complementary dishes based on customer preferences and ordering trends.

4.4 Machine Learning

After the pre-processing pipeline is completed, the two master CSV files serve as the cleaned and enriched data source for the machine learning stage:

1. **Master_Restaurants_Profile_Updated.csv**
   contains structured metadata for each restaurant in Kuala Lumpur, including the restaurant name, inferred cuisine category (Chinese, Malay, Indian, Japanese, Western, Dessert), halal status, address, and overall rating.
2. **Master_Reviews_Cleaned.csv**
   contains user-generated review text, review ratings, and pre-cleaned content associated with each restaurant, which serves as the foundation for sentiment analysis and the training of the content-based recommendation model.

The combination of these two datasets provides both **item features** (used by the content-based model) and **user interaction data** (used by the collaborative filtering model), forming the foundation for the hybrid recommendation system developed in this project.

	Restaurant	url	HALAL RES	Rest_Matc	HALAL_UP	Cuisine
2	Chambers Grill	https://www.tripadvisor.com.my/Restaurant_Review-g29	NO	chambers	NO	Western
3	Positano Risto	https://www.tripadvisor.com.my/Restaurant_Review-g29	NO	positano ri	NO	Western
4	Sausage KL Cafe & Deli	https://www.tripadvisor.com.my/Restaurant_Review-g29	NO	sausage kl	NO	Western
5	Canopy Rooftop Bar and Lounge	https://www.tripadvisor.com.my/Restaurant_Review-g29	NO	canopy roc	NO	Western
6	Iketeru Restaurant	https://www.tripadvisor.com.my/Restaurant_Review-g29	NO	iketeru res	NO	Japanese
7	Knowhere Bangsar	https://www.tripadvisor.com.my/Restaurant_Review-g29	NO	knowhere	NO	Western
8	Ishin Japanese Dining	https://www.tripadvisor.com.my/Restaurant_Review-g29	NO	ishin japan	NO	Japanese

Figure 4.7 Master_Restaurants_Profile_Updated.csv

Author	Review	Rating	Restauran	Rest_Matc	Sentiment_Label
Suhana S	We are meat lover. LOVED our Black Angus Tomahawk. Perfectly made as requested. Portobello sides	5	Chamberâ	chambers	Positive
Lidia Plotkina	Nice cute interior on the ground floor of Hilton. Great and welcoming service. Bug plus on the ambianc	3	Chamberâ	chambers	Neutral
Jack Low	I like my meat and Chambers does it again on my recent visit with an exceptional Angus ribeye med rar	5	Chamberâ	chambers	Positive
Alton Wong	Steaks (especially the tomahawk) truly lived up to my expectations! Saw some pictures of it on IG befor	5	Chamberâ	chambers	Positive
Christ Ng	Very nice hospitality ,especially the General Manager(which we forgot his gameðŸ˜‚) very friendly , nic	5	Chamberâ	chambers	Positive
Joanna Blas	Memorable dinner at Chamber's Grill, Hilton KL. Succulent wagyu tomahawk that melts in your mouth	5	Chamberâ	chambers	Positive
Jewel M.	Tender juicy tomahawk, will be back! Good service from the team. Thank you Adli, Azu, Stephanie, Che	5	Chamberâ	chambers	Positive
Jonathan Lim	A small bar connected to the Chambers grill restaurant. Concoctions are rum based and claims to be t	3	Chamberâ	chambers	Neutral
AJ Ash	Hosted couple of friends at the Chambers Grill of Hilton KL Sentral for dinner yesterday. Pleasant atmc	5	Chamberâ	chambers	Positive

Figure 4.8 Master_Reviews_Cleaned.csv

a) Model Construction

The provided code illustrates the application of two complementary machine learning approaches in the MakanBot recommendation engine: a content-based ensemble classifier (combining SVM and Random Forest with soft voting) and a KNN collaborative filter. The

primary goal is to combine the strengths of both approaches , content features capture *what a restaurant is*, while collaborative features capture *who likes it* producing a hybrid recommendation that is both relevant and personalised.

```
# ========================================
# C. TUNING THE ENSEMBLE
# ========================================
# Using class_weight='balanced' to handle the high-accuracy trap you saw earlier
print("Tuning Ensemble with Text + Sentiment features...")
svm_grid = GridSearchCV(
    SVC(probability=True, class_weight='balanced'),
    {'C': [1, 10], 'kernel': ['linear']}, cv=3
)
svm_grid.fit(X_train_final, y_train)

rf_grid = GridSearchCV(
    RandomForestClassifier(class_weight='balanced'),
    {'n_estimators': [100], 'min_samples_leaf': [5]}, cv=3
)
rf_grid.fit(X_train_final, y_train)

ensemble = VotingClassifier(
    estimators=[('svm', svm_grid.best_estimator_), ('rf', rf_grid.best_estimator_)],
    voting='soft'
)
ensemble.fit(X_train_final, y_train)

print(f"✅ Ensemble Trained. Best SVM C: {svm_grid.best_params_['C']}")
```

Figure 4.9 Ensemble Classifier (SVM + Random Forest)

The first model component is a Voting Classifier that combines a Support Vector Machine (SVM) with a linear kernel and a Random Forest classifier, both tuned through GridSearchCV with 3-fold cross-validation. The SVM uses class_weight='balanced' to handle class imbalance between positive (rating ≥ 4.0) and negative reviews, and explores hyperparameter values $C \in \{1, 10\}$. The Random Forest uses 100 estimators with a minimum of 5 samples per leaf, also with balanced class weights. The two models are combined using soft voting, where each classifier outputs class probabilities and the final prediction is based on the averaged probabilities. This approach leverages SVM's effectiveness in high-dimensional TF-IDF feature spaces while benefiting from Random Forest's robustness to noise. The feature input to the ensemble is a 1001-dimensional vector: the first 1000 dimensions come from the TF-IDF vectoriser applied to the combined *Cuisine* + *Halal* + *Review* text, and the final dimension is the VADER compound sentiment score.

```python
# ========================================
# 2. COLLABORATIVE FILTERING (KNN) - MEMORY OPTIMIZED
# ========================================
print("\n--- TUNING KNN (Item-Based Collaborative) ---")
reader = Reader(rating_scale=(1.0, 5.0))
data = Dataset.load_from_df(reviews_df[['Author', 'Rest_Match', 'Rating']], reader)

# IMPORTANT: We set user_based to False to compare Restaurants instead of Users
# This reduces the matrix size from 73,000x73,000 to 847x847
param_grid = {
    'k': [20, 30, 40],
    'sim_options': {
        'name': ['cosine', 'pearson_baseline'],
        'user_based': [False]  # <--- THIS FIXES THE MEMORY ERROR
    }
}

# Running GridSearchCV with Item-Based settings
gs_knn = SurpriseGridSearch(KNNBasic, param_grid, measures=['rmse', 'mae'], cv=3)
gs_knn.fit(data)

print(f"Best KNN RMSE: {gs_knn.best_score['rmse']:.4f}")

# Train on 80% / Evaluate on 20%
trainset, testset = surprise_split(data, test_size=0.20, random_state=42)
best_knn = gs_knn.best_estimator['rmse']
best_knn.fit(trainset)

# Evaluate
predictions = best_knn.test(testset)
knn_rmse = accuracy.rmse(predictions)
knn_mae = accuracy.mae(predictions)
```

Figure 4.10 KNN Collaborative Filter

The second model component is an item-based K-Nearest Neighbours (KNN) collaborative filter implemented using the Surprise library. The model consumes a (User, Restaurant, Rating) triplet DataFrame and builds a similarity matrix between restaurants rather than users.This design decision was made deliberately to reduce the similarity matrix size from approximately 73,000 × 73,000 (user-based) to 847 × 847 (item-based), thereby avoiding memory errors on typical training hardware. GridSearchCV explores combinations of neighbourhood size $k \in \{20, 30, 40\}$ and similarity metrics (cosine, pearson_baseline) using RMSE and MAE as evaluation metrics with 3-fold cross-validation. The best-performing configuration is then re-trained on 80% of the data and evaluated on a held-out 20% test set. During real-time prediction, the KNN model outputs an estimated rating between 1.0 and 5.0, which is normalised to [0, 1] and fused with the content-based score.

```
# --- HYBRID SCORE FUSION ---
# 🆕 Adaptive weighting: if user has favourites, make room for the boost
if has_favorites:
    # Weights sum to 1.0:  content 0.5, collab 0.2, favourites 0.3
    final_score = (0.5 * content_prob) + (0.2 * collab_pred) + (0.3 * fav_sim)
else:
    # Fall back to original weights when user has no favourites yet
    final_score = (0.7 * content_prob) + (0.3 * collab_pred)

print(f"DEBUG {name}: content={content_prob:.3f}, collab={collab_pred:.3f}, fav_sim={

scored_list.append({
    "name": name,
    "score": final_score,
    "content_prob": content_prob,
    "collab_pred": collab_pred,
    "fav_sim": fav_sim,                    # 🆕 expose for XAI
    "halal": csv_halal,
    "boosted_by_favorites": has_favorites  # 🆕 flag for UI/XAI
})
```

Figure 4.11 Hybrid Score Fusion Logic

The final hybrid score is computed in the RestaurantEngine.recommend() method using adaptive weighting. When the user has saved favourite restaurants, a taste vector is built by averaging the TF-IDF vectors of all favourites, and cosine similarity between each candidate and this taste vector yields a fav_sim score. In this case, the final score is 0.5 × content_prob + 0.2 × collab_pred + 0.3 × fav_sim. When the user has no saved favourites (a cold-start scenario), the system falls back to 0.7 × content_prob + 0.3 × collab_pred, putting more weight on content features which do not require prior user interactions. This adaptive design ensures that new users receive meaningful recommendations immediately while existing users benefit from increasingly personalised results as they save more favourites.

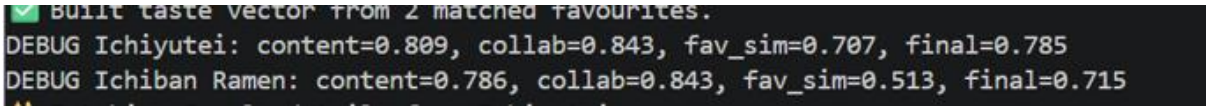


Figure 4.12 Hybrid Score for restaurant Ichiyutei and Ichiban Ramen

Figure 4.12 illustrates an example of the hybrid score computation captured from the backend terminal log during a real recommendation request. After the system has successfully built the taste vector from two matched favourites, two candidate restaurants are scored using the

adaptive weighting formula. Ichiyutei receives a content probability of 0.809, a collaborative prediction of 0.843, and a favourite similarity of 0.707, producing a final score of 0.785, while Ichiban Ramen receives the same content and collaborative scores but a lower favourite similarity of 0.513, resulting in a lower final score of 0.715. This example demonstrates that even when two candidates share identical content and collaborative scores, the favourite similarity component plays a decisive role in differentiating between them, with the candidate that more closely resembles the user's taste vector winning the top ranking. The output also confirms that the hybrid fusion formula 0.5 × content_prob + 0.2 × collab_pred + 0.3 × fav_sim is functioning correctly in the deployed system.

b) Performance Metric

Table 4.1 compares the performance of the individual SVM (linear kernel, C=1), Random Forest (100 estimators), and the final Ensemble (SVM + RF with Soft Voting) classifiers on the held-out 20% test set using four standard metrics: Accuracy, Precision, Recall, and F1-Score. These metrics assess how well each model classifies restaurant reviews as relevant (rating ≥ 4.0) versus not relevant. The comparison highlights the strengths and weaknesses of each model and confirms the benefit of combining them through soft voting.

Table 4.1 Performance Comparison of Content-Based Models

Machine Learning Model	Accuracy	Precision	Recall	F1-Score
SVM (linear kernel, C=1)	0.8721	0.9611	0.8782	0.9178
Random Forest (100 estimators)	0.8860	0.9434	0.9147	0.9288
Ensemble (SVM + RF, Soft Voting)	0.9063	0.9300	0.9568	0.9432

As shown in Table 4.1, the Ensemble model achieves the highest performance across almost every metric, with an accuracy of 90.63%, recall of 95.68%, and F1-score of 94.32%. The SVM classifier alone achieves the highest precision (96.11%), meaning that when SVM predicts a review as relevant, it is almost always correct.However, its lower recall (87.82%) indicates that it misses a number of genuinely relevant reviews. The Random Forest classifier strikes a more balanced compromise between precision (94.34%) and recall (91.47%), reaching an F1-score of 92.88%. By combining the two models through soft voting, the ensemble inherits SVM's precision while compensating for its lower recall using Random Forest's more flexible decision

boundaries, ultimately producing the best F1-score. This justifies the decision to deploy the ensemble as the content-based scoring component in the MakanBot recommendation engine.

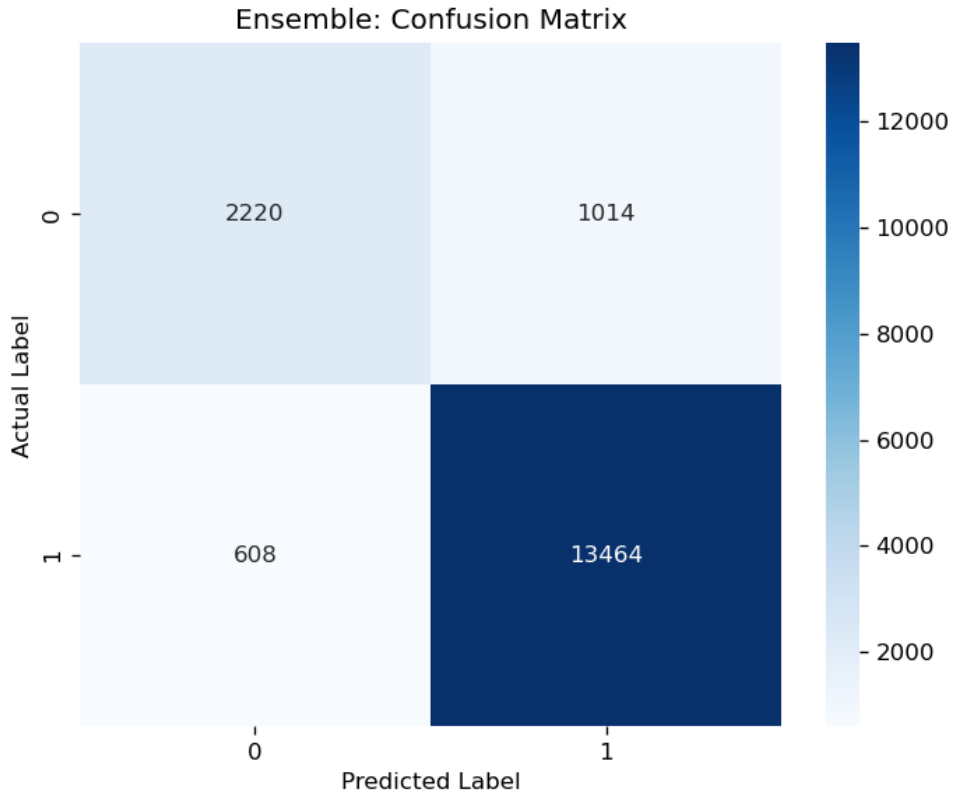


Figure 4.13 Confusion Matrix of the Ensemble Classifier

Figure 4.13 presents the confusion matrix of the Ensemble classifier on the test set. The matrix reveals the following outcomes:

- **True Negatives (TN) = 2,220** reviews correctly classified as *not relevant*
- **False Positives (FP) = 1,014** reviews incorrectly classified as *relevant* when they were actually not
- **False Negatives (FN) = 608** reviews incorrectly classified as *not relevant* when they were actually relevant
- **True Positives (TP) = 13,464** reviews correctly classified as *relevant*

Out of a total of 17,306 test samples, the model correctly classified 15,684 of them (TP + TN), giving the overall accuracy of 90.63%. The high TP count relative to FN confirms the model's strong recall of 95.68%, meaning it successfully identifies the vast majority of genuinely

relevant reviews. The relatively higher FP rate (1,014 compared to 608 FN) reflects a deliberate trade-off: because the MakanBot system's goal is to *surface* good restaurant candidates, occasionally flagging a borderline review as relevant (FP) is less harmful than missing a truly relevant review (FN). This behaviour is consistent with the use of class_weight='balanced' during training, which slightly biases the classifier towards the positive class to avoid the "high-accuracy trap" that arises from class imbalance in the dataset.

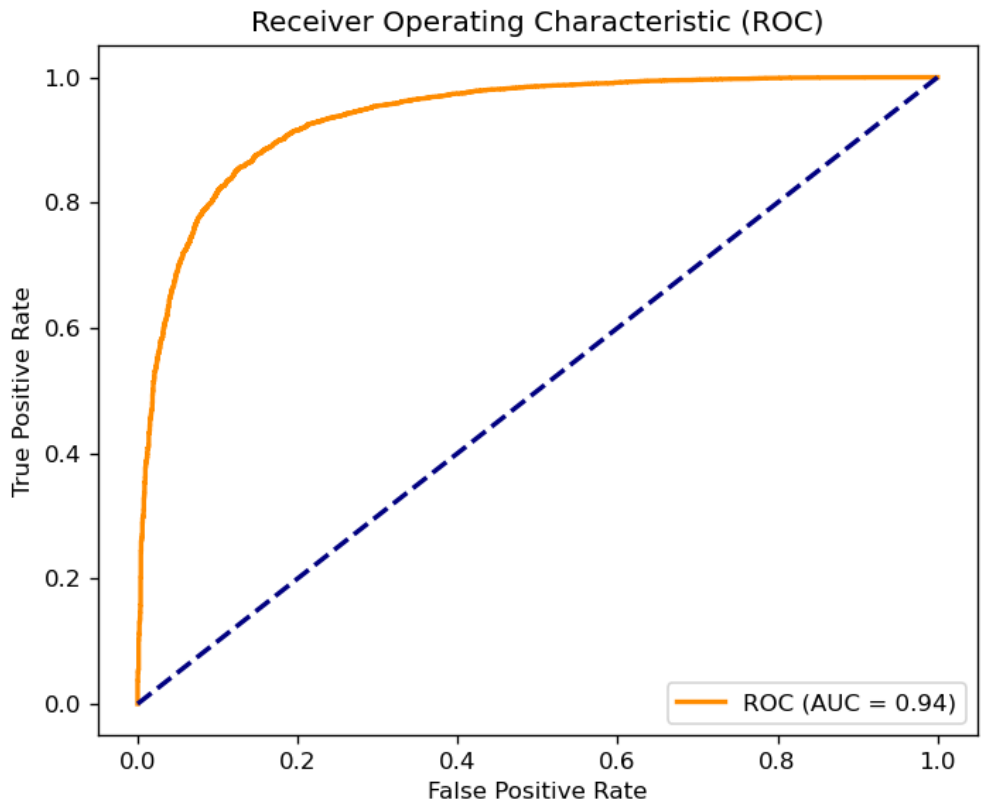


Figure 4.14 ROC Curve of the Ensemble Classifier

Figure 4.14 shows the Receiver Operating Characteristic (ROC) curve of the Ensemble classifier. The ROC curve plots the True Positive Rate (Recall) against the False Positive Rate across all possible probability thresholds, providing a threshold-independent view of the model's discrimination ability. The Area Under the Curve (AUC) value of 0.94 indicates that the ensemble has excellent class-separation capability,an AUC of 1.0 represents a perfect classifier, while 0.5 represents random guessing. The curve rises steeply toward the top-left corner, which is the ideal behaviour, confirming that the ensemble can achieve very high recall while keeping the false positive rate low. This high AUC also means that the probability scores

output by ensemble.predict_proba(...)[:, 1] which are consumed by the hybrid fusion formula as content_prob are well-calibrated and can be meaningfully combined with the collaborative and favourite-similarity scores as described earlier in the Hybrid Score Fusion subsection (Figure 4.11).

Table 4.2 Performance of KNN Collaborative Filter (Item-Based)

Metric	Value
Best k (neighbourhood size)	40
Best similarity metric	pearson_baseline
Matrix orientation	Item-based (user_based = False)
Best RMSE (3-fold GridSearchCV)	1.0861
Test Set RMSE	1.0822
Test Set MAE	0.8210

Table 4.2 summarises the performance of the KNN collaborative filter evaluated on the held-out 20% test set. The best configuration found through 3-fold GridSearch cross-validation used k = 40 neighbours with the pearson_baseline similarity metric, operating in item-based mode (comparing restaurants rather than users). The item-based orientation was chosen deliberately to reduce the similarity matrix size from approximately 73,000 × 73,000 (user-based) to 847 × 847 (item-based), thereby avoiding memory errors during training without sacrificing recommendation quality.

The final test RMSE of 1.0822 and MAE of 0.8210 indicate that, on a rating scale of 1 to 5, the KNN model's predicted rating deviates from the actual rating by approximately 1.08 stars (root mean square error) and 0.82 stars (mean absolute error) on average. While these numbers are acceptable for collaborative filtering on a sparse restaurant-review dataset, they are not accurate enough to serve as a standalone recommendation signal. This empirical finding directly motivates the hybrid design philosophy of MakanBot: the KNN model alone cannot reliably predict user preferences, but when fused with the much stronger content-based ensemble (F1 = 0.9432, AUC = 0.94) and the favourite-similarity cosine score, it provides a valuable *personalisation* signal that captures latent user–restaurant interaction patterns which content features alone cannot represent. This is reflected in the hybrid fusion weight assignment 0.5·content_prob + 0.2·collab_pred + 0.3·fav_sim, where the collaborative score receives a

deliberately smaller weight (0.2) than the content score (0.5), acknowledging both its personalisation value and its lower standalone accuracy.

4.5 Model Deployment and Gemini XAI Integration

After the machine learning models had been trained and evaluated as described in Section 4.4, the next step was to deploy them into the backend service and integrate them with the Google Gemini API to enable explainable recommendation generation. Both deployment and Gemini XAI integration were carried out successfully, and their working state can be observed in the live system walkthrough presented in Chapter 5.

To preserve the trained state of each model, three Python pickle files were generated, namely tfidf_vectorizer.pkl for the TF IDF vectoriser, content_ensemble.pkl for the content based ensemble classifier, and knn_model.pkl for the K Nearest Neighbours collaborative filter. These files capture the trained parameters, vocabulary, and learned similarity matrices, allowing the models to be reloaded later without retraining. Pickle was selected as the serialisation format because it is the standard mechanism in the Python ecosystem for persisting scikit learn and Surprise library objects. All three files were placed inside the models/ folder of the FastAPI backend and were successfully loaded into memory once when the server starts up, which avoids the cost of reloading on every chat request and keeps response times low. The successful loading of the models is verified in the backend startup log shown in Chapter 5.

The Google Gemini API was integrated to provide explainable AI (XAI) for every recommendation served by the system. After the hybrid recommendation engine has selected the top scoring restaurant, the backend constructs a structured prompt and sends it to the Gemini model gemini-3.1-flash-lite-preview via the Google generative AI client. The prompt is dynamically built using contextual information about the user, including their saved cuisine preferences, the number of favourite restaurants they have stored, the original chat message they sent, whether the halal filter is active, the name and cuisine of the recommended restaurant, and the actual scoring breakdown reported by the engine such as the cuisine match percentage and the favourite similarity percentage. Conditional instructions are also included in the prompt to guide the explanation style, for example to mention halal certification when the halal filter is active, or to mention similarity to saved favourites when the favourite similarity score exceeds 40%. Gemini then returns a single friendly sentence of no more than 25 words, which

is displayed to the user inside the recommendation card under the heading "WHY MAKANBOT RECOMMENDS THIS". The successful integration of this XAI layer can be observed in the actual chat responses and restaurant detail screens shown in Chapter 5.

In summary, the deployment of the trained machine learning models into the FastAPI backend and the integration of the Google Gemini API for explainable AI were both implemented successfully. The combination of pickle based model serving and dynamic Gemini prompting transforms the system from a black box recommender into a transparent and responsive assistant. The live operation of these components, including model loading at server startup, real time score computation, Gemini XAI generation, and the resulting user facing explanations, are demonstrated in detail throughout Chapter 5.

Chapter 5

System Implementation

5.1 Hardware and Software Setup

5.1.1 Hardware

The hardware utilized in this project consists of a computer and an Android mobile device. The computer serves as the primary platform for data preprocessing, model training, system development, and the integration of both the sentiment analysis and recommendation modules. Meanwhile, the Android device is employed for testing and deploying the chatbot application, enabling users to receive restaurant recommendations directly on their mobile platform.

Table 5.1 Specifications of laptop

Description	Specifications
Model	ASUS TUF Gaming F15 FX507ZE
Processor	Intel Core i7-12700H
Operating System	Windows 11
Graphic	NVIDIA GeForce RTX 3050 Ti Laptop GPU (4GB GDDR6)
Memory	16GB DDR5 RAM
Storage	512GB NVMe PCIe SSD

Table 5.2 Specifications of Mobile Device

Description	Specifications
Model	Redmi Note 14 Pro 5G
Operating System	Android 14 (HyperOS 1.0)
Processor	MediaTek Dimensity 7300-Ultra, Octa-core, up to 2.5 GHz
Memory	12GB RAM + 6GB Virtual RAM (Total 18GB)
Storage	512GB internal storage

5.1.2 Software

Before initiating the project development, three software tools needed to be set up on the laptop.

Table 5.3 Software Needed

Software	Description	Function
Jupyter Lab	A web-based interactive development environment for notebooks, code, and data.	Used for the experimentation phase: cleaning the reviews datasets and training/testing the SVM, Random Forest, and KNN models.
Chatgpt	A large language model by OpenAI.	Used as a development assistant for troubleshooting code, refining thesis documentation, and brainstorming system logic.
Android Studio Hedgehog	The official IDE for Android development.	Used to build the mobile frontend, including the chatbot interface, sidebars, and CardView layouts.
Drawio	A free online diagram software.	Used to design the System Block Diagram and architecture flowcharts for the thesis report.
Canva	A graphic design platform.	Used for high-fidelity UI design mockups, as well as creating the project poster and presentation slides.
Flask	A modern, high-performance Python web framework for building APIs	Serves as the backend 'Orchestrator' that hosts the /chat endpoint, handles API requests from the mobile app, and integrates the ML models with the Gemini and Google Places APIs.
Anaconda Navigator	A desktop GUI for managing Python distributions.	Used to manage virtual environments and ensure all libraries (Scikit-Learn, Surprise, Pandas) are properly installed.

Firebase	A Google-backed cloud platform for mobile applications	Manages real-time user data, including Authentication (Login/Signup) and storing Saved/Liked restaurants.
Google Places API	A cloud-based service from Google that provides access to detailed information about places, businesses, and points of interest worldwide.	Used in the recommendation engine to enrich each recommended restaurant with live metadata including photos, formatted address, user rating, total review count, official website (menu link), and reservation URL so that the mobile app can display a visually rich and up-to-date restaurant card.
Gemini API	Google's advanced generative AI model.	Powers the chatbot's natural language understanding, allowing it to converse with users and extract their dining preferences.
Surprise Library	An open-source Python library specifically designed for building and analysing collaborative filtering recommender systems.	Used to implement and train the item-based K-Nearest Neighbours (KNN) collaborative filter, including hyperparameter tuning through GridSearchCV and evaluation using RMSE and MAE metrics.
Visual Studio Code	A versatile, lightweight code editor by Microsoft.	Used for writing and debugging the Flask backend (app.py) and managing the project's Python scripts and environment files.
Claude AI	A large language model assistant developed by Anthropic, accessible through a web based interface.	Used as a development assistant for technical brainstorming, drafting and refining the project

		report, and generating system diagrams such as the system architecture diagram, use case diagram, activity diagram, and system block diagram in SVG format.

5.2 Setting and Configuration

Before the MakanBot system can be operated, three separate environments must be set up: the Python backend (FastAPI server + ML models), the Firebase cloud project (authentication and Firestore database), and the Android mobile application. This section documents the step-by-step setup procedure.

5.2.1 Backend Environment Setup

The backend is a Python 3.11 project that runs on the developer's laptop. The following steps were performed to prepare it:

1. **Install Anaconda and create a virtual environment**

   an isolated Python environment was created using Anaconda Navigator to avoid conflicts with system-wide libraries.

2. **Install Python dependencies**

   the required libraries are installed using pip install inside the virtual environment:

```
PS S:\IMPORTANT\FYP-NEW\RestaurantChatbot\Backend_New> pip install fastapi uvicorn firebase-admin google-genai pandas numpy scikit-learn nltk scipy scikit-surprise requests
```

Figure 5.1 Install Required Libraries

3. **Place the trained models**

   The three pre-trained .pkl files generated in Chapter 4 (tfidf_vectorizer.pkl, content_ensemble.pkl, knn_model.pkl) are placed in the models/ folder of the backend project.

4. **Place the cleaned dataset**

   The processed Master_Restaurants_Profile_Updated.csv file is placed in the backend root folder.

5. **Obtain a Firebase Service Account Key**

   A serviceAccountKey.json file is downloaded from the Firebase Console (Project Settings → Service Accounts → Generate New Private Key) and placed in the backend

root folder. This credential allows the FastAPI server to access Firestore on behalf of the admin.

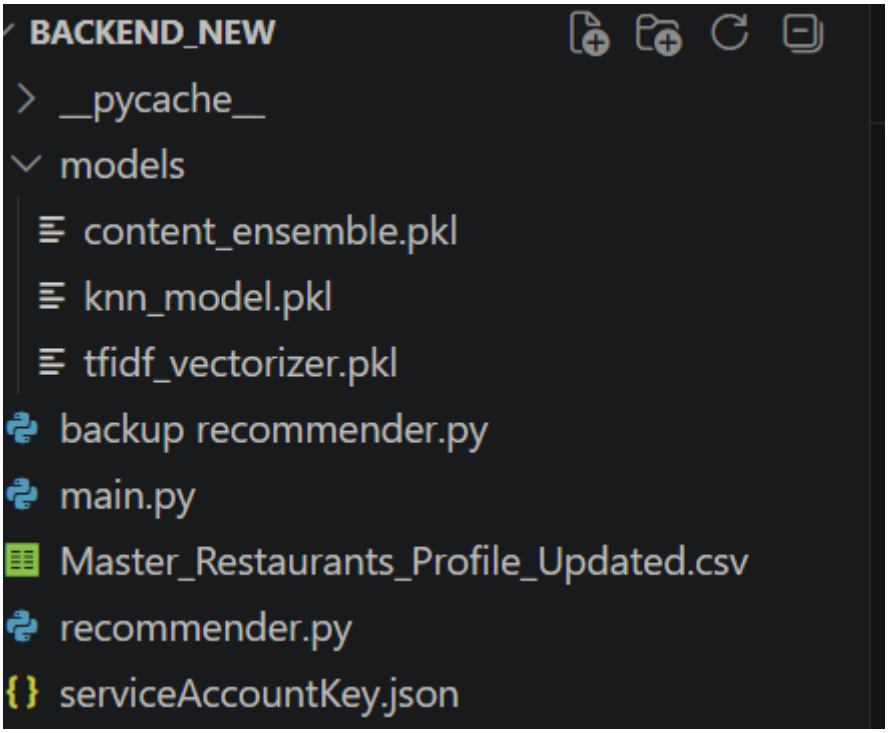


Figure 5.2 Structure of File

6. **Obtain a Gemini API Key**

   A Gemini API key is obtained from Google AI Studio and is stored in each user's Firestore document under the gemini_api_key field.

7. **Start the FastAPI server**

   the server is launched using Uvicorn with hot-reload enabled

```
PS S:\IMPORTANT\FYP-NEW\RestaurantChatbot\Backend_New> uvicorn main:app --host 0.0.0.0 --port 8000 --reload
INFO:     Will watch for changes in these directories: ['S:\\IMPORTANT\\FYP-NEW\\RestaurantChatbot\\Backend_New']
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [41404] using StatReload
```

Figure 5.3 Uvicorn Server Startup

Figure 5.3 shows the terminal output when the Uvicorn server is started. The server binds to port 8000 on all network interfaces, enabling both local testing and remote mobile device access over LAN.

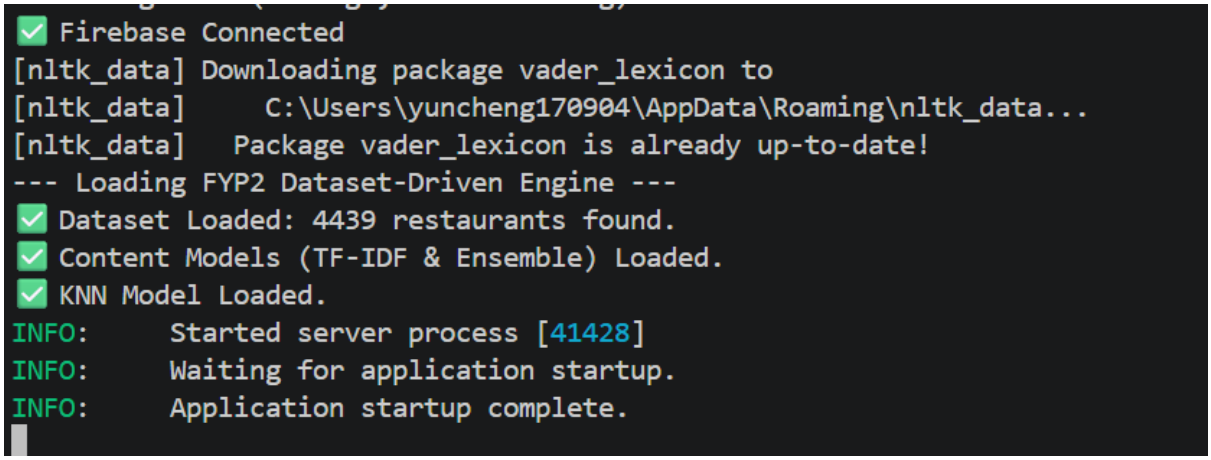


Figure 5.4 Backend Initialisation

Figure 5.4 shows the subsequent startup process of the application. During initialisation, the backend successfully connects to Firebase, downloads the VADER sentiment lexicon, and loads the full dataset containing 4,439 restaurants. All three trained machine learning components which is the TF-IDF vectoriser, the content-based ensemble classifier, and the KNN collaborative filter are deserialised from their .pkl files and made ready to serve recommendation requests in real time.

5.2.2 Firebase Cloud Project Setup

The Firebase project was created through the Firebase Console (https://console.firebase.google.com). The following services were enabled:

1. **Firebase Authentication**

   Email/Password sign-in method was enabled to support user registration and login from the mobile app.

2. **Cloud Firestore**

   a Firestore database was created in Native mode. Two top-level collections were designed:

   - users/{uid} : stores each user's profile, preferences, and Gemini API key
   - users/{uid}/favorites/{restaurant_name}: a subcollection storing each user's saved favourite restaurants

3. **Security Rules**

   Firestore security rules were configured so that each user can only read and write their own documents, preventing unauthorised access to other users' data.

5.2.3 Android Application Setup

The mobile application is a native Android app developed in Java using Android Studio Hedgehog. The setup steps are:

1. Create a new Android project targeting minimum SDK 26 (Android 8.0) with Java as the programming language.
2. Add Firebase to the project
   the google-services.json file is downloaded from the Firebase Console and placed in the app/ folder. The Firebase Authentication and Firestore SDK dependencies are added to build.gradle.
3. Add Retrofit for HTTP communication
   Retrofit and Gson converters are added to handle JSON-based communication with the FastAPI backend.
4. Add Glide for image loading
   Glide is added to handle the asynchronous loading and caching of restaurant photos returned from the Google Places API.
5. Build and install the APK on the physical Android device (Redmi Note 14 Pro 5G) for testing.

5.3 System Operation

This section walks through the complete user journey, demonstrating each screen of the MakanBot application in the order a new user would experience them.

5.3.1 Splash Screen and Authentication

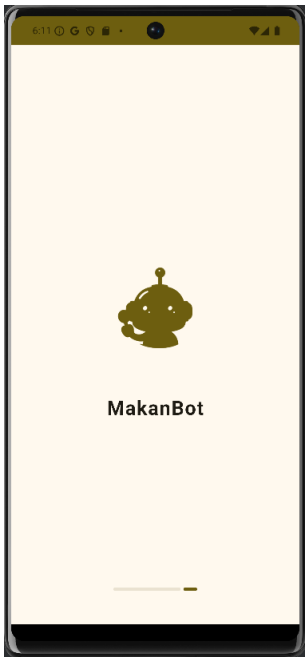


Figure 5.5 Splash Screen

When the application is launched, users are greeted with the splash screen shown in Figure 5.5, featuring the MakanBot logo. After a brief loading period, the app automatically checks the Firebase authentication state and redirects the user either to the login screen (for unauthenticated users) or directly to the main chat screen (for users who are already logged in).

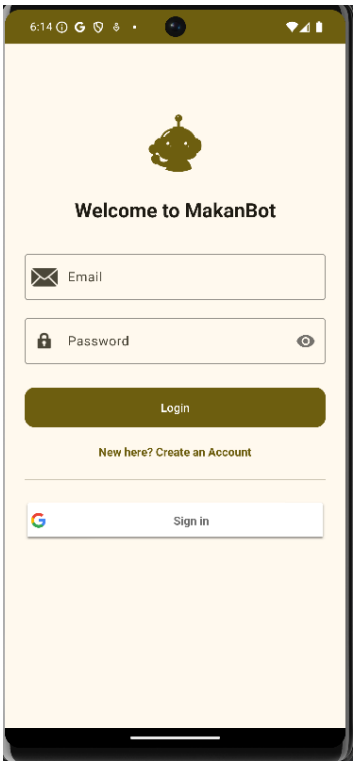


Figure 5.6 Login Screen

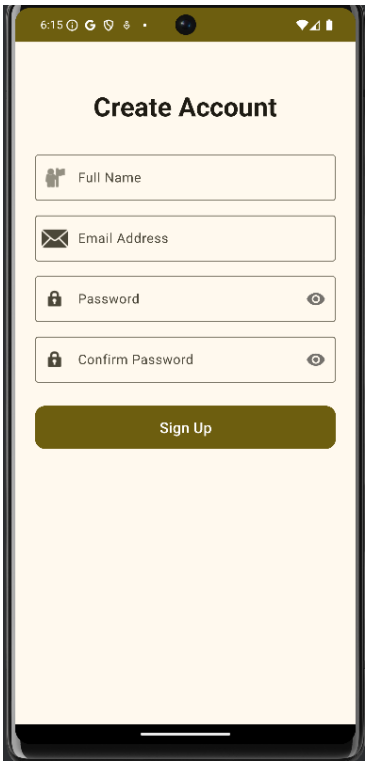


Figure 5.7 Signup Screen

New users are presented with the login screen (Figure 5.6), which provides fields for entering an email and password. A "New here? Create an Account" link routes the user to the signup screen (Figure 5.7), where they can create a new account by providing their full name, email, and password.

Upon successful registration, Firebase Authentication creates a new user record and the app generates a corresponding document in the Firestore users collection. The user is then directed to the preferences screen.

5.3.2 Preferences Selection

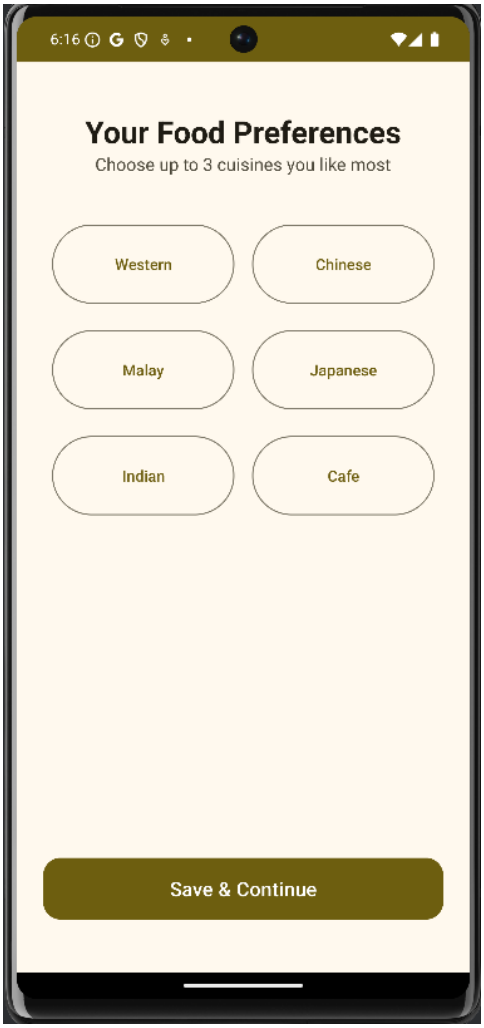


Figure 5.8 Cuisine Preference Selection

Figure 5.8 shows the preferences screen, where the new user is asked to select up to three cuisines they like most from the six available categories (Western, Chinese, Malay, Japanese, Indian, Cafe). The selected cuisines are stored in the user's Firestore document under the preference array field and are used later in the vague-query fallback logic described in Chapter 3.

5.3.3 Chat Interface

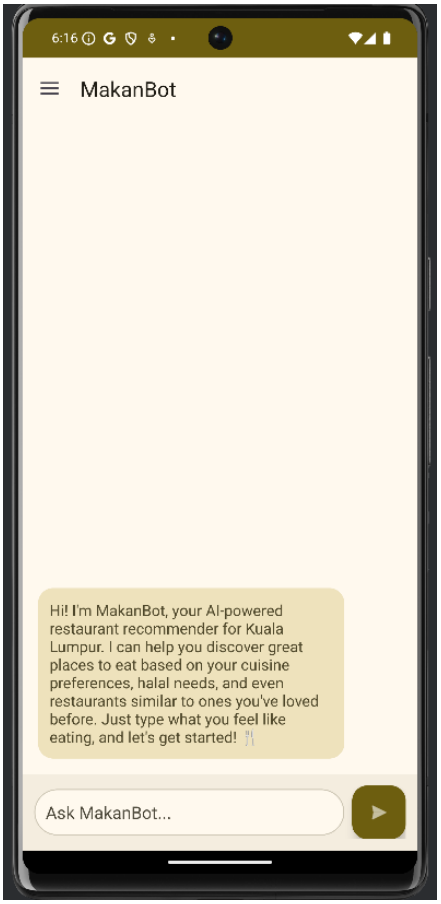


Figure 5.9 Empty Chat Interface with Welcome Message

After saving preferences, the user is routed to the main chat screen (Figure 5.9). On first launch, MakanBot sends an automatic welcome message introducing itself as an AI-powered restaurant recommender for Kuala Lumpur. The input bar at the bottom of the screen allows the user to type natural-language queries such as *"I want Japanese food"*, *"I'm hungry"*, or *"any halal food"*.

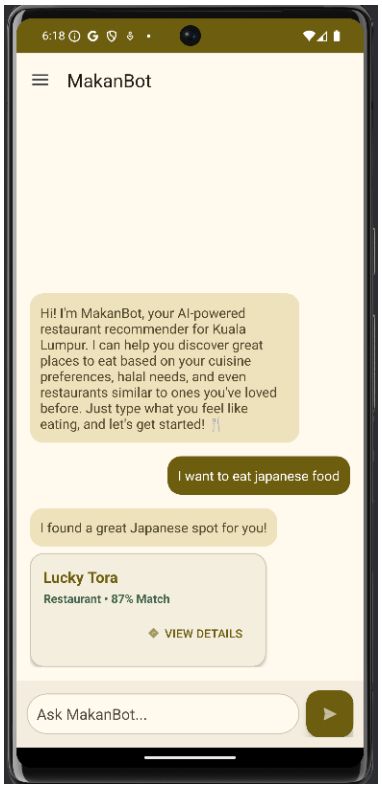


Figure 5.10 Chat Response with Japanese Recommendation

Figure 5.10 shows an example interaction where the user types *"I want to eat japanese food"*. The backend processes the request, invokes the hybrid recommendation engine, and returns a restaurant card displaying the recommended restaurant name (Lucky Tora), type, and a match percentage score (87%) that reflects the hybrid fusion score.

5.3.4 Restaurant Detail View

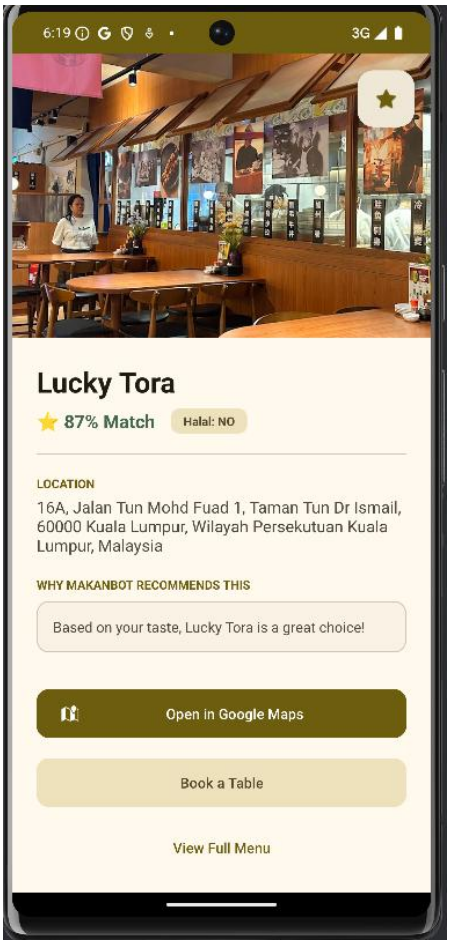


Figure 5.11 Restaurant Detail View (Lucky Tora)

Tapping the restaurant card opens the restaurant detail screen (Figure 5.11), which presents a visually rich view including:

- A high-resolution photo fetched from the Google Places API
- The restaurant's name, match percentage, and halal badge
- Full formatted address from Google Places
- The "WHY MAKANBOT RECOMMENDS THIS" section, which displays the one-sentence explanation generated by the Gemini XAI module
- Three action buttons: Open in Google Maps, Book a Table, and View Full Menu
- A bookmark icon at the top-right to save the restaurant to favourites

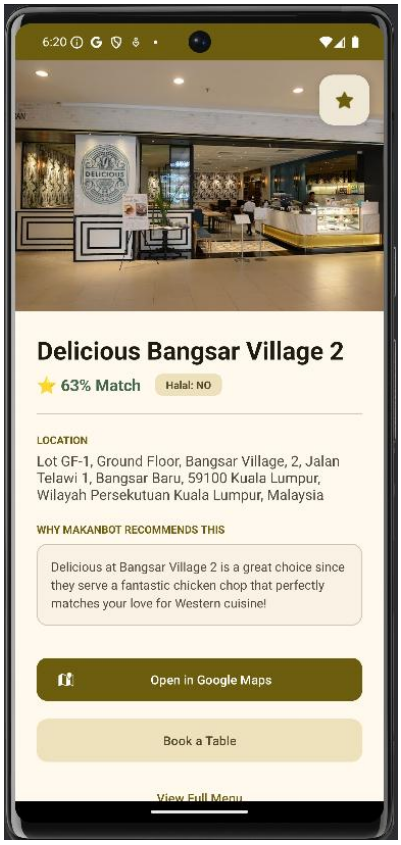


Figure 5.12 Restaurant Detail View with XAI Explanation

Figure 5.12 shows another example (Delicious Bangsar Village 2) where the XAI explanation demonstrates context awareness. The recommendation was generated for a user who requested *"chicken chop"* which is a Western dish. Gemini's XAI output states: *"Delicious at Bangsar Village 2 is a great choice since they serve a fantastic chicken chop that perfectly matches your love for Western cuisine!"* This confirms that the improved intent-analysis prompt correctly infers *chicken chop → Western cuisine* at the semantic level.

5.3.5 Navigation Drawer and Secondary Screens

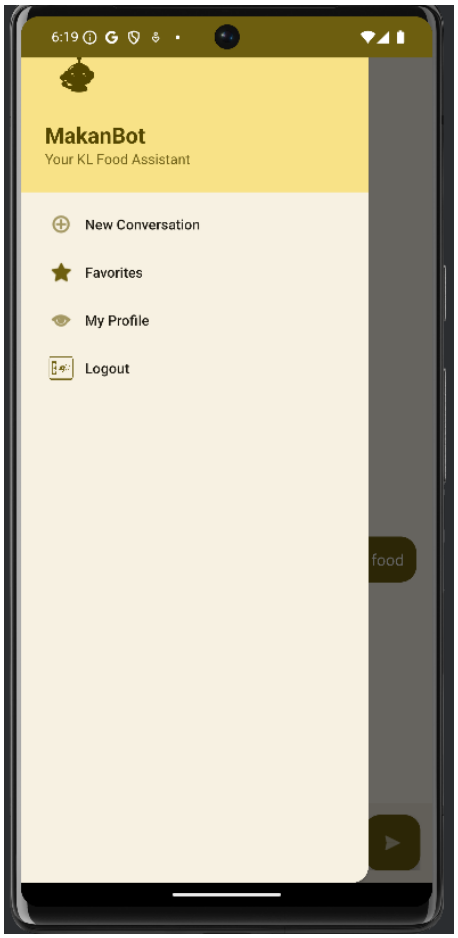


Figure 5.13 Navigation Drawer Menu

The navigation drawer (Figure 5.13), accessible by tapping the hamburger menu icon in the top-left corner, provides access to four options: New Conversation, Favorites, My Profile, and Logout.

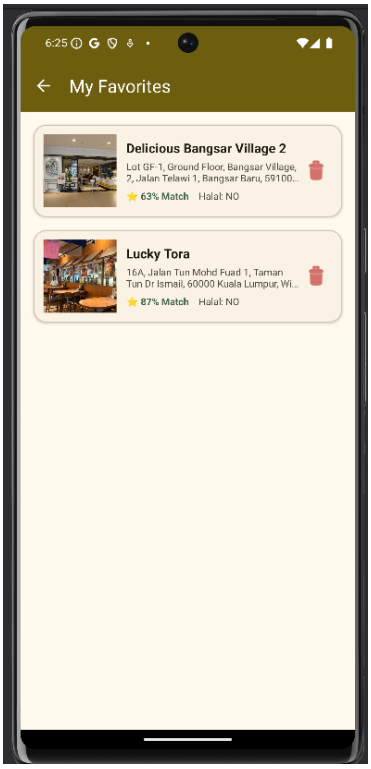


Figure 5.14 My Favorites List

Figure 5.14 shows the My Favorites screen, which lists every restaurant the user has saved. Each entry displays the restaurant's photo, name, address, match percentage, and halal status. A trash-can icon allows the user to remove an individual restaurant from their favourites. The favourites stored here are precisely the same data used by the hybrid engine's taste-vector computation described in Chapter 3.

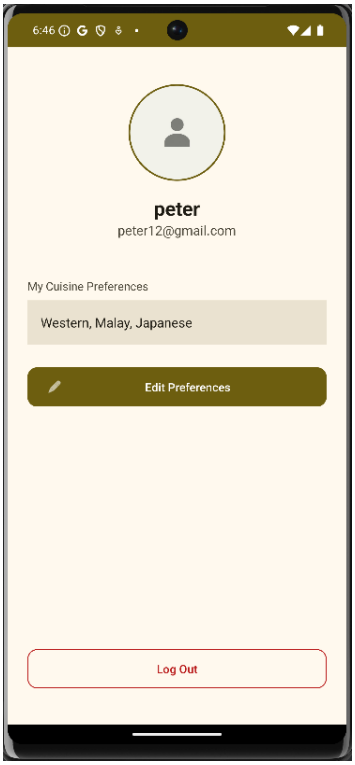


Figure 5.15 Profile Screen

Figure 5.15 shows the My Profile screen, which displays the user's avatar, name, email, and current cuisine preferences. An Edit Preferences button launches the preferences selection screen in edit mode (described in Section 5.2), allowing users to update their cuisine choices at any time. A Log Out button at the bottom signs the user out of the app and returns them to the login screen.

5.4 Backend and Data Flow

This section illustrates what happens behind the scenes when a user interacts with the application. While Section 5.3 focused on the user-facing screens, Section 5.4 documents the corresponding server-side and database operations.

5.4.1 Firebase Authentication and Firestore Data

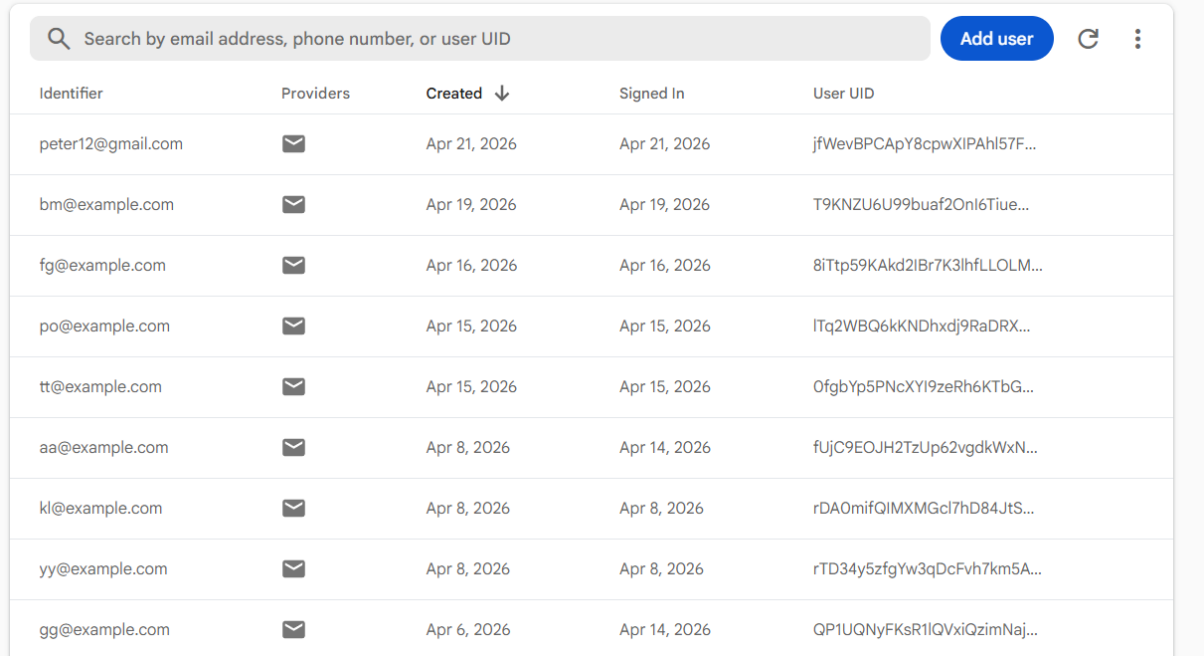


Figure 5.16 Firebase Authentication User List

Figure 5.16 shows the Firebase Authentication dashboard listing all registered users. Each entry displays the email address, sign-in provider (Email/Password), creation date, last sign-in date, and the unique user identifier (UID) assigned by Firebase. The UID is the primary key used throughout the system to associate user data in Firestore.

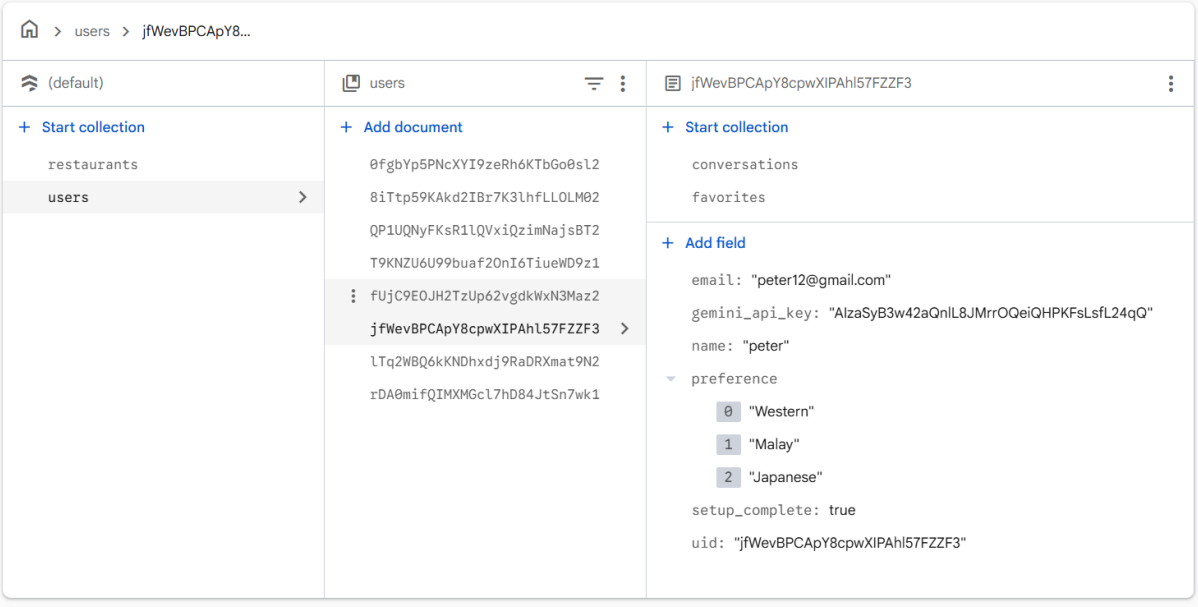


Figure 5.17 Firestore User Document Structure

Figure 5.17 shows a user document in the users collection of Cloud Firestore. The document uses the Firebase Authentication UID as its document ID and contains the following fields:

- **email**
  the registered email address
- **gemini_api_key**
  the Gemini API key used for intent analysis and XAI generation
- **name**
  the user's display name
- **preference**
  an array of selected cuisine preferences (e.g., ["Western", "Malay", "Japanese"])
- **setup_complete**
  a boolean flag indicating whether the user has completed preference selection
- **uid**
  the Firebase Authentication UID (duplicated here for convenience)

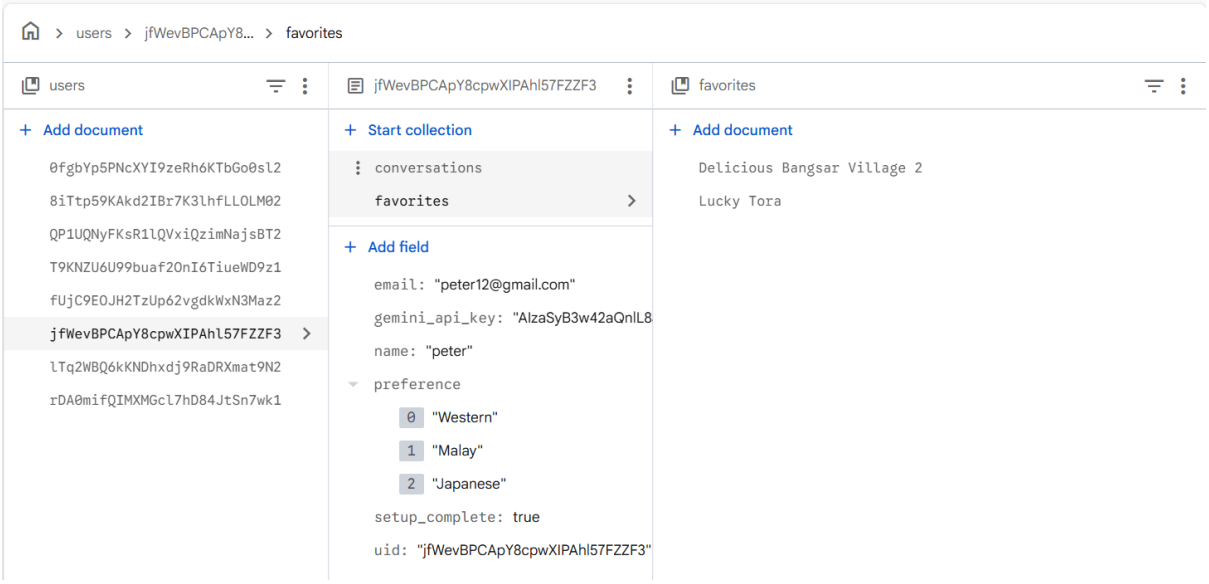


Figure 5.18 Firestore Favorites Subcollection

Figure 5.18 shows the favorites subcollection nested under each user document. Each favourite restaurant is stored as its own document, with the restaurant's display name used as the document ID (e.g., Delicious Bangsar Village 2, Lucky Tora). This subcollection design allows the backend to efficiently retrieve a user's favourites with a single query and is directly consumed by the get_user_favorites() function in main.py to build the taste vector for the favourite-similarity boost.

5.4.2 Live Backend Log Trace

When a user sends a chat message, the FastAPI backend prints a detailed trace of every step to the terminal. Figures 5.19 and 5.20 capture two such log extracts during a typical request.

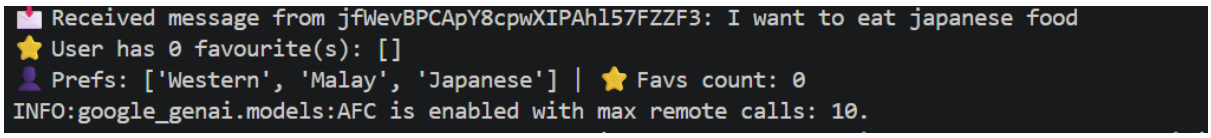


Figure 5.19 Incoming Chat Request

Figure 5.19 shows the beginning of a request. The backend receives the message *"I want to eat japanese food"* from a user identified by their Firebase UID (jfWevBPCApY8cpwXIPAhl57FZZF3). It then fetches the user's profile from Firestore, confirming that the user has saved preferences ['Western', 'Malay', 'Japanese'] but currently has 0 favourites.

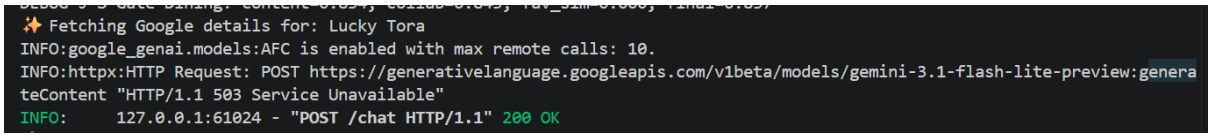


Figure 5.20 Google Places Fetch and XAI Response

Figure 5.20 shows the tail end of the same request, where the backend fetches restaurant metadata from the Google Places API (including photos, address, and rating), invokes the Gemini XAI module (one remote call to generativelanguage.googleapis.com), and finally returns a 200 OK response to the mobile app.

5.5 Recommendation Scenarios

To validate that the hybrid recommendation engine behaves correctly across different input patterns, three distinct scenarios were tested and recorded. Each scenario activates a different combination of the engine's internal mechanisms described in Chapter 3.

5.5.1 Scenario 1 : Vague-Query Fallback with Favourite-Based Personalisation

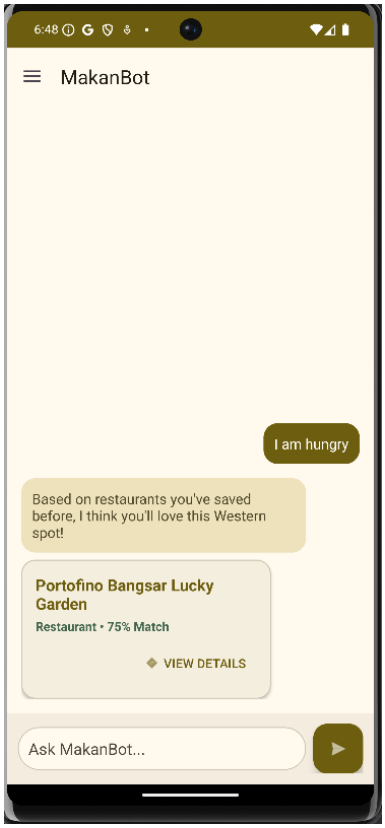


Figure 5.21 Vague Query with Favourite-Based Personalisation

Figure 5.21 shows the user typing an extremely vague query: *"I am hungry"*. Since no cuisine or dish is mentioned, the intent analyser returns cuisine = "ANY", which triggers the vague-query fallback logic. The backend randomly selects a cuisine from the user's saved preferences (Western, in this case) and proceeds to generate a recommendation. Because the user has previously saved favourites, the favourite-similarity boost is also active, and the bot replies: *"Based on restaurants you've saved before, I think you'll love this Western spot!"* followed by a Portofino Bangsar Lucky Garden card with a 75% match score.

5.5.2 Scenario 2 : Halal Filter with Favourite-Based Personalisation

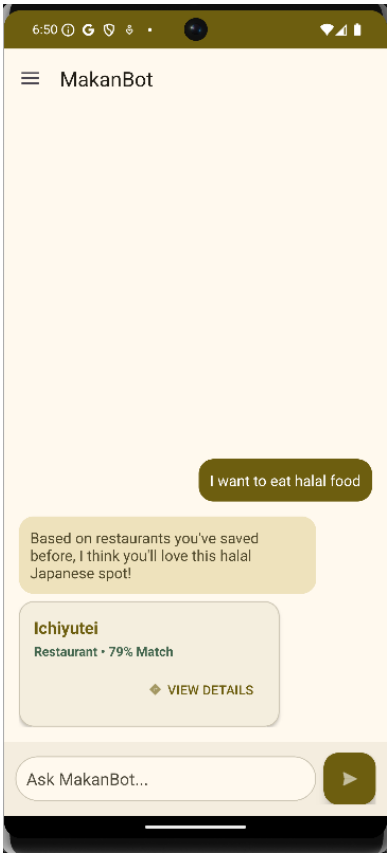


Figure 5.22 Halal Filter with Favourite-Based Personalisation

Figure 5.22 shows a more advanced scenario: the user types *"I want to eat halal food"*. This query does not explicitly mention a cuisine, so the vague-query fallback again selects from the user's saved preferences .For this time, Japanese. However, the intent analyser also detects the word *"halal"* and sets halal_required = true, activating the halal filter in the recommendation engine. The engine then searches only for halal-certified Japanese restaurants and returns Ichiyutei with a 79% match score. The bot's response acknowledges both constraints: *"Based on restaurants you've saved before, I think you'll love this halal Japanese spot!"*

5.5.3 Backend Hybrid Scoring Breakdown

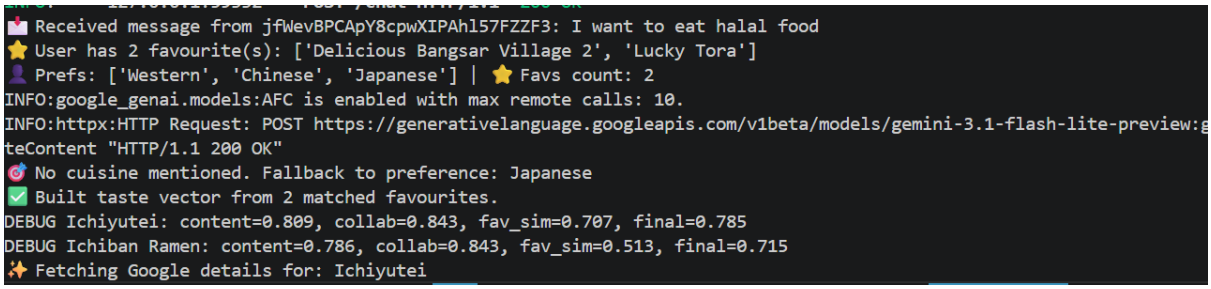


Figure 5.23 Hybrid Scoring Breakdown

Figure 5.23 captures the backend log corresponding to the halal query in Scenario 2. This screenshot is particularly valuable because it exposes the internal hybrid scoring process that is normally hidden from users:

- The user has 2 favourites saved (Delicious Bangsar Village 2, Lucky Tora)
- The vague-query fallback triggers with the cuisine set to Japanese (chosen from the user's preferences)
- A taste vector is built from the 2 matched favourites (confirming the favourite-similarity boost is active)
- Two candidate restaurants are scored:
    - Ichiyutei: content = 0.809, collab = 0.843, fav_sim = 0.707 → final = 0.785
    - Ichiban Ramen: content = 0.786, collab = 0.843, fav_sim = 0.513 → final = 0.715
- Ichiyutei wins because its fav_sim (0.707) is much higher, indicating its TF-IDF vector is more similar to the user's saved taste vector

This log trace provides concrete empirical evidence that all three signals which are content probability, collaborative prediction, and favourite similarity are being computed and fused correctly according to the weighted formula 0.5 × content_prob + 0.2 × collab_pred + 0.3 × fav_sim described in Chapter 4.

5.6 Implementation Issues and Challenges

Several technical challenges were encountered during implementation. Each was diagnosed and resolved with a deliberate design decision.

a) Memory Overflow in Collaborative Filtering

The initial implementation of the KNN collaborative filter used user-based similarity, which required building a similarity matrix of approximately 73,000 × 73,000 entries. This caused memory overflow errors during training. The issue was resolved by switching to item-based similarity (user_based = False), which reduced the matrix size to 847 × 847, a reduction of more than 99%.

b) Class Imbalance in Review Labels

A naïve training attempt on the ensemble classifier initially produced very high but misleading accuracy because the dataset is heavily biased toward positive reviews (ratings ≥ 4). The classifier could simply predict "positive" for everything and still score well. This "high-accuracy trap" was resolved by applying class_weight='balanced' to both the SVM and Random Forest components during training.

c) LLM Literal vs. Semantic Understanding

During user testing, the bot failed to recognise dish-level queries such as *"I want chicken chop"* because the Gemini intent-analysis prompt only checked for literal cuisine keywords. The fix was to expand the intent-analysis prompt into a few-shot prompt with an explicit dish-to-cuisine mapping table, enabling Gemini to semantically infer *chicken chop* → *Western*, *nasi lemak* → *Malay*, and so on.

d) Halal Awareness in Bot Responses

A similar issue was discovered in the final-response generation: when users queried *"any halal food"*, the bot would correctly filter halal restaurants but still reply with the generic "you didn't mention a cuisine" message, which confused users. This was resolved by adding halal-awareness branches to each of the four response templates in main.py Section 7, producing context-appropriate responses such as *"Based on your preference for Western food, here's a halal spot you might like!"*

e) Cold-Start Handling for New Users

New users who have not yet saved any favourites present a cold-start problem for the favourite-similarity boost. Without favourites, the cosine similarity cannot be computed at all. This was solved by introducing adaptive weighting in recommender.py: when a user has no favourites, the fusion weights automatically shift from 0.5/0.2/0.3 to 0.7/0.3, redirecting the removed fav_sim weight back into the content probability score. This ensures that even brand-new users receive meaningful recommendations on their first request.

f) Gemini API Rate Limiting and 503 Errors

During high-volume testing, occasional 503 Service Unavailable errors were observed from the Gemini API (visible in Figure 5.18). To keep the system robust, try/except blocks were wrapped around every Gemini call, with a deterministic fallback explanation generated locally when the API fails. This guarantees that the user always receives a meaningful recommendation card, even if the external LLM service is temporarily unavailable.

5.7 Concluding Remarks

This chapter has demonstrated the complete end-to-end operation of the MakanBot system. Section 5.1 documented the hardware and software environment. Section 5.2 detailed the step-by-step setup procedure for the backend, Firebase cloud project, and Android application. Section 5.3 walked through the user-facing screens in the order a new user would experience them, from splash screen and signup through to the chat interface, restaurant detail, favourites, and profile. Section 5.4 provided a behind-the-scenes view of the authentication, Firestore data structure, and live backend logs. Section 5.5 validated three realistic recommendation scenarios, namely vague-query fallback, halal filtering, and favourite-based personalisation, and confirmed through backend log inspection that the hybrid scoring formula operates correctly in production. Finally, Section 5.6 enumerated six notable implementation challenges and explained how each was resolved through explicit design decisions. Together, these sections confirm that the MakanBot system, as designed in Chapters 3 and 4, has been successfully implemented as a working, responsive, and intelligent mobile chatbot ready for evaluation.

Chapter 6

System Evaluation and Discussion

6.1 System Testing Overview

This chapter presents the testing and evaluation phase of the MakanBot system. Testing is a critical stage of software development that verifies whether the implemented system behaves according to its design and meets the needs of its intended users. Two complementary testing strategies were adopted in this project. The first is functional chatbot testing, which is a white-box, developer-driven procedure that systematically verifies every intelligent behaviour of the chatbot using a controlled set of 32 test cases. The second is a pilot user study, which collects feedback from real users through a structured Google Form survey to evaluate the usability, recommendation quality, and overall perception of the system from an end-user perspective. Together, these two evaluation methods provide a balanced view of both the technical correctness and the user acceptance of the system. Section 6.2 documents the functional testing results. Sections 6.3 to 6.6 present the pilot user study methodology, demographic context, quantitative results, and qualitative feedback. Section 6.7 concludes with an overall discussion of findings.

6.2 Functional Chatbot Testing

To verify that the MakanBot chatbot performs as designed, a total of **32 test cases** were executed across six functional categories. Each test case evaluates a specific aspect of the system's intelligent behaviour, ranging from intent classification and cuisine inference to halal filtering, personalisation, and session memory. Tables 6.1 through 6.6 present the test case results for each category.

6.2.1 Category A: Intent Classification (Chitchat vs Recommendation)

This category verifies that the Gemini-powered intent analyser correctly distinguishes food-related queries from non-food-related queries (chitchat).

Table 6.1 Test Cases for Intent Classification

Test ID	Input Prompt	Expected Output	Actual Output	Status
TC01	"Hello"	Chitchat reply, no recommendation	Returned chitchat reply	Pass
TC02	"What's the weather today?"	Chitchat reply, no recommendation	Returned chitchat reply	Pass
TC03	"Can you book me a hotel?"	Chitchat reply, no recommendation	Returned chitchat reply	Pass
TC04	"Tell me a joke"	Chitchat reply, no recommendation	Returned chitchat reply	Pass
TC05	"Best food in Penang?"	Chitchat reply (non-KL location)	Returned chitchat reply	Pass
TC06	"I want food"	Recommendation (vague fallback triggered)	Returned restaurant from user preferences	Pass

6.2.2 Category B: Cuisine Classification (Direct Cuisine Names)

This category verifies that the intent analyser correctly extracts cuisine names when they are explicitly mentioned.

Table 6.2 Test Cases for Direct Cuisine Classification

Test ID	Input Prompt	Expected Output	Actual Output	Status
TC07	"I want Japanese food"	Returns Japanese restaurant	Returned Japanese restaurant	Pass
TC08	"Recommend me Chinese"	Returns Chinese restaurant	Returned Chinese restaurant	Pass
TC09	"Any good Malay restaurant?"	Returns Malay restaurant	Returned Malay restaurant	Pass
TC10	"Indian food please"	Returns Indian restaurant	Returned Indian restaurant	Pass
TC11	"Western food"	Returns Western restaurant	Returned Western restaurant	Pass
TC12	"Cafe please"	Returns Cafe/Dessert restaurant	Returned Cafe/Dessert restaurant	Pass

6.2.3 Category C: Dish-to-Cuisine Semantic Inference

This category is the most critical test of the improved intent-analysis prompt, which uses few-shot learning to map specific dish names to their corresponding cuisines.

Table 6.3 Test Cases for Dish-to-Cuisine Semantic Inference

Test ID	Input Prompt	Expected Cuisine	Actual Output	Status
TC13	"I want chicken chop"	Western	Returned Western restaurant	Pass
TC14	"I'm craving nasi lemak"	Malay	Returned Malay restaurant	Pass
TC15	"Where can I get ramen?"	Japanese	Returned Japanese restaurant	Pass
TC16	"Dim sum?"	Chinese	Returned Chinese restaurant	Pass
TC17	"Roti canai recommend"	Indian	Returned Indian restaurant	Pass
TC18	"I want waffles and coffee"	Cafe/Dessert	Returned Cafe/Dessert restaurant	Pass
TC19	"Pizza pls"	Western	Returned Western restaurant	Pass
TC20	"Biryani tonight"	Indian	Returned Indian restaurant	Pass

6.2.4 Category D: Halal Filter Detection

This category verifies that the intent analyser detects the halal requirement and that the recommendation engine correctly filters only halal-certified restaurants.

Table 6.4 Test Cases for Halal Filter Detection

Test ID	Input Prompt	Expected Behaviour	Actual Output	Status
TC21	"Any halal food?"	Halal filter active, cuisine from preferences	Returned halal restaurant matching preference	Pass
TC22	"Halal Japanese please"	Halal filter active, cuisine = Japanese	Returned halal Japanese restaurant	Pass
TC23	"Halal nasi lemak"	Halal filter active, cuisine = Malay	Returned halal Malay restaurant	Pass
TC24	"I want Japanese food"	Halal filter NOT active	Returned Japanese restaurant, halal filter off	Pass
TC25	"Muslim-friendly restaurant"	Halal filter active	Returned halal restaurant only	Pass

6.2.5 Category E: Vague-Query Fallback and Personalisation

This category tests the vague-query fallback logic (selecting from user preferences when no cuisine is specified) and the favourite-based personalisation boost.

Table 6.5 Test Cases for Vague-Query Fallback and Personalisation

Test ID	Pre-condition	Input Prompt	Expected Behaviour	Actual Output	Status
TC26	Prefs: [Western, Japanese], 0 favourites	"I'm hungry"	Returns Western or Japanese	Returned restaurant matching saved preference	Pass
TC27	Prefs: [Western, Japanese], 0 favourites	"recommend me something"	Returns Western or Japanese	Returned restaurant matching saved preference	Pass
TC28	Prefs: [Western, Japanese], 0 favourites	"Japanese food"	Normal Japanese recommendation, no boost	Returned Japanese restaurant without favourite boost	Pass
TC29	Prefs: [Japanese], 3 Japanese favourites saved	"Japanese food"	Favourite-similarity boost triggered	Returned Japanese restaurant with high fav_sim score, bot mentioned "based on your saved favourites"	Pass
TC30	New user, 0 prefs, 0 favourites	"I'm hungry"	Random cuisine from default pool	Returned random cuisine restaurant, bot said "Not sure what you're craving?"	Pass

6.2.6 Category F: Session Memory and Non-Repetition

This category tests the session-memory mechanism that prevents the same restaurant from being recommended twice within one session.

Table 6.6 Test Cases for Session Memory

Test ID	Input Sequence	Expected Behaviour	Actual Output	Status
TC31	"Japanese food" typed 3 times consecutively	3 different Japanese restaurants returned	Returned 3 different Japanese restaurants	Pass
TC32	"Japanese food" typed until all candidates exhausted	Bot replies "I've run out of new spots for that cuisine"	Returned exhaustion message correctly	Pass

6.2.7 Summary of Functional Testing

Table 6.7 Summary of Functional Testing Results

Category	Description	Total Cases	Passed	Failed	Pass Rate
A	Intent Classification	6	6	0	100%
B	Direct Cuisine Classification	6	6	0	100%
C	Dish-to-Cuisine Semantic Inference	8	8	0	100%
D	Halal Filter Detection	5	5	0	100%
E	Vague-Query Fallback and Personalisation	5	5	0	100%
F	Session Memory and Non-Repetition	2	2	0	100%
Total		32	32	0	100%

As shown in Table 6.7, all 32 functional test cases passed successfully, achieving a perfect pass rate of 100%. This indicates that every intelligent behaviour of the MakanBot chatbot, including intent detection, cuisine inference (both direct and semantic), halal filtering, cold-start handling, favourite-based personalisation, and session memory, operates correctly as designed in Chapters 3 and 4. The consistent pass results also validate the effectiveness of the

few-shot prompt-engineering approach used in the Gemini intent analyser and the adaptive weighting scheme used in the hybrid recommendation engine.

6.3 Pilot User Study Methodology

A pilot user study was conducted with a total of 10 respondents to gather preliminary feedback on the MakanBot system's usability, recommendation quality, and overall user perception(Appendix C). Due to the time constraints of the FYP development cycle, the study was designed as a small-scale qualitative-quantitative evaluation rather than a full-scale user acceptance test. The findings, while not statistically generalisable to a wider population, provide valuable indicative insights into user perception and identify areas for improvement.
The study was conducted using a Google Form survey structured in four sections:

1. System Usability and Navigation
2. Recommendation Quality and Trust
3. Overall Perception
4. Qualitative Feedback

A 7-point Likert scale was adopted for Sections 1 and 2, where a score of 1 represents "Strongly Agree" and 7 represents "Strongly Disagree". In this scale, a lower average score indicates stronger positive agreement with the statement. Section 3 used a 5-point semantic adjective scale (Exceptional, Good, Average, Poor, Very Bad) to capture users' overall perception of visual design and satisfaction. Section 4 gathered qualitative feedback through a multi-select feature-preference question and an open-ended improvement suggestion.

Respondents were recruited through convenience sampling from the researcher's personal network (classmates, family, and friends) and were invited to complete the survey after spending at least 5 to 10 minutes interacting with the MakanBot application. All responses were collected anonymously, and no personally identifying information was retained in the analysis.

6.4 Demographic Overview

All 10 respondents successfully completed the survey, producing a complete response rate of 100% (10/10 responses). The respondents were drawn primarily from a young, technology-savvy demographic, which aligns with the target user base for a mobile chatbot application. Detailed age-group and dining-frequency demographics were not captured in the pilot form in order to keep the questionnaire short; future iterations of the evaluation may expand the demographic section to allow subgroup analysis.

6.5 Quantitative Evaluation Results

This section presents the quantitative results from Sections 1, 2, and 3 of the Google Form. Figure 6.1 provides a consolidated visualisation of the average Likert scores for the seven rating-scale questions covering usability and recommendation quality. Figure 6.3 presents the overall perception of visual design and user satisfaction.

6.5.1 System Usability and Navigation

The first section evaluated the usability, interface quality, and intuitiveness of the MakanBot mobile application. A 7-point Likert scale was used for all questions, where 1 = Strongly Agree and 7 = Strongly Disagree, meaning a lower average score indicates stronger positive agreement. Table 6.8 summarises the average scores for each question.

Table 6.8 Usability and Navigation Scores

Question	Statement	Average Score	Interpretation
Q1	The app was simple and easy to navigate	2.30	Agree
Q2	The chat interface felt clean, professional, and easy to use	2.30	Agree
Q3	I felt comfortable using MakanBot without needing a manual or instructions	2.60	Slightly Agree

The average scores for Q1 and Q2 were both 2.30, which falls within the "Agree" range and indicates that the majority of respondents found the navigation and visual design of the application simple and professional. The Q3 score of 2.60 is slightly less positive but still within a favourable range, suggesting that while most users could navigate the app without

external guidance, a small number of respondents would benefit from an in-app tutorial or onboarding walkthrough explaining the features.

6.5.2 Recommendation Quality and Trust

The second section focused on evaluating the quality of restaurant recommendations and the level of trust users placed in the system. The same 7-point Likert scale was applied, where 1 = Strongly Agree and 7 = Strongly Disagree. Table 6.9 summarises the results.

Table 6.9 Recommendation Quality and Trust Scores

Question	Statement	Average Score	Interpretation
Q4	The restaurants recommended by MakanBot were relevant to my search criteria	2.30	Agree
Q5	The system provided enough information (e.g., location, menu, rating, halal status) to help me make a decision	2.20	Agree
Q6	I would follow the recommendations provided by MakanBot in a real-life situation	2.40	Agree
Q7	The "Why MakanBot recommends this" explanation was helpful in building my trust in the recommendation	2.30	Agree

All four questions in this section received average scores between 2.20 and 2.40, firmly within the "Agree" range. The strongest score was Q5 (2.20), indicating that respondents found the information provided in each restaurant card (photo, address, rating, halal status, menu, and reservation links) to be sufficient for real-world decision making. This result validates the design decision to enrich every recommendation with live Google Places metadata as described in Chapter 3.

Q7 (XAI explanation) also scored well at 2.30, supporting the project's claim that the Gemini-powered explanation layer contributes meaningfully to user trust. This confirms one of the main contributions of MakanBot: adding explainability on top of a hybrid recommendation engine improves user confidence in the output.

6.5.3 Consolidated Likert Results

Figure 6.1 visualises the average Likert scores for all seven rating-scale questions in Sections 1 and 2. The chart adopts an inverted y-axis convention so that taller bars indicate stronger agreement.

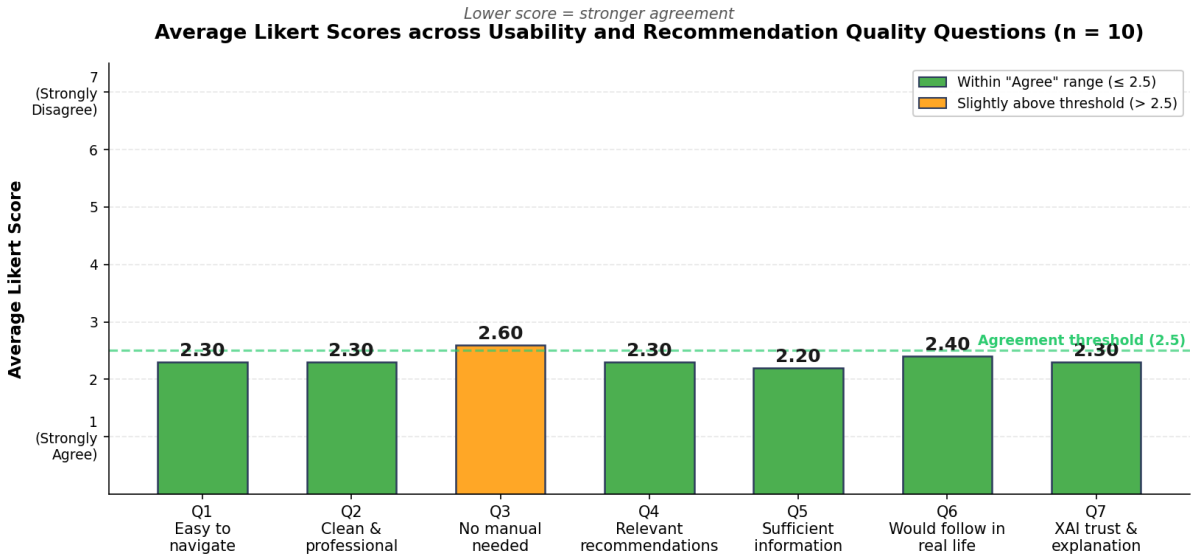


Figure 6.1 Likert Scale Averages

As shown in Figure 6.1, all seven scores fall on the "Agree" side of the scale (below the 2.5 agreement threshold for six of the seven questions), producing an overall system usability and recommendation-quality mean of 2.34. This consistently positive pattern indicates that respondents found the system to be usable, accurate, and trustworthy across multiple dimensions.

6.5.4 Overall Perception

The third section of the survey measured users' overall impression of the system using a semantic-adjective scale. Two questions were asked: one on visual design and one on overall satisfaction. Figure 6.2 illustrates the distribution of responses.

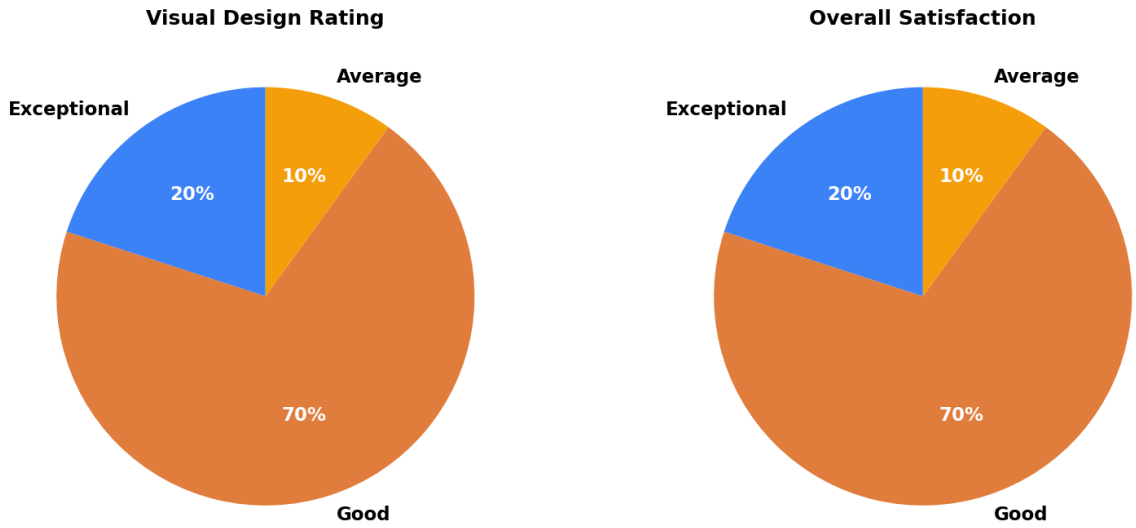


Figure 6.2 Distribution of Visual Design Ratings and Overall Satisfaction

As shown in Figure 6.2, the distributions for visual design and overall satisfaction were identical: 70% of respondents rated MakanBot as "Good", 20% rated it as "Exceptional", and 10% rated it as "Average". No respondent gave a negative rating ("Poor" or "Very Bad") in either question. This indicates that the combined favourable ratings ("Good" plus "Exceptional") account for 90% of responses in both dimensions, confirming a strongly positive overall impression of the system.

6.6 Qualitative Feedback Analysis

The fourth section of the survey collected qualitative feedback through one multi-select question on feature usefulness (Q10) and one open-ended question on improvement suggestions (Q11).

6.6.1 Most Useful Features (Q10)

Respondents were asked to select one or more features of MakanBot that they found most useful. Figure 6.3 displays the results.

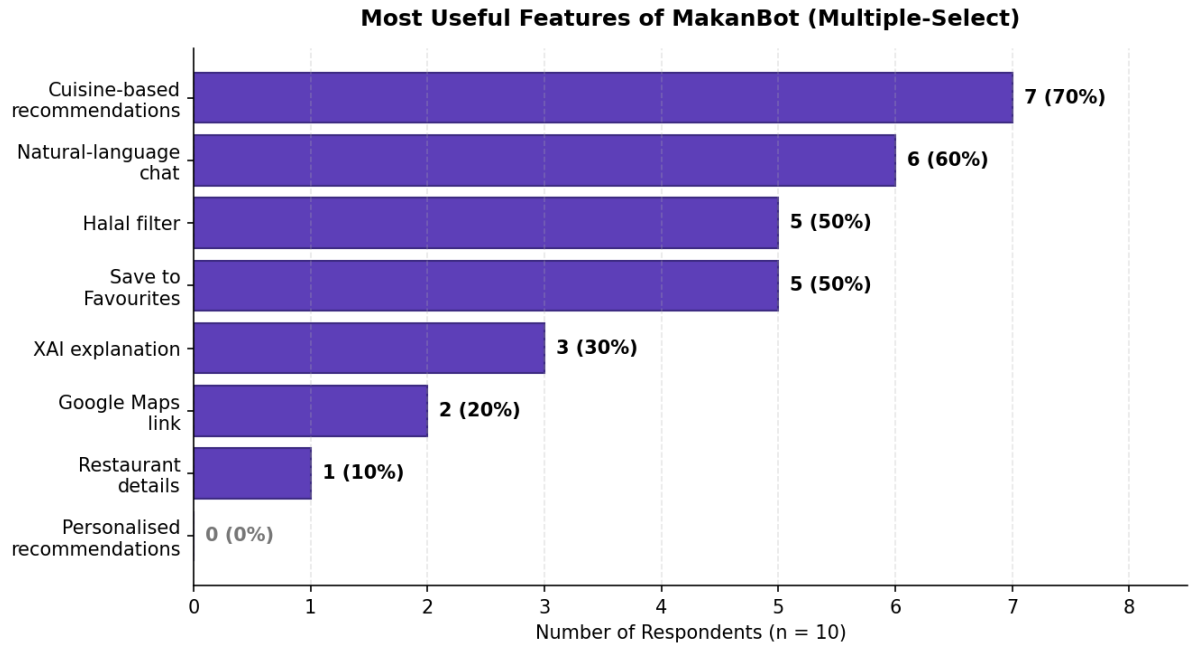


Figure 6.3 Distribution of Features Rated as Most Useful

The top three features identified by respondents were:

1. Cuisine-based recommendations (70%)
2. Natural-language chat (60%)
3. Halal filter and Save to Favourites (50%)

The strong preference for cuisine-based recommendations confirms that the core functionality of the system, which is mapping user queries to appropriate cuisine categories and retrieving matching restaurants, delivers real user value. The natural-language chat interface was rated the second most useful feature, validating the design decision to use Google Gemini as the intent-analysis layer instead of structured form-based input.

The halal filter (50%) and the save-to-favourites feature (50%) were also highly appreciated, demonstrating the relevance of these two differentiating features in the Malaysian cultural context. The relatively lower ratings for the XAI explanation (30%) and Google Maps integration (20%) suggest that while these features are valuable, they are perceived by users as

secondary benefits rather than primary reasons to use the app. Interestingly, no respondent selected "Personalised recommendations based on saved favourites" as a most useful feature (0%), which may be due to the limited amount of time respondents spent with the app during testing, leaving them unable to build up a favourites history long enough to experience the personalisation effect.

6.6.2 Suggestions for Improvement (Q11)

The open-ended improvement question received two substantive responses:

- *"Improve user interface"*
- *"I think can improved by adding more easy menu to the user"*

Both comments point in a similar direction: further refinement of the user interface and the addition of an easier, more discoverable menu. This feedback is consistent with the slightly lower Q3 score (2.60) on whether the app is comfortable to use without instructions, suggesting that a simplified navigation structure or on-boarding tutorial would be a valuable next step. The low number of open-ended responses (2 out of 10) may indicate that most respondents had no strong complaints, which is further supported by the high overall satisfaction scores discussed in Section 6.5.4.

6.7 Objective Evaluation

This section revisits the three project objectives stated in Chapter 1 and evaluates whether each has been successfully achieved based on the testing results, machine learning evaluation metrics, and user study findings discussed earlier in this chapter.

1. To prepare user generated restaurant review data

The first objective focused on collecting, cleaning, formatting, and organising textual restaurant review data so that it could be used for sentiment classification and preference extraction. As described in Chapter 3, a publicly available dataset containing 4,439 Kuala Lumpur restaurants and approximately 78,000 user reviews was acquired from Kaggle and underwent five systematic preprocessing steps, namely deduplication, handling of missing values, text normalisation, cuisine inference through keyword grouping, and halal label synchronisation. The cleaned outputs were saved as Master_Restaurants_Profile_Updated.csv and Master_Reviews_Cleaned.csv and used directly as the training data in Chapter 4. The

strong performance of the trained models, with the ensemble classifier reaching an accuracy of 90.63% and an F1 score of 94.32%, confirms that the prepared dataset was both clean and informative enough to support reliable sentiment classification. This objective is therefore considered successfully achieved.

2. To develop suitable machine learning models for sentiment classification and recommendation logic

The second objective focused on developing machine learning models capable of interpreting user reviews and mapping them to relevant restaurant attributes and user preferences. Three complementary models were developed and integrated into a hybrid recommendation engine, namely a TF IDF vectoriser, a content based ensemble classifier combining Support Vector Machine and Random Forest through soft voting, and an item based K Nearest Neighbours collaborative filter implemented using the Surprise library. As reported in Section 4.4(b), the ensemble classifier achieved an accuracy of 90.63%, a precision of 93.00%, a recall of 95.68%, an F1 score of 94.32%, and an ROC AUC of 0.94, outperforming both the standalone SVM and Random Forest models. The KNN collaborative filter recorded an RMSE of 1.0822 and an MAE of 0.8210 on the held out test set, with the best configuration using k = 40 and pearson_baseline similarity. The three signals were further fused using the adaptive weighting scheme described in Chapter 4, allowing the system to balance content relevance, collaborative patterns, and personalised taste similarity. This objective is therefore considered successfully achieved.

3. To create a mobile chatbot interface that integrates with the backend recommendation system

The third objective focused on creating a mobile chatbot interface that integrates seamlessly with the backend recommendation system and allows users to interact conversationally. As documented in Chapter 5, a native Android application was developed in Java with seven core activities covering authentication, preference management, chat, restaurant detail viewing, favourites, and profile management. The mobile frontend communicates with the FastAPI backend through HTTPS, while Firebase Authentication and Cloud Firestore handle user accounts and persistent storage. Natural language understanding is provided by the Google Gemini API, and live restaurant metadata is enriched through the Google Places API. The functional testing reported in Section 6.2 confirmed that all 32 chatbot test cases passed,

achieving a 100% pass rate across intent classification, cuisine inference, halal filtering, vague query fallback, favourite based personalisation, and session memory. The pilot user study reported in Sections 6.5 and 6.6 further confirmed user acceptance, with an average Likert agreement score of 2.34 out of 7 and 90% favourable ratings on overall satisfaction. This objective is therefore considered successfully achieved.

6.8 Concluding Remarks

This chapter presented a two-pronged evaluation of the MakanBot system combining systematic developer-driven functional testing with a pilot user study. In Section 6.2, all 32 functional test cases covering intent classification, direct and semantic cuisine inference, halal filtering, vague-query fallback, favourite-based personalisation, and session memory passed successfully, producing a 100% pass rate that confirms the correctness of the hybrid recommendation logic designed in Chapters 3 and 4. In Section 6.3 to Section 6.6, a pilot user study with 10 respondents produced consistently positive results: an average Likert score of 2.34 across seven usability and recommendation-quality questions, 90% favourable ratings (Good or Exceptional) for both visual design and overall satisfaction, and a clear ranking of the most valued features led by cuisine-based recommendations, natural-language chat, the halal filter, and the save-to-favourites function. Qualitative feedback pointed to user interface refinement and a simpler in-app menu as potential improvement areas. Taken together, the functional tests and user feedback confirm that the MakanBot system operates reliably, is perceived positively by real users, and fulfils the project objectives stated in Chapter 1. Areas identified for future enhancement, including a larger-scale user acceptance test with broader demographic coverage and an improved onboarding flow, are discussed further in the next chapter.

Chapter 7

Conclusion and Recommendation

7.1 Conclusion

In this project, a mobile chatbot for restaurant recommendation called MakanBot was developed to assist users in discovering suitable restaurants within Kuala Lumpur, Malaysia. The project began with data collection and preprocessing, where a dataset of 4,439 restaurants and approximately 78,000 user reviews was cleaned, normalised, and structured for machine learning. Three machine learning components, namely a TF-IDF vectoriser, a content-based ensemble classifier combining Support Vector Machine (SVM) and Random Forest through soft voting, and an item-based K-Nearest Neighbours collaborative filter, were trained, tested, and integrated into a hybrid recommendation engine. The ensemble classifier achieved the best performance with an accuracy of 90.63%, a recall of 95.68%, and an F1-score of 94.32%, outperforming the individual models.

The trained models were embedded into a backend service built with the FastAPI framework and connected to a native Android application developed in Java. The system supports natural-language chatting powered by the Google Gemini API, cuisine-based filtering, halal filtering, favourite-based personalisation, and explainable AI through a "Why MakanBot recommends this" message displayed on every recommendation card. Integration with the Google Places API was introduced to enrich each recommendation with live metadata such as photos, addresses, ratings, menus, and reservation links, while Firebase Authentication and Cloud Firestore were used to handle user accounts and persistent storage of preferences and saved favourites. Despite challenges such as memory overflow during collaborative filter training, class imbalance in the review dataset, and the literal-versus-semantic limitation of the initial intent-analysis prompt, the project was successful in fulfilling all of its initial objectives. The MakanBot system was successfully developed to deliver a user-friendly, intelligent, and personalised dining recommendation platform. As part of recognising the academic contribution of this work, the project has been submitted for participation in the 7th International Conference on Artificial Intelligence and Data Sciences (AiDAS 2026), a hybrid international conference scheduled for 2 – 3 September 2026 in Malaysia, with evidence of participation provided in Appendix D.

7.2 Future Work

Several enhancements in future work would further improve the MakanBot system. First, expanding the restaurant dataset to cover other major Malaysian cities beyond Kuala Lumpur, such as Penang, Johor Bahru, and Kota Kinabalu, would provide a broader set of recommendations and enhance the system's usability for users across the country. Including multilingual support for Bahasa Malaysia, Mandarin, and Tamil through natural language processing techniques would further increase the inclusivity of the system.

In addition, the current user interface could be refined based on feedback collected from the pilot user study, particularly by introducing a simpler in-app menu and an onboarding tutorial to help new users become comfortable with the features more quickly. Adding context-aware features such as a geolocation-based proximity filter, a price-range filter, and voice input would also enhance the practical usefulness of the system for everyday dining decisions. Finally, running the system on a scalable cloud platform and strengthening backend security elements would allow MakanBot to be launched to the public with good performance, stability, and protection of user data.

REFERENCES

[1] R. Khan, “Artificial Intelligence and Machine Learning in Food Industries: A Study,” *Journal of Food Chemistry and Nanotechnology*, vol. 07, no. 03, 2021, doi: https://doi.org/10.17756/jfcn.2021-114.

[2] J. Liu, Y. Yu, F. Mehraliyev, S. Hu, and J. Chen, “What affects the online ratings of restaurant consumers: a research perspective on text-mining big data analysis,” *International Journal of Contemporary Hospitality Management*, May 2022, doi: https://doi.org/10.1108/ijchm-06-2021-0749.

[3] M. S. Hossain, M. F. Rahman, M. K. Uddin, and M. K. Hossain, “Customer sentiment analysis and prediction of halal restaurants using machine learning approaches,” *Journal of Islamic Marketing*, Jun. 2022, doi: https://doi.org/10.1108/jima-04-2021-0125.

[4] D. R. Patil, D. Shukla, A. Kumar, Y. Rajanak and Y. Pratap Singh, "Machine Learning for Sentiment Analysis and Classification of Restaurant Reviews," *2022 3rd International Conference on Computing, Analytics and Networks (ICAN)*, Rajpura, Punjab, India, 2022, pp. 1-5, doi: 10.1109/ICAN56228.2022.10007390.

[5] M. Sanan. Nawaz, A. Hassan, Abu Huraira, I. Hussain, S. Qadri, and J. Jabeen, “NEXT-GENERATION FOOD RECOMMENDATION SYSTEM: A REAL-TIME, FEEDBACK-DRIVEN CHATBOT SOLUTION FOR RESTAURANTS,” *Kashf Journal of Multidisciplinary Research*, vol. 2, no. 06, pp. 16–29, 2025, doi: https://doi.org/10.71146/kjmr480.

[6] H. Kim, S. Jung, and G. Ryu, “A Study on the Restaurant Recommendation Service App Based on AI Chatbot Using Personalization Information,” *International Journal of Advanced Culture Technology*, vol. 8, no. 4, pp. 263–270, 2020, doi: https://doi.org/10.17703/IJACT.2020.8.4.263.

[7] M. Romero-Charneco, A.-M. Casado-Molina, P. Alarcón-Urbistondo, and J. P. Cabrera Sánchez, “Customer intentions toward the adoption of WhatsApp chatbots for restaurant recommendations,” *Journal of Hospitality and Tourism Technology*, Jan. 2025, doi: https://doi.org/10.1108/jhtt-01-2024-0024.

[8] H. Li, B. X. B. Yu, G. Li, and H. Gao, “Restaurant survival prediction using customer-generated content: An aspect-based sentiment analysis of online reviews,” *Tourism Management*, vol. 96, p. 104707, Jun. 2023, doi: https://doi.org/10.1016/j.tourman.2022.104707.

[9] R.S. Mohana, K. Kousalya, B. Krishnakumar, Lohappriya D, A. Khan, and M. Nafeez, “Restaurant based recommender system based on sentimental analysis,” *2022 International Conference on Computer Communication and Informatics (ICCCI)*, pp. 1–6, Jan. 2022, doi: https://doi.org/10.1109/iccci54379.2022.9740920

[10] "Restaurant Recommendation System using Machine Learning," *International Journal of Advanced Trends in Computer Science and Engineering*, vol. 10, no. 3, pp. 1671–1675, Jun. 2021, doi: https://doi.org/10.30534/ijatcse/2021/261032021

[11] S. Goyal, N. T. Singh, Iesh Bajetha, S. Singh, H. Singh, and P. Kumar, "Restaurant Recommendation System using Machine Learning Algorithms," vol. 10, pp. 1–5, Apr. 2024, doi: https://doi.org/10.1109/ickecs61492.2024.10616430.

[12] N. Shirisha, T. Bhaskar, A. Kiran, and K. Alankruthi, "Restaurant Recommender System Based On Sentiment Analysis," *2022 International Conference on Computer Communication and Informatics (ICCCI)*, Jan. 2023, doi: https://doi.org/10.1109/iccci56745.2023.10128234.

[13] M. Chemlal, A. Zedadra, M. Nadjib Kouahla, and O. Zedadra, "A Restaurant Recommendation System based on Collaborative Filtering using Machine Learning Algorithms and the Multi-Criteria Method. ⋆." Accessed: Apr. 12, 2025. [Online]. Available: https://ceur-ws.org/Vol-3935/Paper13.pdf

[14] E. Asani, H. Vahdat-Nejad, and J. Sadri, "Restaurant recommender system based on sentiment analysis," *Machine Learning with Applications*, p. 100114, Jul. 2021, doi: https://doi.org/10.1016/j.gruwa.2021.100114.

[15] R. J. Kuo, C.-K. Chen, and S.-H. Keng, "Application of hybrid metaheuristic with perturbation-based K-nearest neighbors algorithm and densest imputation to collaborative filtering in recommender systems," *Information Sciences*, vol. 575, pp. 90–115, Oct. 2021, doi: https://doi.org/10.1016/j.ins.2021.06.026

[16] K.S. Kalaivani, C.S. Kanimozhiselvi, B.R. Narmatha, and Z. H. Mohamed. Bilal, "Improving the Restaurant Recommendation Performance using Sentiment Analysis," *2022 4th International Conference on Inventive Research in Computing Applications (ICIRCA)*, pp. 296–300, Aug. 2023, doi: https://doi.org/10.1109/icirca57980.2023.10220869.

[17] F. Fkih, "Similarity measures for Collaborative Filtering-based Recommender Systems: Review and experimental comparison," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 9, Sep. 2021, doi: https://doi.org/10.1016/j.jksuci.2021.09.014.

[18] R. De Croon, L. Van Houdt, N. N. Htun, G. Štiglic, V. Vanden Abeele, and K. Verbert, "Health Recommender Systems: Systematic Review," *Journal of Medical Internet Research*, vol. 23, no. 6, p. e18035, Jun. 2021, doi: https://doi.org/10.2196/18035.

[19] M. M. Afsar, T. Crump, and B. Far, "Reinforcement Learning based Recommender Systems: A Survey," *ACM Computing Surveys*, Jun. 2022, doi: https://doi.org/10.1145/3543846.

[20] Y. Gao, T. Sheng, Y. Xiang, Y. Xiong, H. Wang, and J. Zhang, “Chat-REC: Towards Interactive and Explainable LLMs-Augmented Recommender System,” Mar. 2023, doi: https://doi.org/10.48550/arxiv.2303.14524.

[21] C. Gao, X. Wang, X. He, and Y. Li, “Graph Neural Networks for Recommender System,” *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*, Feb. 2022, doi: https://doi.org/10.1145/3488560.3501396.

[22] T. N. T. Tran, A. Felfernig, C. Trattner, and A. Holzinger, “Recommender systems in the healthcare domain: state-of-the-art and research issues,” *Journal of Intelligent Information Systems*, vol. 57, pp. 171–201, Dec. 2020, doi: https://doi.org/10.1007/s10844-020-00633-6.

[23] Y. Hou *et al.*, “Large Language Models are Zero-Shot Rankers for Recommender Systems,” *Lecture notes in computer science*, pp. 364–381, Jan. 2024, doi: https://doi.org/10.1007/978-3-031-56060-6_24.

[24] J. Yu, H. Yin, X. Xia, T. Chen, J. Li, and Z. Huang, “Self-Supervised Learning for Recommender Systems: A Survey,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 1, pp. 335–355, Jun. 2023, doi: https://doi.org/10.1109/tkde.2023.3282907.

[25] S. Wang *et al.*, “Graph Learning based Recommender Systems: A Review,” *arXiv (Cornell University)*, Aug. 2021, doi: https://doi.org/10.24963/ijcai.2021/630.

[26] Andrés Solano-Barliza *et al.*, “Recommender systems applied to the tourism industry: a literature review,” *Cogent Business & Management*, vol. 11, no. 1, Jun. 2024, doi: https://doi.org/10.1080/23311975.2024.2367088.

[27] S. Forouzandeh, K. Berahmand, E. Nasiri, and M. Rostami, “A Hotel Recommender System for Tourists Using the Artificial Bee Colony Algorithm and Fuzzy TOPSIS Model: A Case Study of TripAdvisor,” *International Journal of Information Technology & Decision Making*, vol. 20, no. 01, pp. 399–429, Jan. 2021, doi: https://doi.org/10.1142/s0219622020500522.

[28] A. Melese, "Food and Restaurant Recommendation System Using Hybrid Filtering Mechanism," *North American Academic Research*, vol. 4, no. 4, pp. 268–281, 2021, doi: 10.5281/zenodo.4712849.

[29] Vidula N.A, T. Hemanth. Babu, V. G. Kiran, R. M. Balakrishnan, and S. Bhaskaran, “Prompt Palate: A Prompt-Based Restaurant Recommendation System with Scheduling and Chatbot Integration,” pp. 1–6, Nov. 2024, doi: https://doi.org/10.1109/csitss64042.2024.10816735.

[30] C. Gao, W. Lei, X. He, M. de Rijke, and T.-S. Chua, “Advances and challenges in conversational recommender systems: A survey,” *AI Open*, vol. 2, pp. 100–126, 2021, doi: https://doi.org/10.1016/j.aiopen.2021.06.002.

[31]` Q. Zhang, J. Lu, and Y. Jin, "Artificial Intelligence in Recommender Systems," *Complex & Intelligent Systems*, vol. 7, no. 1, pp. 439–457, Nov. 2020, doi: https://doi.org/10.1007/s40747-020-00212-w.

[32] D. Roy and M. Dutta, "A systematic review and research perspective on recommender systems," *Journal of Big Data*, vol. 9, no. 1, May 2022, doi: https://doi.org/10.1186/s40537-022-00592-5.

[33] H. Ko, S. Lee, Y. Park, and A. Choi, "A Survey of Recommendation Systems: Recommendation Models, Techniques, and Application Fields," *Electronics*, vol. 11, no. 1, p. 141, Jan. 2022, doi: https://doi.org/10.3390/electronics11010141

[34] Y. Luo and X. Xu, "Comparative study of deep learning models for analysing online restaurant reviews in the era of the COVID-19 pandemic," *International Journal of Hospitality Management*, vol. 94, p. 102849, Apr. 2021, doi: https://doi.org/10.1016/j.ijhm.2020.102849.

[35] A. K. Dixit, R. R. Nair and T. Babu, "Analysis and Classification of Restaurants Based on Rating with XGBoost Model," 2022 3rd International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT), Ghaziabad, India, 2022, pp. 1-6, doi: 10.1109/ICICT55121.2022.10064491

[36] M. Al-Hubaishi, Ali, and N. M. Tahir, "Personalized Food Recommendations: A Machine Learning Model for Enhanced Dining Choices," pp. 316–320, Aug. 2024, doi: https://doi.org/10.1109/iccsce61582.2024.10696536.

[37] M. Lee, W. Kwon, and K.-J. Back, "Artificial intelligence for hospitality big data analytics: developing a prediction model of restaurant review helpfulness for customer decision-making," International Journal of Contemporary Hospitality Management, vol. 33, no. 6, pp. 2117–2136, Jun. 2021, doi: https://doi.org/10.1108/ijchm-06-2020-0587.

[38] Rutuja Deepak Abhang, Bhakti Deepak Bailurkar, Sakshi Shailesh Save, Pradnya Dilip Ingale, and Manali Yashwant Patekar, "Zomato Review Analysis Using Machine Learning," May 2023, doi: https://doi.org/10.1109/iconscept57958.2023.10170538.

[39] V. Umarani, A. Julian, and J. Deepa, "Sentiment Analysis using various Machine Learning and Deep Learning Techniques," Journal of the Nigerian Society of Physical Sciences, pp. 385–394, Nov. 2021, doi: https://doi.org/10.46481/jnsps.2021.308.

[40] Harini Burra and P. Mishra, "Restaurant Reviews Sentimental Analysis Using Machine Learning Approach," pp. 414–417, Aug. 2024, doi: https://doi.org/10.1109/icetci62771.2024.10704184.

[41] R. Obiedat *et al.*, "Sentiment Analysis of Customers' Reviews Using a Hybrid Evolutionary SVM-Based Approach in an Imbalanced Data Distribution," *IEEE Access*, vol. 10, pp. 22260–22273, 2022, doi: https://doi.org/10.1109/access.2022.3149482.

[42] G. A. Fadhlullah, Z. K. A. Baizal, and N. Ikhsan, "Conversational Recommender Systems Based on Mobile Chatbot for Culinary," *JURNAL MEDIA INFORMATIKA BUDIDARMA*, vol. 5, no. 4, p. 1233, Oct. 2021, doi: https://doi.org/10.30865/mib.v5i4.3242.

[43] V. Ravi, J. Staddon, J. Chase, and A. Research, "Hungry Enough for a Chatbot: Automation Opportunities for a Restaurant Recommender." Accessed: Jul. 23, 2025. [Online]. Available: https://conversations2022.wordpress.com/wp-content/uploads/2022/11/conversations_2022_projectpresentation_6_ravi.pdf

[44] K. N. Lajis, "Navigating Culinary Choices: An Exploration of AI Chatbots for Personalized Restaurant Recommendations," *Pakistan Journal of Life and Social Sciences (PJLSS)*, vol. 22, no. 2, 2024, doi: https://doi.org/10.57239/pjlss-2024-22.2.001075.

[45] R. Garg *et al.*, "NLP Based Chatbot for Multiple Restaurants," *2021 10th International Conference on System Modeling & Advancement in Research Trends (SMART)*, Dec. 2021, doi: https://doi.org/10.1109/smart52563.2021.9676218.

[46] J. Yoon and H. Yu, "Impact of customer experience on attitude and utilization intention of a restaurant-menu curation chatbot service," *Journal of Hospitality and Tourism Technology*, vol. ahead-of-print, no. ahead-of-print, Mar. 2022, doi: https://doi.org/10.1108/jhtt-03-2021-0089.

[47] C. K. Ng, *Malaysia Restaurant Review Datasets*, Research Project, Master of Data Science, University of Malaya, 2022. [Online]. Available: kaggle.com/datasets/choonkhonng/malaysia-restaurant-review-datasets

APPENDIX

Appendix A: Python Backend Source Code

main.py

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import firebase_admin
from firebase_admin import credentials, firestore
from google import genai
from google.genai import types
import json
import random
import traceback
from recommender import RestaurantEngine

# --- INITIALIZE FIREBASE ---
try:
    if not firebase_admin._apps:
        cred = credentials.Certificate("serviceAccountKey.json")
        firebase_admin.initialize_app(cred)
    db = firestore.client()
    print(" Firebase Connected")
except Exception as e:
    print(f" Firebase Init Error: {e}")

import logging
logging.basicConfig(level=logging.INFO)

app = FastAPI(title="MakanBot Hybrid API")
engine = RestaurantEngine()
recommendation_history = {}

MODEL_NAME = "gemini-3.1-flash-lite-preview"

class ChatRequest(BaseModel):
    user_id: str
    message: str

# --- INTENT ANALYSIS ---
async def analyze_intent_with_llm(user_message: str, user_client: genai.Client):
    ALLOWED_CUISINES = "Chinese, Malay, Indian, Japanese, Western, Cafe/Dessert,
General"
    prompt = f"""
    Analyze this user message about food: "{user_message}"

    RULES:
    1. If the user asks for things NOT related to food/restaurants (e.g., hotels, parks, weather),
```

```
set intent_type to "chitchat".
   2. If the user mentions a city OTHER than Kuala Lumpur, set intent_type to "chitchat".
   3. If the user asks for food/restaurant recommendation BUT does NOT specify any cuisine
or dish
      (e.g., "I'm hungry", "recommend me something", "any good food"), set cuisine to
"ANY".
   4. IMPORTANT  Dish-to-Cuisine Mapping. If the user mentions a SPECIFIC DISH,
      infer the cuisine from these examples:

      Chinese   -> dim sum, char siew, wonton noodles, roast duck, steamboat,
                   hokkien mee, bak kut teh, kung pao chicken, fried rice,
                   kuey teow, porridge, dumplings
      Malay     -> nasi lemak, rendang, satay, ayam goreng, nasi kerabu,
                   ikan bakar, sambal, nasi padang, mee rebus
      Indian    -> roti canai, tosai, biryani, curry, tandoori, naan,
                   banana leaf rice, masala, mamak food
      Japanese  -> sushi, ramen, sashimi, tempura, udon, teriyaki, donburi,
                   tonkatsu, unagi, izakaya
      Western   -> chicken chop, steak, burger, pasta, pizza, fish and chips,
                   carbonara, spaghetti, grilled chicken, sandwich, ribs, brunch
      Cafe/Dessert -> cake, waffle, ice cream, pancake, bingsu, crepe, coffee,
                      donut, gelato, pastry, bakery

      If the dish clearly matches one of these categories, extract that cuisine.
      If the dish doesn't fit any clear category, set cuisine to "ANY".
   5. If the user explicitly names a cuisine (e.g., "Japanese food"), use that directly.

   Extract: cuisine (from {ALLOWED_CUISINES} or "ANY"), halal_required (bool),
intent_type (chitchat/recommendation).
   Return ONLY JSON. Example: {{"cuisine": "Western", "halal_required": false,
"intent_type": "recommendation"}}
   """
   try:
      response = user_client.models.generate_content(model=MODEL_NAME,
contents=prompt)
      text = response.text.strip()
      if "```json" in text:
         text = text.split("```json")[1].split("```")[0].strip()
      elif "```" in text:
         text = text.split("```")[1].split("```")[0].strip()
      return json.loads(text)
   except Exception as e:
      print(f" LLM Intent Analysis Failed: {e}. Using defaults.")
      return {"cuisine": "ANY", "halal_required": False, "intent_type": "recommendation"}


def get_user_preferences(user_data: dict) -> list:
   """Reads saved cuisine preferences, handling 'preference' vs 'preferences' mismatch."""
   prefs = user_data.get("preference") or user_data.get("preferences")
```

```
    if prefs is None:
        return []
    if isinstance(prefs, list):
        return [str(p).strip() for p in prefs if p]
    if isinstance(prefs, str):
        return [p.strip() for p in prefs.split(",") if p.strip()]
    return []


# Fetch user's favourites from Firestore
def get_user_favorites(user_id: str) -> list:
    """
    Returns a list of restaurant names the user has saved to favourites.
    Reads from: users/{uid}/favorites/*  (one doc per restaurant, doc ID = name)
    """
    try:
        favs_ref = db.collection("users").document(user_id).collection("favorites")
        docs = favs_ref.stream()
        names = [doc.id for doc in docs]
        print(f" User has {len(names)} favourite(s): {names[:5]}{'...' if len(names) > 5 else ''}")
        return names
    except Exception as e:
        print(f" Failed to fetch favourites: {e}")
        return []


# --- MAIN CHAT ENDPOINT ---
@app.post("/chat")
async def chat_endpoint(request: ChatRequest):
    try:
        print(f" Received message from {request.user_id}: {request.message}")

        # --- 1. DATA FETCHING ---
        user_ref = db.collection("users").document(request.user_id).get()
        if not user_ref.exists:
            raise HTTPException(status_code=404, detail="User not found")

        user_data = user_ref.to_dict()
        user_key = user_data.get("gemini_api_key")
        user_prefs_list = get_user_preferences(user_data)
        user_prefs_text = ", ".join(user_prefs_list) if user_prefs_list else "general food"

        # Load favourites
        user_favorites = get_user_favorites(request.user_id)

        print(f" Prefs: {user_prefs_list} | Favs count: {len(user_favorites)}")

        if not user_key:
            return {"bot_response": "Please set your Gemini API key in settings.",
```

```
"is_recommendation": False}

    user_client = genai.Client(api_key=user_key)

    # --- 2. INTENT ANALYSIS ---
    analysis = await analyze_intent_with_llm(request.message, user_client)

    if analysis.get("intent_type") == "chitchat":
        return {"bot_response": "Hi! I'm MakanBot. Tell me what you want to eat in KL!",
"is_recommendation": False}

    # Extract halal filter ONCE, use it throughout the function
    halal_filter = analysis.get("halal_required", False)

    # --- 3. SESSION MEMORY ---
    user_history = recommendation_history.get(request.user_id, [])

    # --- 4. VAGUE-QUERY FALLBACK ---
    requested_cuisine = analysis.get("cuisine", "ANY")
    used_fallback = False
    fallback_source = None

    if requested_cuisine in ("ANY", "General", "", None):
        if user_prefs_list:
            requested_cuisine = random.choice(user_prefs_list)
            used_fallback = True
            fallback_source = "preference"
            print(f" No cuisine mentioned. Fallback to preference: {requested_cuisine}")
        else:
            requested_cuisine = random.choice(["Chinese", "Malay", "Western", "Japanese",
"Indian"])
            used_fallback = True
            fallback_source = "random"
            print(f" No prefs either. Random cuisine: {requested_cuisine}")

    # --- 5. HYBRID RECOMMENDATION ---
    results = engine.recommend(
        user_id=request.user_id,
        query_cuisine=requested_cuisine,
        halal_only=halal_filter,
        top_n=1,
        exclude_names=user_history,
        favorite_names=user_favorites
    )

    # Retry across preferences if empty
    if not results and used_fallback and fallback_source == "preference":
        remaining_prefs = [p for p in user_prefs_list if p != requested_cuisine]
        for alt in remaining_prefs:
```

```
            print(f" Retrying with: {alt}")
            results = engine.recommend(
                user_id=request.user_id,
                query_cuisine=alt,
                halal_only=halal_filter,
                top_n=1,
                exclude_names=user_history,
                favorite_names=user_favorites
            )
            if results:
                requested_cuisine = alt
                break

    # --- 6. CAPTURE SCORING INFO BEFORE XAI OVERWRITES IT ---
    was_boosted = False
    fav_match_pct = 0

    if results:
        new_rest = results[0]

        if request.user_id not in recommendation_history:
            recommendation_history[request.user_id] = []
        recommendation_history[request.user_id].append(new_rest['name'])

        scoring_data = new_rest.get('evidence', {})
        was_boosted = scoring_data.get('boosted_by_favorites', False)
        fav_match_pct = scoring_data.get('favorite_match', 0)

        xai_prompt = f"""
        User's saved cuisine preferences: {user_prefs_text}
        User has {len(user_favorites)} favourite restaurant(s) saved.
        Current request: "{request.message}"
        Halal filter active: {halal_filter}
        Recommended Restaurant: {new_rest['name']} ({requested_cuisine})
        Scores: cuisine_match={scoring_data.get('cuisine_match')}%,
favorite_similarity={fav_match_pct}%

        Write ONE friendly sentence (max 25 words) explaining why this restaurant fits this
user.
        - If Halal filter is active, mention that this place is halal-certified.
        - If favorite_similarity is above 40%, mention it feels similar to places they've saved
before.
        - If the user asked vaguely (e.g. "I'm hungry") but gave a halal filter, mention it
matches their preferences AND is halal.
        - If the user asked vaguely with no halal filter, mention it matches their saved
preferences.
        - Otherwise just explain why based on cuisine.
        """
        try:
```

```
            xai_res = user_client.models.generate_content(model=MODEL_NAME,
contents=xai_prompt)
            new_rest['evidence'] = xai_res.text.strip()
        except:
            if was_boosted and fav_match_pct >= 40:
                new_rest['evidence'] = f"{new_rest['name']} shares a similar vibe to the
restaurants you've saved before!"
            elif used_fallback and fallback_source == "preference":
                new_rest['evidence'] = f"Since you didn't specify, I picked {new_rest['name']}
from your saved love for {requested_cuisine} food!"
            else:
                new_rest['evidence'] = f"Based on your taste, {new_rest['name']} is a great
choice!"

    # --- 7. FINAL BOT MESSAGE ---
    if results:
        if was_boosted and fav_match_pct >= 40:
            if halal_filter:
                bot_msg = f"Based on restaurants you've saved before, I think you'll love this
halal {requested_cuisine} spot!"
            else:
                bot_msg = f"Based on restaurants you've saved before, I think you'll love this
{requested_cuisine} spot!"

        elif used_fallback and fallback_source == "preference":
            if halal_filter:
                bot_msg = f"Based on your preference for {requested_cuisine} food, here's a
halal spot you might like!"
            else:
                bot_msg = f"You didn't mention a cuisine, so I picked a {requested_cuisine}
spot from your favourites!"

        elif used_fallback and fallback_source == "random":
            if halal_filter:
                bot_msg = f"Here's a halal {requested_cuisine} spot you might enjoy!"
            else:
                bot_msg = f"Not sure what you're craving? Try this {requested_cuisine} spot!"

        else:
            if halal_filter:
                bot_msg = f"I found a great halal {requested_cuisine} spot for you!"
            else:
                bot_msg = f"I found a great {requested_cuisine} spot for you!"
    else:
        if halal_filter:
            bot_msg = f"I couldn't find a halal {requested_cuisine} spot right now. Try a
different cuisine?"
        else:
            bot_msg = "I've run out of new spots for that cuisine. Try a different one?"
```

```
        return {
            "bot_response": bot_msg,
            "is_recommendation": len(results) > 0,
            "recommendations": results
        }

    except Exception as e:
        print(traceback.format_exc())
        raise HTTPException(status_code=500, detail=str(e))
```

recommender.py

```
import pandas as pd
import numpy as np
import pickle
import requests
import urllib.parse
import os
import traceback
from surprise import dump
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer
from scipy.sparse import hstack, vstack
from sklearn.metrics.pairwise import cosine_similarity

class RestaurantEngine:
    def __init__(self):
        self.sia = SentimentIntensityAnalyzer()
        nltk.download('vader_lexicon')
        print("--- Loading FYP2 Dataset-Driven Engine ---")

        # 1. Load local CSV dataset
        try:
            self.df = pd.read_csv('Master_Restaurants_Profile_Updated.csv')
            self.df.columns = self.df.columns.str.strip()
            print(f" Dataset Loaded: {len(self.df)} restaurants found.")
        except Exception as e:
            print(f"CRITICAL ERROR: Could not load
Master_Restaurants_Profile_Updated.csv. {e}")
            self.df = pd.DataFrame()

        # 2. Load ML Models
        try:
            with open('models/tfidf_vectorizer.pkl', 'rb') as f:
                self.tfidf = pickle.load(f)
            with open('models/content_ensemble.pkl', 'rb') as f:
                self.ensemble = pickle.load(f)
            print(" Content Models (TF-IDF & Ensemble) Loaded.")
        except:
```

```
            print(" Model Load Failed. Check if models folder exists.")

        try:
            _, self.knn = dump.load('models/knn_model.pkl')
            print(" KNN Model Loaded.")
        except:
            self.knn = None

        self.google_api_key = "AIzaSyBVEI7ScFbYV-8ukJGs6VUM2qN8y_fERkM"

    def get_google_metadata(self, restaurant_name):
        """Fetches Google Places metadata with fallbacks for images."""
        url =
f"https://maps.googleapis.com/maps/api/place/textsearch/json?query={restaurant_name}+Ku
ala+Lumpur&key={self.google_api_key}"
        try:
            resp = requests.get(url, timeout=5).json()
            if not resp.get('results'):
                return None

            place = resp['results'][0]
            place_id = place.get('place_id')

            photo_ref = ""
            if 'photos' in place and len(place['photos']) > 0:
                photo_ref = place['photos'][0].get('photo_reference', "")

            details_url =
f"https://maps.googleapis.com/maps/api/place/details/json?place_id={place_id}&fields=phot
os,formatted_address,rating,user_ratings_total,website,reservations_uri&key={self.google_a
pi_key}"
            details_resp = requests.get(details_url, timeout=5).json().get('result', {})

            if 'photos' in details_resp and len(details_resp['photos']) > 0:
                photo_ref = details_resp['photos'][0].get('photo_reference', photo_ref)

            img = ""
            if photo_ref:
                safe_ref = urllib.parse.quote(photo_ref)
                img =
f"https://maps.googleapis.com/maps/api/place/photo?maxwidth=800&photoreference={safe_
ref}&key={self.google_api_key}"

            return {
                "address": details_resp.get('formatted_address', place.get('formatted_address',
"Kuala Lumpur")),
                "rating": details_resp.get('rating', place.get('rating', 0.0)),
                "reviews": details_resp.get('user_ratings_total', place.get('user_ratings_total', 0)),
                "image_url": img,
```

```
            "menu_link": details_resp.get('website',
f"https://www.google.com/search?q={restaurant_name.replace(' ', '+')}+menu"),
            "reservation_link": details_resp.get('reservations_uri',
f"https://www.google.com/search?q={restaurant_name.replace(' ', '+')}+reservation")
        }
    except Exception as e:
        print(f" Google API Fetch Warning: {e}")
        return None

# NEW METHOD Build the user's "taste vector" from favourites
def _build_favorite_vector(self, favorite_names):
    """
    Creates a single averaged TF-IDF vector representing everything the user loves.
    Returns None if no favourites or favourites aren't in the dataset.
    """
    if not favorite_names or self.df.empty:
        return None

    # Find all favourited restaurants in our dataset
    fav_rows = self.df[self.df['Restaurant'].isin(favorite_names)]
    if fav_rows.empty:
        print(f" None of the {len(favorite_names)} favourites match dataset. Skipping
boost.")
        return None

    # Build feature strings exactly as we do during scoring (consistency is critical)
    fav_features = [
        f"{row['Restaurant']} {row.get('Cuisine', '')}".lower()
        for _, row in fav_rows.iterrows()
    ]

    # Transform all favourites into TF-IDF vectors
    fav_vectors = self.tfidf.transform(fav_features)

    # Average them into ONE "taste profile" vector
    # Dense mean because sparse matrices don't have .mean() that returns sparse
    avg_vector = np.asarray(fav_vectors.mean(axis=0))

    print(f" Built taste vector from {len(fav_rows)} matched favourites.")
    return avg_vector

def recommend(self, user_id, query_cuisine, halal_only=False, top_n=1,
              exclude_names=None, favorite_names=None):
    """
    Hybrid recommendation with three signals:
      1. Content-based (TF-IDF + Sentiment + ensemble classifier)
      2. Collaborative (KNN)
      3. Favourite-similarity boost (cosine sim with user's taste vector)
    """
```

```
        try:
            if self.df.empty: return []

            target_exclude = exclude_names if exclude_names is not None else []
            favorite_names = favorite_names if favorite_names is not None else []

            # Pre-compute the user's taste vector ONCE (not per-candidate)
            taste_vector = self._build_favorite_vector(favorite_names)
            has_favorites = taste_vector is not None

            # 1. Filter by Cuisine
            candidates = self.df[self.df['Cuisine'].str.contains(query_cuisine, case=False,
na=False)].copy()

            if candidates.empty:
                print(f"ℹ No local matches for {query_cuisine}")
                return []

            # Don't recommend something the user has already favourited
            candidates = candidates[~candidates['Restaurant'].isin(favorite_names)]

            scored_list = []
            for _, row in candidates.iterrows():
                name = row['Restaurant']

                if name in target_exclude:
                    continue

                # --- CONTENT SCORING ---
                feat = f"{name} {row.get('Cuisine', '')} {query_cuisine}"
                input_vec_tfidf = self.tfidf.transform([feat.lower()])
                prediction_sentiment = self.sia.polarity_scores(feat)['compound']
                input_vec_final = hstack([input_vec_tfidf, np.array([[prediction_sentiment]])])
                content_prob = self.ensemble.predict_proba(input_vec_final)[0][1]

                # --- COLLABORATIVE SCORING ---
                try:
                    collab_pred = self.knn.predict(user_id, name).est / 5.0 if self.knn else 0.7
                except:
                    collab_pred = 0.6

                # ---  FAVOURITE-SIMILARITY BOOST ---
                fav_sim = 0.0
                if has_favorites:
                    # Cosine similarity between candidate TF-IDF and user's taste vector
                    candidate_vec = input_vec_tfidf.toarray()
                    sim = cosine_similarity(candidate_vec, taste_vector)[0][0]
                    fav_sim = float(max(0.0, min(1.0, sim)))  # Clamp to [0, 1]
```

```
            # --- HALAL FILTER ---
            csv_halal = str(row.get('HALAL_UPDATED', 'NO')).upper()
            if halal_only and csv_halal == "NO":
                continue

            # --- HYBRID SCORE FUSION ---
            #  Adaptive weighting: if user has favourites, make room for the boost
            if has_favorites:
                # Weights sum to 1.0:  content 0.5, collab 0.2, favourites 0.3
                final_score = (0.5 * content_prob) + (0.2 * collab_pred) + (0.3 * fav_sim)
            else:
                # Fall back to original weights when user has no favourites yet
                final_score = (0.7 * content_prob) + (0.3 * collab_pred)

            print(f"DEBUG {name}: content={content_prob:.3f}, collab={collab_pred:.3f},
fav_sim={fav_sim:.3f}, final={final_score:.3f}")

            scored_list.append({
                "name": name,
                "score": final_score,
                "content_prob": content_prob,
                "collab_pred": collab_pred,
                "fav_sim": fav_sim,                  #  expose for XAI
                "halal": csv_halal,
                "boosted_by_favorites": has_favorites  # flag for UI/XAI
            })

        # 2. Rank
        scored_list.sort(key=lambda x: x["score"], reverse=True)
        winners = scored_list[:top_n]
        final_results = []

        for win in winners:
            print(f"Fetching Google details for: {win['name']}")
            g = self.get_google_metadata(win['name'])

            safe_name = win['name'].replace(' ', '+')
            backup_menu = f"https://www.google.com/search?q={safe_name}+menu"
            backup_res = f"https://www.google.com/search?q={safe_name}+reservation"

            final_results.append({
                "name": win['name'],
                "score": round(float(win['score']), 4),
                "address": g['address'] if g else "Kuala Lumpur, Malaysia",
                "image_url": g['image_url'] if g else "",
                "rating": g['rating'] if g else 4.0,
                "halal": win['halal'],
                "menu_link": g['menu_link'] if g else backup_menu,
                "reservation_link": g['reservation_link'] if g else backup_res,
```

```
                "location_link":
f"https://www.google.com/maps/search/?api=1&query={safe_name}+Kuala+Lumpur",
                "evidence": {
                    "cuisine_match": round(win['content_prob'] * 100, 1),
                    "personal_match": round(win['collab_pred'] * 100, 1),
                    "favorite_match": round(win['fav_sim'] * 100, 1),      #
                    "boosted_by_favorites": win['boosted_by_favorites'],    #
                    "google_rating": g['rating'] if g else 4.0,
                    "reviews": g['reviews'] if g else 100
                },
            })

        return final_results

    except Exception as e:
        print(f"Engine Error: {e}")
        traceback.print_exc()
        return []
```

Appendix B: Android Java Source Code

SplashActivity.java

```
package com.example.makanbot;

import android.content.Intent;
import android.os.Bundle;
import android.os.Handler;
import android.view.animation.Animation;
import android.view.animation.AnimationUtils;
import android.widget.ImageView;
import androidx.appcompat.app.AppCompatActivity;

public class SplashActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_splash);

        ImageView logo = findViewById(R.id.splash_logo);

        // Load and start the animation
        Animation fadeIn = AnimationUtils.loadAnimation(this, R.anim.fade_in);
        logo.startAnimation(fadeIn);

        // Timer to move to next screen after 3 seconds
        new Handler().postDelayed(new Runnable() {
            @Override
            public void run() {
```

```
            // Change LoginActivity.class to whatever your next screen is named
            Intent intent = new Intent(SplashActivity.this, LoginActivity.class);
            startActivity(intent);
            finish(); // Destroys SplashActivity so user can't "Go Back" to it
        }
    }, 3000);
  }
}
```

LoginActivity.java

```
package com.example.makanbot;

import android.content.Intent;
import android.os.Bundle;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;
import com.google.android.gms.common.SignInButton;
import com.google.firebase.auth.FirebaseAuth;
import com.google.firebase.auth.FirebaseUser;

public class LoginActivity extends AppCompatActivity {

  private FirebaseAuth mAuth;
  private EditText etEmail, etPassword;
  private Button btnLogin;
  private SignInButton btnGoogle;

  @Override
  protected void onStart() {
    super.onStart();
    // --- 1. SESSION CHECK ---
    // If user is already logged in, send them straight to MainActivity
    mAuth = FirebaseAuth.getInstance();
    FirebaseUser currentUser = mAuth.getCurrentUser();
    if (currentUser != null) {
      startActivity(new Intent(LoginActivity.this, MainActivity.class));
      finish();
    }
  }

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_login);

    mAuth = FirebaseAuth.getInstance();
    etEmail = findViewById(R.id.etEmail);
    etPassword = findViewById(R.id.etPassword);
```

```
        btnLogin = findViewById(R.id.btnLogin);
        btnGoogle = findViewById(R.id.btnGoogle);

        // 2. Standard Email Login
        btnLogin.setOnClickListener(v -> {
            String email = etEmail.getText().toString().trim();
            String pass = etPassword.getText().toString().trim();

            if (email.isEmpty() || pass.isEmpty()) {
                Toast.makeText(this, "Please enter email and password",
Toast.LENGTH_SHORT).show();
                return;
            }

            mAuth.signInWithEmailAndPassword(email, pass)
                    .addOnCompleteListener(task -> {
                        if (task.isSuccessful()) {
                            // CRITICAL: Clear local SQLite DB so the new user starts fresh
                            // before downloading their own history from Firebase
                            new Thread(() -> {
                                ChatDatabase.getInstance(LoginActivity.this).chatDao().deleteAll();
                                runOnUiThread(() -> {
                                    startActivity(new Intent(LoginActivity.this, MainActivity.class));
                                    finish();
                                });
                            }).start();
                        } else {
                            Toast.makeText(LoginActivity.this, "Login Failed: " +
                                    task.getException().getMessage(), Toast.LENGTH_SHORT).show();
                        }
                    });
        });

        // --- 3. SIGN UP REDIRECT (Updated) ---
        findViewById(R.id.tvSignUp).setOnClickListener(v -> {
            // This now points to the SignupActivity we planned
            startActivity(new Intent(LoginActivity.this, SignupActivity.class));
        });

        // 4. Google Sign In placeholder
        btnGoogle.setOnClickListener(v -> {
            Toast.makeText(this, "Google Sign-In clicked!", Toast.LENGTH_SHORT).show();
        });
    }
}
```

SignupActivity.java

```
package com.example.makanbot;

import androidx.appcompat.app.AppCompatActivity;
```

```
import android.content.Intent;
import android.os.Bundle;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;

import com.google.firebase.auth.FirebaseAuth;
import com.google.firebase.auth.FirebaseUser;

public class SignupActivity extends AppCompatActivity {
    private EditText etName, etEmail, etPassword, etConfirmPassword;
    private Button btnSignup;
    private FirebaseAuth mAuth;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_signup);

        // Initialize Firebase Auth
        mAuth = FirebaseAuth.getInstance();

        etName = findViewById(R.id.etName);
        etEmail = findViewById(R.id.etEmail);
        etPassword = findViewById(R.id.etPassword);
        etConfirmPassword = findViewById(R.id.etConfirmPassword);
        btnSignup = findViewById(R.id.btnSignup);

        btnSignup.setOnClickListener(v -> handleSignup());
    }

    private void handleSignup() {
        String name = etName.getText().toString().trim();
        String email = etEmail.getText().toString().trim();
        String pass = etPassword.getText().toString();
        String confirmPass = etConfirmPassword.getText().toString();

        // 1. Validation Checks
        if (name.isEmpty() || email.isEmpty() || pass.isEmpty()) {
            Toast.makeText(this, "Please fill all fields", Toast.LENGTH_SHORT).show();
            return;
        }

        if (!pass.equals(confirmPass)) {
            Toast.makeText(this, "Passwords do not match", Toast.LENGTH_SHORT).show();
            return;
        }

        if (pass.length() < 6) {
```

```
            Toast.makeText(this, "Password must be at least 6 characters",
Toast.LENGTH_SHORT).show();
            return;
        }

        // 2. Firebase Authentication ONLY
        mAuth.createUserWithEmailAndPassword(email, pass).addOnCompleteListener(task ->
{
            if (task.isSuccessful()) {
                FirebaseUser user = mAuth.getCurrentUser();
                if (user != null) {
                    // Success! Move to PreferencesActivity immediately
                    // We pass the UID and Name so the next activity can save them to Firestore
later
                    Intent intent = new Intent(SignupActivity.this, PreferencesActivity.class);
                    intent.putExtra("USER_ID", user.getUid());
                    intent.putExtra("USER_NAME", name);
                    intent.putExtra("USER_EMAIL", email);

                    startActivity(intent);
                    finish();
                }
            } else {
                // If Auth fails (e.g., email already exists), show why
                String error = task.getException() != null ? task.getException().getMessage() :
"Unknown error";
                Toast.makeText(this, "Signup Failed: " + error, Toast.LENGTH_LONG).show();
            }
        });
    }
}
```

PreferencesActivity.java

```
package com.example.makanbot;

import android.content.Intent;
import android.content.res.ColorStateList;
import android.graphics.Color;
import android.os.Bundle;
import android.util.Log;
import android.widget.Button;
import android.widget.Toast;

import androidx.appcompat.app.AppCompatActivity;

import com.google.firebase.auth.FirebaseAuth;
import com.google.firebase.firestore.FirebaseFirestore;

import java.util.ArrayList;
import java.util.HashMap;
```

```
import java.util.List;
import java.util.Map;

public class PreferencesActivity extends AppCompatActivity {

    private List<String> selectedPrefs = new ArrayList<>();
    private FirebaseFirestore db;
    private FirebaseAuth mAuth;

    private String userId;
    private String userName;
    private String userEmail;
    private boolean isEditMode = false;  // NEW FLAG

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_preferences);

        db = FirebaseFirestore.getInstance();
        mAuth = FirebaseAuth.getInstance();

        // Retrieve Intent data
        isEditMode = getIntent().getBooleanExtra("EDIT_MODE", false);
        userId = getIntent().getStringExtra("USER_ID");
        userName = getIntent().getStringExtra("USER_NAME");
        userEmail = getIntent().getStringExtra("USER_EMAIL");

        // UI Setup
        setupButton(R.id.btnWestern, "Western");
        setupButton(R.id.btnChinese, "Chinese");
        setupButton(R.id.btnMalay, "Malay");
        setupButton(R.id.btnJapanese, "Japanese");
        setupButton(R.id.btnIndian, "Indian");
        setupButton(R.id.btnCafe, "Cafe");

        Button btnSave = findViewById(R.id.btnSavePrefs);

        // Change button label based on mode
        if (isEditMode) {
            btnSave.setText("Update Preferences");
            loadExistingPreferences();  // Pre-select current prefs
        } else {
            btnSave.setText("Save Preferences");
        }

        btnSave.setOnClickListener(v -> {
            if (selectedPrefs.isEmpty()) {
                Toast.makeText(this, "Please select at least one preference",
```

```
Toast.LENGTH_SHORT).show();
            } else {
                if (isEditMode) {
                    updateUserPreferences();
                } else {
                    saveUserToFirestore();
                }
            }
        });
    }

    // Load existing preferences and visually pre-select them
    private void loadExistingPreferences() {
        String uid = (userId != null) ? userId : mAuth.getUid();
        if (uid == null) return;

        db.collection("users").document(uid).get()
                .addOnSuccessListener(doc -> {
                    if (doc.exists()) {
                        List<String> existing = (List<String>) doc.get("preference");
                        if (existing != null) {
                            for (String pref : existing) {
                                selectedPrefs.add(pref);
                                highlightButtonForPref(pref);
                            }
                        }
                    }
                });
    }

    private void highlightButtonForPref(String pref) {
        Button btn = null;
        switch (pref) {
            case "Western":  btn = findViewById(R.id.btnWestern);  break;
            case "Chinese":  btn = findViewById(R.id.btnChinese);  break;
            case "Malay":    btn = findViewById(R.id.btnMalay);    break;
            case "Japanese": btn = findViewById(R.id.btnJapanese); break;
            case "Indian":   btn = findViewById(R.id.btnIndian);   break;
            case "Cafe":     btn = findViewById(R.id.btnCafe);     break;
        }
        if (btn != null) {
            btn.setBackgroundTintList(ColorStateList.valueOf(Color.parseColor("#4CAF50")));
            btn.setTextColor(Color.WHITE);
        }
    }

    private void setupButton(int id, String type) {
        Button btn = findViewById(id);
        if (btn == null) return;
```

```
        btn.setOnClickListener(v -> {
            if (selectedPrefs.contains(type)) {
                selectedPrefs.remove(type);

btn.setBackgroundTintList(ColorStateList.valueOf(Color.parseColor("#E0E0E0")));
                btn.setTextColor(Color.BLACK);
            } else {
                if (selectedPrefs.size() < 3) {
                    selectedPrefs.add(type);

btn.setBackgroundTintList(ColorStateList.valueOf(Color.parseColor("#4CAF50")));
                    btn.setTextColor(Color.WHITE);
                } else {
                    Toast.makeText(this, "Maximum 3 allowed", Toast.LENGTH_SHORT).show();
                }
            }
        });
    }

    // For first-time signup (unchanged)
    private void saveUserToFirestore() {
        String uid = (userId != null) ? userId : mAuth.getUid();
        if (uid == null) {
            Toast.makeText(this, "Session Error. Please sign up again.",
Toast.LENGTH_LONG).show();
            return;
        }

        Map<String, Object> userProfile = new HashMap<>();
        userProfile.put("uid", uid);
        userProfile.put("name", userName != null ? userName : "User");
        userProfile.put("email", userEmail != null ? userEmail : "");
        userProfile.put("preference", selectedPrefs);
        userProfile.put("gemini_api_key",
"AIzaSyB3w42aQnlL8JMrrOQeiQHPKFsLsfL24qQ");
        userProfile.put("setup_complete", true);

        db.collection("users").document(uid)
                .set(userProfile)
                .addOnSuccessListener(aVoid -> {
                    Intent intent = new Intent(PreferencesActivity.this, MainActivity.class);
                    intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
Intent.FLAG_ACTIVITY_CLEAR_TASK);
                    startActivity(intent);
                    finish();
                })
                .addOnFailureListener(e ->
                        Toast.makeText(this, "Database Error: " + e.getMessage(),
```

```
Toast.LENGTH_LONG).show());
    }

    // NEW METHOD  For editing from ProfileActivity
    private void updateUserPreferences() {
        String uid = (userId != null) ? userId : mAuth.getUid();
        if (uid == null) {
            Toast.makeText(this, "Session Error", Toast.LENGTH_SHORT).show();
            return;
        }

        db.collection("users").document(uid)
                .update("preference", selectedPrefs)
                .addOnSuccessListener(aVoid -> {
                    Toast.makeText(this, "Preferences Updated!", Toast.LENGTH_SHORT).show();
                    finish();  // returns to ProfileActivity, which reloads via onResume()
                })
                .addOnFailureListener(e ->
                        Toast.makeText(this, "Update Failed: " + e.getMessage(),
Toast.LENGTH_SHORT).show());
    }
}
```

MainActivity.java

```
package com.example.makanbot;

import android.content.Intent;
import android.os.Bundle;
import android.view.MenuItem;
import android.widget.EditText;
import android.widget.ImageButton;
import android.widget.Toast;

import androidx.annotation.NonNull;
import androidx.appcompat.app.ActionBarDrawerToggle;
import androidx.appcompat.app.AppCompatActivity;
import androidx.appcompat.widget.Toolbar;
import androidx.core.view.GravityCompat;
import androidx.drawerlayout.widget.DrawerLayout;
import androidx.recyclerview.widget.LinearLayoutManager;
import androidx.recyclerview.widget.RecyclerView;

import com.google.android.material.navigation.NavigationView;
import com.google.firebase.auth.FirebaseAuth;
import com.google.firebase.firestore.FirebaseFirestore;
import com.google.firebase.firestore.Query;

import org.json.JSONArray;
import org.json.JSONObject;
```

APPENDIX

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;

import okhttp3.Call;
import okhttp3.Callback;
import okhttp3.MediaType;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.RequestBody;
import okhttp3.Response;

public class MainActivity extends AppCompatActivity implements
NavigationView.OnNavigationItemSelectedListener {

    private DrawerLayout drawerLayout;
    private RecyclerView chatRecyclerView;
    private EditText etMessage;
    private ImageButton btnSend;

    private ChatAdapter chatAdapter;
    private List<ChatMessage> messageList;
    private ChatDatabase db;
    private OkHttpClient client;
    private FirebaseFirestore firestore;
    private String currentUserId;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        firestore = FirebaseFirestore.getInstance();
        currentUserId = FirebaseAuth.getInstance().getUid();

        client = new OkHttpClient.Builder()
                .connectTimeout(120, TimeUnit.SECONDS)
                .writeTimeout(120, TimeUnit.SECONDS)
                .readTimeout(120, TimeUnit.SECONDS)
                .build();

        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        drawerLayout = findViewById(R.id.drawer_layout);
        NavigationView navigationView = findViewById(R.id.nav_view);
        navigationView.setNavigationItemSelectedListener(this);
```

```
        ActionBarDrawerToggle toggle = new ActionBarDrawerToggle(this, drawerLayout,
toolbar,
                R.string.navigation_drawer_open, R.string.navigation_drawer_close);
        drawerLayout.addDrawerListener(toggle);
        toggle.syncState();

        etMessage = findViewById(R.id.etMessage);
        btnSend = findViewById(R.id.btnSend);
        chatRecyclerView = findViewById(R.id.chatRecyclerView);

        db = ChatDatabase.getInstance(this);
        messageList = new ArrayList<>();

        chatAdapter = new ChatAdapter(messageList);
        LinearLayoutManager layoutManager = new LinearLayoutManager(this);
        layoutManager.setStackFromEnd(true);
        chatRecyclerView.setLayoutManager(layoutManager);
        chatRecyclerView.setAdapter(chatAdapter);

        loadChatHistory();

        btnSend.setOnClickListener(v -> {
            String text = etMessage.getText().toString().trim();
            if (!text.isEmpty()) {
                handleUserMessage(text);
            }
        });
    }

    private void handleUserMessage(String text) {
        ChatMessage userMsg = new ChatMessage("User", text, System.currentTimeMillis(),
false);
        saveAndDisplayMessage(userMsg);
        etMessage.setText("");

        if (text.toLowerCase().contains("more") || text.toLowerCase().contains("any other") ||
text.toLowerCase().contains("suggest another")) {
            if (handleShowMoreLocally()) {
                return;
            }
        }

        fetchBotResponse(text);
    }

    private boolean handleShowMoreLocally() {
        for (int i = messageList.size() - 1; i >= 0; i--) {
            ChatMessage msg = messageList.get(i);
            if (msg.isRecommendation() && msg.getAllRecommendationsJson() != null) {
```

```
            try {
                JSONArray recs = new JSONArray(msg.getAllRecommendationsJson());
                if (recs.length() > 1) {
                    JSONObject nextRes = recs.getJSONObject(1);
                    ChatMessage moreMsg = new ChatMessage("Bot", "Here is another top
suggestion for you:", System.currentTimeMillis(), true);
                    mapJsonToMessage(moreMsg, nextRes);
                    saveAndDisplayMessage(moreMsg);
                    return true;
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
    return false;
  }

  private void fetchBotResponse(String text) {
    ChatMessage loadingMsg = new ChatMessage("Bot", "", System.currentTimeMillis(),
false);
    loadingMsg.setLoading(true);

    messageList.add(loadingMsg);
    int loadingPosition = messageList.size() - 1;
    chatAdapter.notifyItemInserted(loadingPosition);
    chatRecyclerView.scrollToPosition(loadingPosition);

    JSONObject json = new JSONObject();
    try {
      json.put("user_id", FirebaseAuth.getInstance().getUid());
      json.put("message", text);
    } catch (Exception e) { e.printStackTrace(); }

    RequestBody body = RequestBody.create(json.toString(),
MediaType.get("application/json; charset=utf-8"));
    Request request = new Request.Builder()
        .url("http://10.0.2.2:8000/chat")
        .post(body)
        .build();

    client.newCall(request).enqueue(new Callback() {
      @Override
      public void onFailure(@NonNull Call call, @NonNull IOException e) {
        runOnUiThread(() -> {
          messageList.remove(loadingPosition);
          chatAdapter.notifyItemRemoved(loadingPosition);
          Toast.makeText(MainActivity.this, "Network Error",
Toast.LENGTH_LONG).show();
```

```
            });
        }

        @Override
        public void onResponse(@NonNull Call call, @NonNull Response response) throws
IOException {
            if (response.isSuccessful()) {
                try {
                    String responseData = response.body().string();
                    JSONObject obj = new JSONObject(responseData);
                    String botReply = obj.getString("bot_response");
                    boolean isRec = obj.getBoolean("is_recommendation");

                    runOnUiThread(() -> {
                        loadingMsg.setLoading(false);
                        loadingMsg.setContent(botReply);
                        loadingMsg.setRecommendation(isRec);

                        if (isRec) {
                            try {
                                JSONArray recs = obj.getJSONArray("recommendations");
                                loadingMsg.setAllRecommendationsJson(recs.toString());
                                if (recs.length() > 0) {
                                    mapJsonToMessage(loadingMsg, recs.getJSONObject(0));
                                }
                            } catch (Exception e) { e.printStackTrace(); }
                        }

                        new Thread(() -> db.chatDao().insert(loadingMsg)).start();
                        chatAdapter.notifyItemChanged(loadingPosition);
                    });
                } catch (Exception e) { e.printStackTrace(); }
            }
        }
    });
  }

  private void mapJsonToMessage(ChatMessage msg, JSONObject data) throws Exception
{
    msg.setRestaurantName(data.getString("name"));
    msg.setAddress(data.getString("address"));
    msg.setHalal(data.getString("halal"));
    msg.setExplanation(data.getString("evidence"));
    msg.setRestaurantImageUrl(data.optString("image_url", ""));

    double score = data.optDouble("score", 0.0);
    msg.setMatchScore(Math.round(score * 100) + "% Match");

    msg.setLocationUrl(data.optString("location_link", ""));
```

```
        msg.setReservationUrl(data.optString("reservation_link", ""));
        msg.setMenuUrl(data.optString("menu_link", ""));
    }

    private void saveAndDisplayMessage(ChatMessage msg) {
        new Thread(() -> db.chatDao().insert(msg)).start();

        if (currentUserId != null) {
            firestore.collection("users")
                    .document(currentUserId)
                    .collection("conversations")
                    .add(msg)
                    .addOnFailureListener(e -> Toast.makeText(this, "Cloud Sync Failed",
Toast.LENGTH_SHORT).show());
        }

        runOnUiThread(() -> {
            messageList.add(msg);
            chatAdapter.notifyItemInserted(messageList.size() - 1);
            chatRecyclerView.scrollToPosition(messageList.size() - 1);
        });
    }

    private void loadChatHistory() {
        new Thread(() -> {
            List<ChatMessage> history = db.chatDao().getAllMessages();
            runOnUiThread(() -> {
                if (history.isEmpty()) {
                    fetchHistoryFromFirebase();
                } else {
                    messageList.addAll(history);
                    chatAdapter.notifyDataSetChanged();
                    chatRecyclerView.scrollToPosition(messageList.size() - 1);
                }
            });
        }).start();
    }

    private void fetchHistoryFromFirebase() {
        firestore.collection("users")
                .document(currentUserId)
                .collection("conversations")
                .orderBy("timestamp", Query.Direction.ASCENDING)
                .get()
                .addOnSuccessListener(queryDocumentSnapshots -> {
                    List<ChatMessage> cloudHistory =
queryDocumentSnapshots.toObjects(ChatMessage.class);
                    if (cloudHistory.isEmpty()) {
                        saveAndDisplayMessage(new ChatMessage("Bot", "Hi! I'm MakanBot, your
```

AI-powered restaurant recommender for Kuala Lumpur. I can help you discover great places to eat based on your cuisine preferences, halal needs, and even restaurants similar to ones you've loved before. Just type what you feel like eating, and let's get started!
\uD83C\uDF74", System.currentTimeMillis(), false));
                } else {
                    messageList.addAll(cloudHistory);
                    chatAdapter.notifyDataSetChanged();
                    new Thread(() -> {
                        for (ChatMessage m : cloudHistory) db.chatDao().insert(m);
                    }).start();
                }
            });
    }

    @Override
    public boolean onNavigationItemSelected(@NonNull MenuItem item) {
        int id = item.getItemId();

        if (id == R.id.nav_new_chat) {
            new Thread(() -> {
                db.chatDao().deleteAll();
                runOnUiThread(() -> {
                    messageList.clear();
                    chatAdapter.notifyDataSetChanged();
                    Toast.makeText(this, "Chat history cleared", Toast.LENGTH_SHORT).show();
                });
            }).start();

        } else if (id == R.id.nav_favorites) {
            // 🆕 Open Favorites page
            Intent intent = new Intent(MainActivity.this, FavoritesActivity.class);
            startActivity(intent);

        } else if (id == R.id.nav_profile) {
            Intent intent = new Intent(MainActivity.this, ProfileActivity.class);
            startActivity(intent);

        } else if (id == R.id.nav_logout) {
            FirebaseAuth.getInstance().signOut();
            new Thread(() -> {
                db.chatDao().deleteAll();
                runOnUiThread(() -> {
                    Intent intent = new Intent(MainActivity.this, LoginActivity.class);
                    intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
Intent.FLAG_ACTIVITY_CLEAR_TASK);
                    startActivity(intent);
                    finish();
                });
            }).start();

```
        }

        drawerLayout.closeDrawer(GravityCompat.START);
        return true;
    }

    @Override
    public void onBackPressed() {
        if (drawerLayout.isDrawerOpen(GravityCompat.START)) {
            drawerLayout.closeDrawer(GravityCompat.START);
        } else {
            super.onBackPressed();
        }
    }
}
```

ChatAdapter.java

```
package com.example.makanbot;

import android.content.Intent;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.TextView;
import androidx.annotation.NonNull;
import androidx.recyclerview.widget.RecyclerView;
import com.airbnb.lottie.LottieAnimationView;
import java.util.List;

public class ChatAdapter extends RecyclerView.Adapter<RecyclerView.ViewHolder> {

    private static final int VIEW_TYPE_USER = 1;
    private static final int VIEW_TYPE_BOT = 2;
    private List<ChatMessage> messageList;

    public ChatAdapter(List<ChatMessage> messageList) {
        this.messageList = messageList;
    }

    @Override
    public int getItemViewType(int position) {
        if (messageList.get(position).getSender().equalsIgnoreCase("User")) {
            return VIEW_TYPE_USER;
        } else {
            return VIEW_TYPE_BOT;
        }
    }

    @NonNull
```

```
    @Override
    public RecyclerView.ViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int
viewType) {
        if (viewType == VIEW_TYPE_USER) {
            View view =
LayoutInflater.from(parent.getContext()).inflate(R.layout.item_chat_user, parent, false);
            return new UserViewHolder(view);
        } else {
            View view =
LayoutInflater.from(parent.getContext()).inflate(R.layout.item_chat_bot, parent, false);
            return new BotViewHolder(view);
        }
    }

    @Override
    public void onBindViewHolder(@NonNull RecyclerView.ViewHolder holder, int
position) {
        ChatMessage message = messageList.get(position);

        if (holder instanceof UserViewHolder) {
            ((UserViewHolder) holder).tvUserMessage.setText(message.getMessage());
        } else {
            BotViewHolder botHolder = (BotViewHolder) holder;

            // Reset the bubble container visibility just in case
            View bubbleContainer =
botHolder.itemView.findViewById(R.id.botBubbleContainer);

            if (message.isLoading()) {
                // 1. STATE: TYPING
                botHolder.typingAnimation.setVisibility(View.VISIBLE);
                botHolder.typingAnimation.playAnimation();

                // Hide the actual text so the bubble only shows the dots
                botHolder.tvBotMessage.setVisibility(View.GONE);

                // Ensure the bubble background is visible
                if (bubbleContainer != null) bubbleContainer.setVisibility(View.VISIBLE);

                // Hide the recommendation card until the bot is done "thinking"
                botHolder.cardView.setVisibility(View.GONE);

            } else {
                // 2. STATE: MESSAGE RECEIVED
                botHolder.typingAnimation.setVisibility(View.GONE);
                botHolder.typingAnimation.cancelAnimation();

                botHolder.tvBotMessage.setVisibility(View.VISIBLE);
                botHolder.tvBotMessage.setText(message.getMessage());
```

```
            // --- RECOMMENDATION CARD LOGIC ---
            if (message.isRecommendation()) {
                botHolder.cardView.setVisibility(View.VISIBLE);
                botHolder.tvResName.setText(message.getRestaurantName());

                String cuisine = message.getRestaurantCuisine() != null ?
message.getRestaurantCuisine() : "Restaurant";
                String shortDetails = cuisine + " • " + message.getMatchScore();
                botHolder.tvResDetails.setText(shortDetails);

                // Open Detail Activity on Click
                botHolder.cardView.setOnClickListener(v -> {
                    Intent intent = new Intent(v.getContext(), RestaurantDetailActivity.class);
                    intent.putExtra("name", message.getRestaurantName());
                    intent.putExtra("rating", message.getMatchScore());
                    intent.putExtra("halal", message.getHalal());
                    intent.putExtra("address", message.getAddress());
                    intent.putExtra("evidence", message.getExplanation());
                    intent.putExtra("image_url", message.getRestaurantImageUrl());
                    intent.putExtra("loc_url", message.getLocationUrl());
                    intent.putExtra("res_url", message.getReservationUrl());
                    intent.putExtra("menu_url", message.getMenuUrl());
                    v.getContext().startActivity(intent);
                });
            } else {
                botHolder.cardView.setVisibility(View.GONE);
            }
        }
    }
}

@Override
public int getItemCount() {
    return messageList.size();
}

// --- VIEW HOLDERS ---

static class UserViewHolder extends RecyclerView.ViewHolder {
    TextView tvUserMessage;
    UserViewHolder(View itemView) {
        super(itemView);
        tvUserMessage = itemView.findViewById(R.id.tvUserMessage);
    }
}

static class BotViewHolder extends RecyclerView.ViewHolder {
    TextView tvBotMessage;
```

```
        View cardView;
        TextView tvResName, tvResDetails;
        LottieAnimationView typingAnimation;

        BotViewHolder(View itemView) {
            super(itemView);
            tvBotMessage = itemView.findViewById(R.id.tvBotMessage);
            cardView = itemView.findViewById(R.id.recommendation_card);
            tvResName = itemView.findViewById(R.id.resName);
            tvResDetails = itemView.findViewById(R.id.resDetails);
            typingAnimation = itemView.findViewById(R.id.typingAnimation);
        }
    }
}
```

ChatMessage.java

```
package com.example.makanbot;

import androidx.room.Entity;
import androidx.room.PrimaryKey;
import androidx.room.Ignore;

@Entity(tableName = "chat_messages")
public class ChatMessage {

    @PrimaryKey(autoGenerate = true)
    private int id;

    private String sender;        // "User" or "Bot"
    private String message;
    private long timestamp;
    private boolean isRecommendation;

    // --- FIX 1: DECLARE THE VARIABLE AND TELL ROOM TO IGNORE IT ---
    @Ignore
    private boolean isLoading = false;

    // --- RECOMMENDATION ENGINE FIELDS ---
    private String restaurantName;
    private String allRecommendationsJson;
    private String restaurantImageUrl;
    private String restaurantCuisine;
    private String matchScore;
    private String reservationUrl;
    private String explanation;
    private String popularDishes;
    private String menuUrl;
    private String address;
    private String halal;
    private String locationUrl;
```

```
    // --- FIX 2: ROOM WILL USE THIS CONSTRUCTOR ---

    // 1. Required for Firebase
    public ChatMessage() {
    }
    public ChatMessage(String sender, String message, long timestamp, boolean
isRecommendation) {
        this.sender = sender;
        this.message = message;
        this.timestamp = timestamp;
        this.isRecommendation = isRecommendation;
    }

    // --- FIX 3: ROOM WILL IGNORE THIS CONSTRUCTOR ---
    @Ignore
    public ChatMessage(String sender, String message, boolean isRecommendation) {
        this.sender = sender;
        this.message = message;
        this.timestamp = System.currentTimeMillis();
        this.isRecommendation = isRecommendation;
        this.isLoading = false; // Now this will work because the variable is declared
    }

    // --- GETTERS AND SETTERS ---

    // UI Helper methods
    public boolean isLoading() { return isLoading; }
    public void setLoading(boolean loading) { this.isLoading = loading; }

    public void setContent(String content) { this.message = content; }
    public String getContent() { return this.message; }

    // Database getters/setters
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getSender() { return sender; }
    public void setSender(String sender) { this.sender = sender; }

    public String getMessage() { return message; }
    public void setMessage(String message) { this.message = message; }

    public String getAllRecommendationsJson() { return allRecommendationsJson; }
    public void setAllRecommendationsJson(String json) { this.allRecommendationsJson =
json; }

    public String getRestaurantImageUrl() { return restaurantImageUrl; }
    public void setRestaurantImageUrl(String url) { this.restaurantImageUrl = url; }
```

```
    public long getTimestamp() { return timestamp; }
    public void setTimestamp(long timestamp) { this.timestamp = timestamp; }

    public boolean isRecommendation() { return isRecommendation; }
    public void setRecommendation(boolean recommendation) { isRecommendation =
recommendation; }

    public String getRestaurantName() { return restaurantName; }
    public void setRestaurantName(String restaurantName) { this.restaurantName =
restaurantName; }

    public String getRestaurantCuisine() { return restaurantCuisine; }
    public void setRestaurantCuisine(String restaurantCuisine) { this.restaurantCuisine =
restaurantCuisine; }

    public String getMatchScore() { return matchScore; }
    public void setMatchScore(String matchScore) { this.matchScore = matchScore; }

    public String getReservationUrl() { return reservationUrl; }
    public void setReservationUrl(String reservationUrl) { this.reservationUrl =
reservationUrl; }

    public String getExplanation() { return explanation; }
    public void setExplanation(String explanation) { this.explanation = explanation; }

    public String getPopularDishes() { return popularDishes; }
    public void setPopularDishes(String popularDishes) { this.popularDishes = popularDishes;
}

    public String getMenuUrl() { return menuUrl; }
    public void setMenuUrl(String menuUrl) { this.menuUrl = menuUrl; }

    public String getAddress() { return address; }
    public void setAddress(String address) { this.address = address; }

    public String getHalal() { return halal; }
    public void setHalal(String halal) { this.halal = halal; }

    public String getLocationUrl() { return locationUrl; }
    public void setLocationUrl(String url) { this.locationUrl = url; }
}
```

ChatDao.java

```
package com.example.makanbot;

import androidx.room.Dao;
import androidx.room.Insert;
import androidx.room.Query;
import com.example.makanbot.ChatMessage;
```

```
import java.util.List;

@Dao
public interface ChatDao {
    @Insert
    void insert(ChatMessage message);

    @Query("SELECT * FROM chat_messages ORDER BY timestamp ASC")
    List<ChatMessage> getAllMessages();

    @Query("DELETE FROM chat_messages")
    void deleteAll();
}
```

ChatDatabase.java

```
package com.example.makanbot;

import android.content.Context;
import androidx.room.Database;
import androidx.room.Room;
import androidx.room.RoomDatabase;

// --- FIXED: Added exportSchema = false to remove the warning ---
@Database(entities = {ChatMessage.class}, version = 1, exportSchema = false)
public abstract class ChatDatabase extends RoomDatabase {

    public abstract ChatDao chatDao();

    private static volatile ChatDatabase instance;

    public static ChatDatabase getInstance(Context context) {
        if (instance == null) {
            synchronized (ChatDatabase.class) {
                if (instance == null) {
                    instance = Room.databaseBuilder(context.getApplicationContext(),
                            ChatDatabase.class, "makanbot_local_db")
                        // Useful for FYP: If you add/change fields in ChatMessage,
                        // this prevents the app from crashing by recreating the DB.
                        .fallbackToDestructiveMigration()
                        .build();
                }
            }
        }
        return instance;
    }
}
```

RestaurantDetailActivity.java

```
package com.example.makanbot;

import android.content.Intent;
```

```
import android.net.Uri;
import android.os.Bundle;
import android.widget.Button;
import android.widget.ImageView;
import android.widget.TextView;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;
import com.bumptech.glide.Glide;
import com.google.android.material.floatingactionbutton.FloatingActionButton;
import com.google.firebase.auth.FirebaseAuth;
import com.google.firebase.firestore.DocumentReference;
import com.google.firebase.firestore.FirebaseFirestore;

public class RestaurantDetailActivity extends AppCompatActivity {

    private ImageView ivRestaurant;
    private TextView tvName, tvRating, tvHalalStatus, tvAddress, tvEvidence;
    private Button btnLocation, btnReservation, btnMenu;
    private FloatingActionButton btnSave;

    // Intent data kept as fields so the save handler can use them
    private String name, rating, halal, address, evidence, imageUrl;
    private String locUrl, resUrl, menuUrl;

    private FirebaseFirestore firestore;
    private String uid;
    private boolean isFavorited = false;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_restaurant_detail);

        // Initialize UI
        ivRestaurant = findViewById(R.id.ivRestaurant);
        tvName = findViewById(R.id.tvName);
        tvRating = findViewById(R.id.tvRating);
        tvHalalStatus = findViewById(R.id.tvHalalStatus);
        tvAddress = findViewById(R.id.tvAddress);
        tvEvidence = findViewById(R.id.tvEvidence);
        btnLocation = findViewById(R.id.btnLocation);
        btnReservation = findViewById(R.id.btnReservation);
        btnMenu = findViewById(R.id.btnMenu);
        btnSave = findViewById(R.id.btnSave);

        // Firebase
        firestore = FirebaseFirestore.getInstance();
        uid = FirebaseAuth.getInstance().getUid();
```

```
        // Get Data from Intent
        Intent intent = getIntent();
        name = intent.getStringExtra("name");
        rating = intent.getStringExtra("rating");
        halal = intent.getStringExtra("halal");
        address = intent.getStringExtra("address");
        evidence = intent.getStringExtra("evidence");
        imageUrl = intent.getStringExtra("image_url");
        locUrl = intent.getStringExtra("loc_url");
        resUrl = intent.getStringExtra("res_url");
        menuUrl = intent.getStringExtra("menu_url");

        // Set Data to Views
        tvName.setText(name);
        tvRating.setText("⭐ " + rating);
        tvHalalStatus.setText("Halal: " + halal);
        tvAddress.setText(address);
        tvEvidence.setText(evidence);

        if (imageUrl != null && !imageUrl.isEmpty()) {
            Glide.with(this).load(imageUrl).into(ivRestaurant);
        }

        // Button Click Listeners
        btnLocation.setOnClickListener(v -> openUrl(locUrl));
        btnReservation.setOnClickListener(v -> openUrl(resUrl));
        btnMenu.setOnClickListener(v -> openUrl(menuUrl));

        // Check if this restaurant is already a favorite on open, then toggle on tap
        checkIfFavorited();
        btnSave.setOnClickListener(v -> toggleFavorite());
    }

    /** Check Firestore to see if this restaurant is already saved, so the star shows correctly. */
    private void checkIfFavorited() {
        if (uid == null || name == null) return;

        getFavDocRef().get()
                .addOnSuccessListener(doc -> {
                    isFavorited = doc.exists();
                    updateStarIcon();
                });
    }

    /** Tap handler: add to Firestore if not saved, remove if already saved. */
    private void toggleFavorite() {
        if (uid == null) {
            Toast.makeText(this, "Please log in to save favorites",
Toast.LENGTH_SHORT).show();
```

```
            return;
        }
        if (name == null || name.isEmpty()) {
            Toast.makeText(this, "Cannot save this restaurant",
Toast.LENGTH_SHORT).show();
            return;
        }

        DocumentReference ref = getFavDocRef();

        if (isFavorited) {
            // Already saved -> remove
            ref.delete()
                    .addOnSuccessListener(aVoid -> {
                        isFavorited = false;
                        updateStarIcon();
                        Toast.makeText(this, name + " removed from favorites",
Toast.LENGTH_SHORT).show();
                    })
                    .addOnFailureListener(e ->
                            Toast.makeText(this, "Error: " + e.getMessage(),
Toast.LENGTH_SHORT).show());
        } else {
            // Not saved -> add
            FavoriteRestaurant fav = new FavoriteRestaurant(
                    name,
                    address,
                    halal,
                    rating,
                    evidence,
                    imageUrl,
                    locUrl,
                    resUrl,
                    menuUrl,
                    System.currentTimeMillis()
            );

            ref.set(fav)
                    .addOnSuccessListener(aVoid -> {
                        isFavorited = true;
                        updateStarIcon();
                        Toast.makeText(this, name + " saved to favorites!",
Toast.LENGTH_SHORT).show();
                    })
                    .addOnFailureListener(e ->
                            Toast.makeText(this, "Error: " + e.getMessage(),
Toast.LENGTH_SHORT).show());
        }
    }
```

```
    /** Filled star if saved, empty star if not. */
    private void updateStarIcon() {
        if (isFavorited) {
            btnSave.setImageResource(android.R.drawable.btn_star_big_on);
        } else {
            btnSave.setImageResource(android.R.drawable.btn_star_big_off);
        }
    }

    /** Firestore path: users/{uid}/favorites/{restaurantName} */
    private DocumentReference getFavDocRef() {
        return firestore.collection("users")
                .document(uid)
                .collection("favorites")
                .document(name);
    }

    private void openUrl(String url) {
        if (url != null && !url.isEmpty()) {
            Intent i = new Intent(Intent.ACTION_VIEW, Uri.parse(url));
            startActivity(i);
        } else {
            Toast.makeText(this, "Link not available", Toast.LENGTH_SHORT).show();
        }
    }
}
```

FavoritesActivity.java

```
package com.example.makanbot;

import android.os.Bundle;
import android.view.MenuItem;
import android.view.View;
import android.widget.ProgressBar;
import android.widget.TextView;
import android.widget.Toast;

import androidx.appcompat.app.AppCompatActivity;
import androidx.appcompat.widget.Toolbar;
import androidx.recyclerview.widget.LinearLayoutManager;
import androidx.recyclerview.widget.RecyclerView;

import com.google.firebase.auth.FirebaseAuth;
import com.google.firebase.firestore.FirebaseFirestore;
import com.google.firebase.firestore.Query;

import java.util.ArrayList;
import java.util.List;
```

```
public class FavoritesActivity extends AppCompatActivity {

    private RecyclerView recyclerFavs;
    private FavoritesAdapter adapter;
    private List<FavoriteRestaurant> favoritesList;
    private ProgressBar progressBar;
    private TextView tvEmpty;

    private FirebaseFirestore firestore;
    private String uid;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_favorites);

        // Toolbar with back arrow
        Toolbar toolbar = findViewById(R.id.toolbarFavorites);
        setSupportActionBar(toolbar);
        if (getSupportActionBar() != null) {
            getSupportActionBar().setDisplayHomeAsUpEnabled(true);
            getSupportActionBar().setTitle("My Favorites");
        }

        recyclerFavs = findViewById(R.id.recyclerFavorites);
        progressBar = findViewById(R.id.progressFavorites);
        tvEmpty = findViewById(R.id.tvEmptyFavorites);

        favoritesList = new ArrayList<>();
        adapter = new FavoritesAdapter(this, favoritesList);
        recyclerFavs.setLayoutManager(new LinearLayoutManager(this));
        recyclerFavs.setAdapter(adapter);

        firestore = FirebaseFirestore.getInstance();
        uid = FirebaseAuth.getInstance().getUid();

        loadFavorites();
    }

    private void loadFavorites() {
        if (uid == null) {
            Toast.makeText(this, "Please log in first", Toast.LENGTH_SHORT).show();
            finish();
            return;
        }

        progressBar.setVisibility(View.VISIBLE);
        tvEmpty.setVisibility(View.GONE);
```

```
        firestore.collection("users")
                .document(uid)
                .collection("favorites")
                .orderBy("savedAt", Query.Direction.DESCENDING)
                .get()
                .addOnSuccessListener(snapshots -> {
                    progressBar.setVisibility(View.GONE);
                    favoritesList.clear();
                    favoritesList.addAll(snapshots.toObjects(FavoriteRestaurant.class));
                    adapter.notifyDataSetChanged();
                    toggleEmptyState();
                })
                .addOnFailureListener(e -> {
                    progressBar.setVisibility(View.GONE);
                    Toast.makeText(this, "Failed to load: " + e.getMessage(),
Toast.LENGTH_LONG).show();
                    toggleEmptyState();
                });
    }

    /** Called by adapter after removing an item so the empty state stays accurate. */
    public void toggleEmptyState() {
        if (favoritesList.isEmpty()) {
            tvEmpty.setVisibility(View.VISIBLE);
            recyclerFavs.setVisibility(View.GONE);
        } else {
            tvEmpty.setVisibility(View.GONE);
            recyclerFavs.setVisibility(View.VISIBLE);
        }
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        if (item.getItemId() == android.R.id.home) {
            finish();
            return true;
        }
        return super.onOptionsItemSelected(item);
    }
}
```

FavoritesAdapter.java

```
package com.example.makanbot;

import android.content.Context;
import android.content.Intent;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageButton;
```

```
import android.widget.ImageView;
import android.widget.TextView;
import android.widget.Toast;

import androidx.annotation.NonNull;
import androidx.recyclerview.widget.RecyclerView;

import com.bumptech.glide.Glide;
import com.google.firebase.auth.FirebaseAuth;
import com.google.firebase.firestore.FirebaseFirestore;

import java.util.List;

public class FavoritesAdapter extends
RecyclerView.Adapter<FavoritesAdapter.FavViewHolder> {

    private final Context context;
    private final List<FavoriteRestaurant> favorites;

    public FavoritesAdapter(Context context, List<FavoriteRestaurant> favorites) {
        this.context = context;
        this.favorites = favorites;
    }

    @NonNull
    @Override
    public FavViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType)
{
        View view = LayoutInflater.from(context).inflate(R.layout.item_favorite, parent, false);
        return new FavViewHolder(view);
    }

    @Override
    public void onBindViewHolder(@NonNull FavViewHolder holder, int position) {
        FavoriteRestaurant fav = favorites.get(position);

        holder.tvName.setText(fav.getName());
        holder.tvAddress.setText(fav.getAddress() != null ? fav.getAddress() : "");
        holder.tvRating.setText("⭐ " + (fav.getRating() != null ? fav.getRating() : "N/A"));
        holder.tvHalal.setText("Halal: " + (fav.getHalal() != null ? fav.getHalal() : "N/A"));

        if (fav.getImageUrl() != null && !fav.getImageUrl().isEmpty()) {
            Glide.with(context)
                    .load(fav.getImageUrl())
                    .placeholder(android.R.drawable.ic_menu_gallery)
                    .into(holder.ivThumb);
        } else {
            holder.ivThumb.setImageResource(android.R.drawable.ic_menu_gallery);
        }
```

```
        // Tap card -> open RestaurantDetailActivity with full data
        holder.itemView.setOnClickListener(v -> {
            Intent intent = new Intent(context, RestaurantDetailActivity.class);
            intent.putExtra("name", fav.getName());
            intent.putExtra("address", fav.getAddress());
            intent.putExtra("halal", fav.getHalal());
            intent.putExtra("rating", fav.getRating());
            intent.putExtra("evidence", fav.getEvidence());
            intent.putExtra("image_url", fav.getImageUrl());
            intent.putExtra("loc_url", fav.getLocationUrl());
            intent.putExtra("res_url", fav.getReservationUrl());
            intent.putExtra("menu_url", fav.getMenuUrl());
            context.startActivity(intent);
        });

        // Remove from favourites
        holder.btnRemove.setOnClickListener(v -> {
            String uid = FirebaseAuth.getInstance().getUid();
            if (uid == null || fav.getName() == null) return;

            FirebaseFirestore.getInstance()
                    .collection("users").document(uid)
                    .collection("favorites").document(fav.getName())
                    .delete()
                    .addOnSuccessListener(aVoid -> {
                        int pos = holder.getAdapterPosition();
                        if (pos != RecyclerView.NO_POSITION) {
                            favorites.remove(pos);
                            notifyItemRemoved(pos);
                            Toast.makeText(context, "Removed from favorites",
Toast.LENGTH_SHORT).show();
                            if (context instanceof FavoritesActivity) {
                                ((FavoritesActivity) context).toggleEmptyState();
                            }
                        }
                    })
                    .addOnFailureListener(e ->
                            Toast.makeText(context, "Failed to remove: " + e.getMessage(),
Toast.LENGTH_SHORT).show());
        });
    }

    @Override
    public int getItemCount() {
        return favorites.size();
    }

    static class FavViewHolder extends RecyclerView.ViewHolder {
```

```
        ImageView ivThumb;
        TextView tvName, tvAddress, tvRating, tvHalal;
        ImageButton btnRemove;

        FavViewHolder(@NonNull View itemView) {
            super(itemView);
            ivThumb = itemView.findViewById(R.id.ivFavThumb);
            tvName = itemView.findViewById(R.id.tvFavName);
            tvAddress = itemView.findViewById(R.id.tvFavAddress);
            tvRating = itemView.findViewById(R.id.tvFavRating);
            tvHalal = itemView.findViewById(R.id.tvFavHalal);
            btnRemove = itemView.findViewById(R.id.btnFavRemove);
        }
    }
}
```

FavoriteRestaurant.java

```
package com.example.makanbot;

/**
 * Data model for a saved favourite restaurant.
 * Mirrors the fields passed via Intent from MainActivity -> RestaurantDetailActivity,
 * plus a timestamp for sorting.
 *
 * Has a no-arg constructor so Firestore can auto-deserialize it.
 */
public class FavoriteRestaurant {

    private String name;
    private String address;
    private String halal;
    private String rating;
    private String evidence;
    private String imageUrl;
    private String locationUrl;
    private String reservationUrl;
    private String menuUrl;
    private long savedAt;

    // Required empty constructor for Firestore
    public FavoriteRestaurant() {}

    public FavoriteRestaurant(String name, String address, String halal, String rating,
                              String evidence, String imageUrl, String locationUrl,
                              String reservationUrl, String menuUrl, long savedAt) {
        this.name = name;
        this.address = address;
        this.halal = halal;
        this.rating = rating;
```

```
        this.evidence = evidence;
        this.imageUrl = imageUrl;
        this.locationUrl = locationUrl;
        this.reservationUrl = reservationUrl;
        this.menuUrl = menuUrl;
        this.savedAt = savedAt;
    }

    // Getters (Firestore uses these)
    public String getName() { return name; }
    public String getAddress() { return address; }
    public String getHalal() { return halal; }
    public String getRating() { return rating; }
    public String getEvidence() { return evidence; }
    public String getImageUrl() { return imageUrl; }
    public String getLocationUrl() { return locationUrl; }
    public String getReservationUrl() { return reservationUrl; }
    public String getMenuUrl() { return menuUrl; }
    public long getSavedAt() { return savedAt; }

    // Setters (Firestore uses these when reading back)
    public void setName(String name) { this.name = name; }
    public void setAddress(String address) { this.address = address; }
    public void setHalal(String halal) { this.halal = halal; }
    public void setRating(String rating) { this.rating = rating; }
    public void setEvidence(String evidence) { this.evidence = evidence; }
    public void setImageUrl(String imageUrl) { this.imageUrl = imageUrl; }
    public void setLocationUrl(String locationUrl) { this.locationUrl = locationUrl; }
    public void setReservationUrl(String reservationUrl) { this.reservationUrl =
reservationUrl; }
    public void setMenuUrl(String menuUrl) { this.menuUrl = menuUrl; }
    public void setSavedAt(long savedAt) { this.savedAt = savedAt; }
}
```

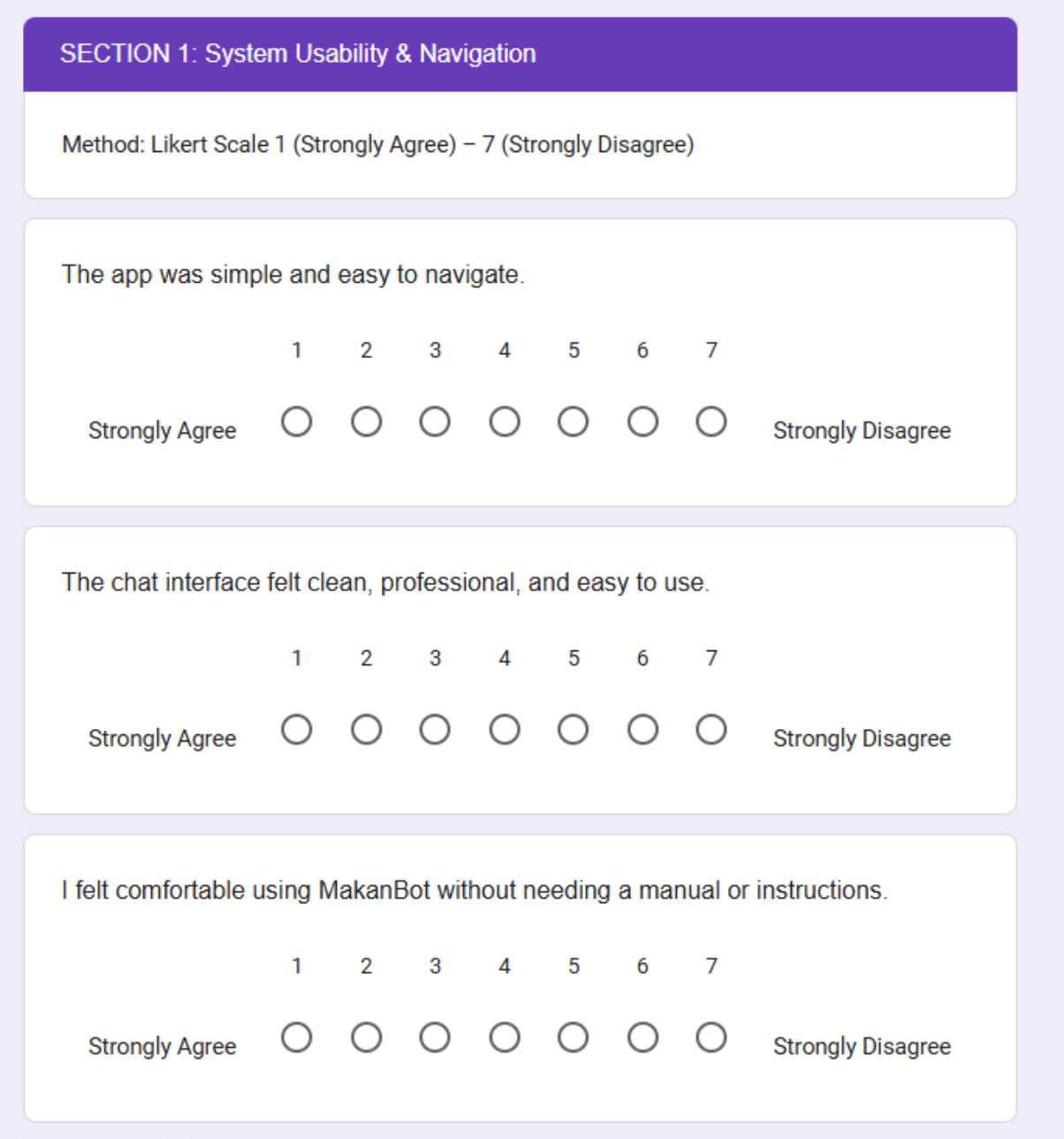
ProfileActivity.java

```
package com.example.makanbot;

import android.content.Intent;
import android.os.Bundle;
import android.widget.Button;
import android.widget.ImageView;
import android.widget.TextView;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;
import com.bumptech.glide.Glide;
import com.google.firebase.auth.FirebaseAuth;
import com.google.firebase.auth.FirebaseUser;
import com.google.firebase.firestore.FirebaseFirestore;
```

```
import java.util.List;

public class ProfileActivity extends AppCompatActivity {

    private FirebaseAuth mAuth;
    private FirebaseFirestore db;
    private TextView tvPreferences;
    private TextView tvName;
    private FirebaseUser user;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_profile);

        mAuth = FirebaseAuth.getInstance();
        db = FirebaseFirestore.getInstance();
        user = mAuth.getCurrentUser();

        ImageView ivProfile = findViewById(R.id.ivProfilePic);
        TextView tvEmail = findViewById(R.id.tvUserEmail);
        tvName = findViewById(R.id.tvUserName);
        tvPreferences = findViewById(R.id.tvPreferences);
        Button btnUpdate = findViewById(R.id.btnUpdateProfile);
        Button btnLogout = findViewById(R.id.btnLogout);

        if (user == null) {
            startActivity(new Intent(this, LoginActivity.class));
            finish();
            return;
        }

        tvEmail.setText(user.getEmail());

        if (user.getPhotoUrl() != null) {
            Glide.with(this).load(user.getPhotoUrl()).circleCrop().into(ivProfile);
        }

        // Launch PreferencesActivity in EDIT mode
        btnUpdate.setOnClickListener(v -> {
            Intent intent = new Intent(ProfileActivity.this, PreferencesActivity.class);
            intent.putExtra("EDIT_MODE", true);   // flag to tell PreferencesActivity we're
editing
            intent.putExtra("USER_ID", user.getUid());
            startActivity(intent);
        });

        btnLogout.setOnClickListener(v -> {
```

```
            mAuth.signOut();
            Intent intent = new Intent(ProfileActivity.this, LoginActivity.class);
            intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
Intent.FLAG_ACTIVITY_CLEAR_TASK);
            startActivity(intent);
            finish();
        });
    }

    // Reload preferences every time the user returns from PreferencesActivity
    @Override
    protected void onResume() {
        super.onResume();
        loadUserData();
    }

    private void loadUserData() {
        if (user == null) return;

        db.collection("users").document(user.getUid())
                .get()
                .addOnSuccessListener(documentSnapshot -> {
                    if (documentSnapshot.exists()) {
                        String name = documentSnapshot.getString("name");
                        if (name != null) tvName.setText(name);

                        List<String> prefsList = (List<String>) documentSnapshot.get("preference");
                        if (prefsList != null && !prefsList.isEmpty()) {
                            tvPreferences.setText(String.join(", ", prefsList));
                        } else {
                            tvPreferences.setText("No preferences set yet");
                        }
                    }
                })
                .addOnFailureListener(e ->
                        Toast.makeText(this, "Error loading data", Toast.LENGTH_SHORT).show());
    }
}
```

Appendix C: Google Form Survey



SECTION 2: Recommendation Quality & Trust

Method: Likert Scale 1 (Strongly Agree) – 7 (Strongly Disagree)

The restaurants recommended by MakanBot were relevant to my search criteria.

	1	2	3	4	5	6	7	
Strongly Agree	○	○	○	○	○	○	○	Strongly Disagree

The system provided enough information (e.g., location, menu, rating, halal status) to help me make a decision.

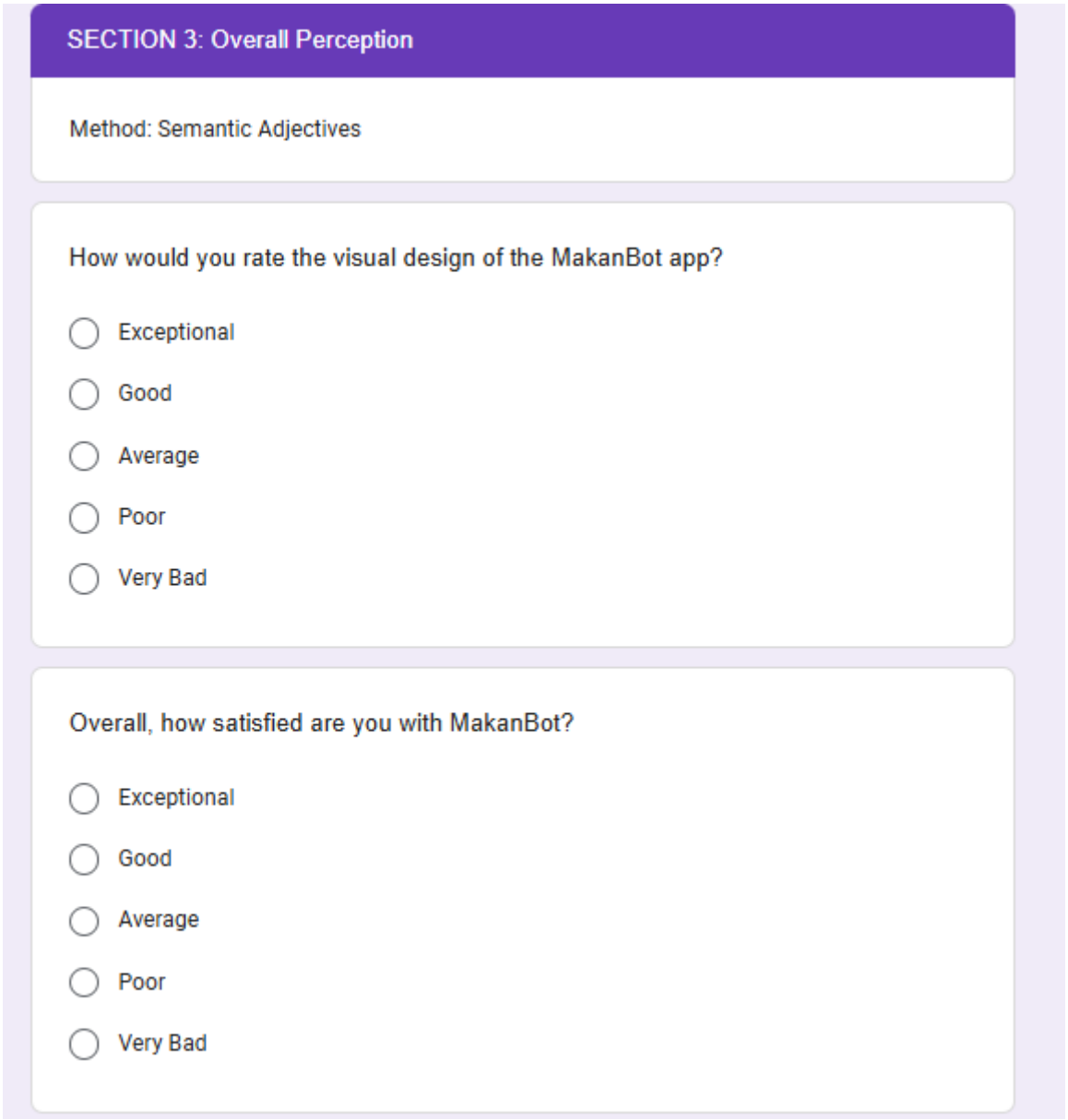
	1	2	3	4	5	6	7	
Strongly Agree	○	○	○	○	○	○	○	Strongly Disagree

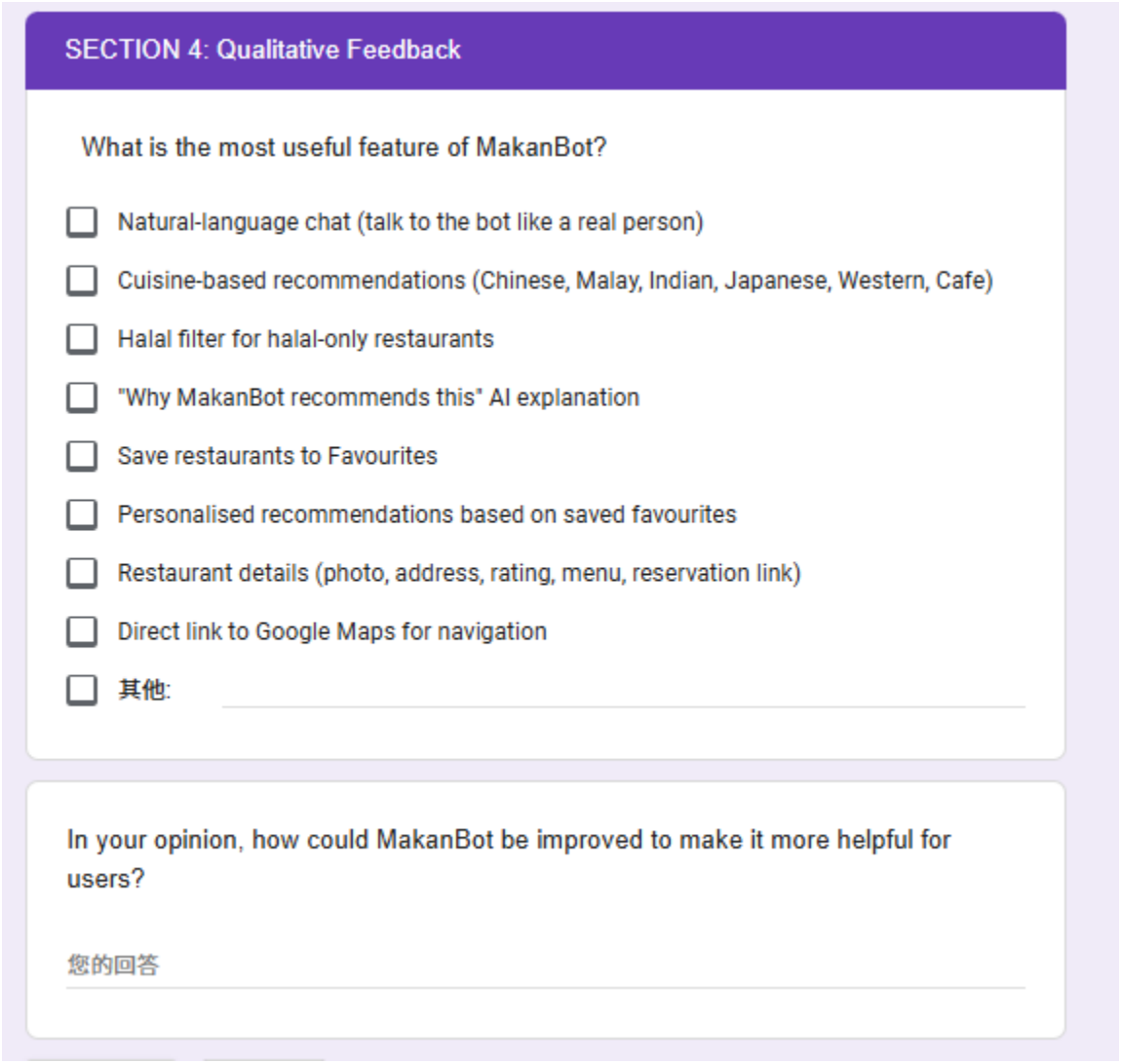
I would follow the recommendations provided by MakanBot in a real-life situation.

	1	2	3	4	5	6	7	
Strongly Agree	○	○	○	○	○	○	○	Strongly Disagree

The "Why MakanBot recommends this" explanation was helpful in building my trust in the recommendation.

	1	2	3	4	5	6	7	
Strongly Agree	○	○	○	○	○	○	○	Strongly Disagree


SECTION 3: Overall Perception
Method: Semantic Adjectives
How would you rate the visual design of the MakanBot app?
Exceptional
Good
Average
Poor
Very Bad
Overall, how satisfied are you with MakanBot?
Exceptional
Good
Average
Poor
Very Bad



SECTION 4: Qualitative Feedback

What is the most useful feature of MakanBot?

- [ ] Natural-language chat (talk to the bot like a real person)
- [ ] Cuisine-based recommendations (Chinese, Malay, Indian, Japanese, Western, Cafe)
- [ ] Halal filter for halal-only restaurants
- [ ] "Why MakanBot recommends this" AI explanation
- [ ] Save restaurants to Favourites
- [ ] Personalised recommendations based on saved favourites
- [ ] Restaurant details (photo, address, rating, menu, reservation link)
- [ ] Direct link to Google Maps for navigation
- [ ] 其他:

In your opinion, how could MakanBot be improved to make it more helpful for users?

您的回答

Appendix D:

Paper Submission

You have 1 paper submitted

New Submission

1. Paper AiDAS2026: 028-024

Conference:	The 7th International Conference on Artificial Intelligence and Data Sciences
Title:	MakanBot: A Mobile Chatbot for Explainable and Personalised Restaurant Recommendation Using Hybrid Machine Learning and Large Language Models
First Author:	Lew Yun Cheng
Affiliation:	Universiti Tunku Abdul Rahman
Co-Author(s):	Muhammad Husaini Nadri (husaini@utar.edu.my); Norliana Muslim* (norliana@unikl.edu.my); Tseu Kwan Lee (tseukl@utar.edu.my); Rohaya Abu Hassan (rohaya@unikl.edu.my); Lyana Izzati Mohd Asri (lyana@utar.edu.my)
Presenter(s):	Lew Yun Cheng
Sub-theme:	Artificial Intelligence Tracks: Machine Learning
File Name:	AIDAS2026_Paper Submission.pdf
File Type:	Adobe Acrobat Document
Total Pages:	6
Paper Status:	Pending Review

Edit & Upload Full Paper | Drop

MakanBot: A Mobile Chatbot for Explainable and Personalised Restaurant Recommendation Using Hybrid Machine Learning and Large Language Models

Lew Yun Cheng
Department of Computer Science
Faculty of Information and Communication Technology,
Universiti Tunku Abdul Rahman
Perak, Malaysia
yuncheng170904@1utar.edu.my

Muhammad Husaini Nadri
Department of Computer Science
Faculty of Information and Communication Technology,
Universiti Tunku Abdul Rahman
Perak, Malaysia
husaini@utar.edu.my

Norliana Muslim*
Artificial Intelligence Section
Malaysian Institute of Information Technology
Universiti Kuala Lumpur
Kuala Lumpur, Malaysia
norliana@unikl.edu.my

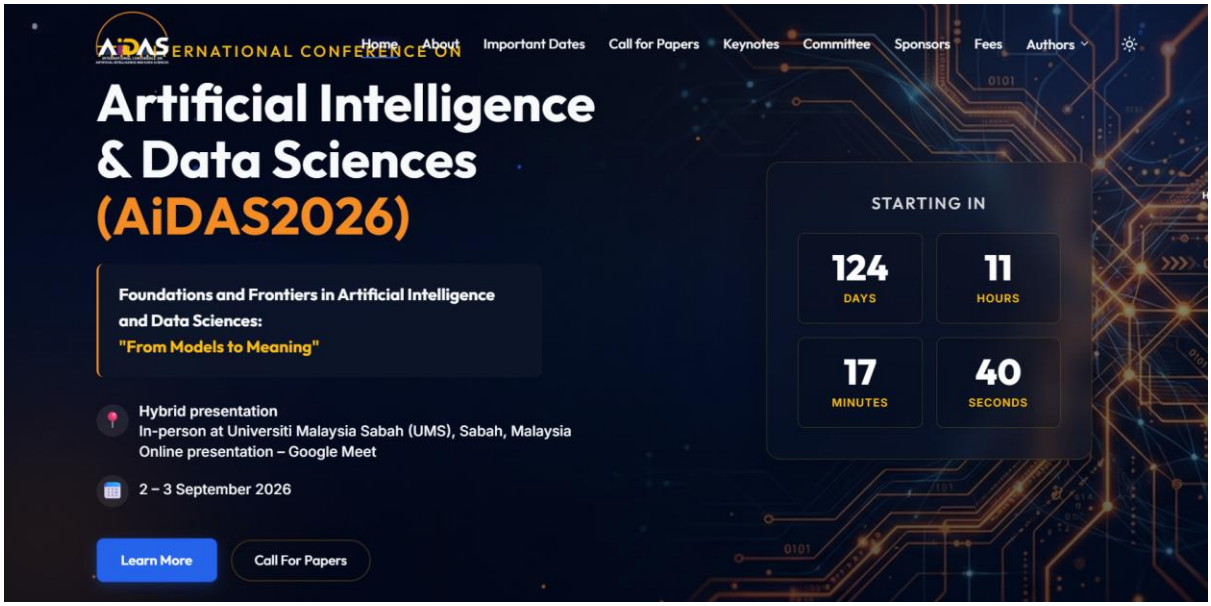
Tseu Kwan Lee
Department of Computer Science
Faculty of Information and Communication Technology,
Universiti Tunku Abdul Rahman
Perak, Malaysia
tseukl@utar.edu.my

Rohaya Abu Hassan
Software Engineering Section
Malaysian Institute of Information Technology
Universiti Kuala Lumpur
Kuala Lumpur, Malaysia
rohaya@unikl.edu.my

Lyana Izzati Mohd Asri
Department of Computing
Lee Kong Chian Faculty of Engineering and Science,
Universiti Tunku Abdul Rahman
Selangor, Malaysia
lyana@utar.edu.my

***Abstract*—Diners in Kuala Lumpur often face decision fatigue when choosing where to eat, while existing restaurant discovery platforms rely largely on keyword search and rarely consider individual taste history, halal certification, or dish level intent. This paper presents a mobile chatbot that delivers personalised, halal sensitive, and explainable restaurant recommendations through natural language interaction. The system was trained on a public dataset of 4,439 Kuala Lumpur restaurants and approximately 78,000 reviews, and integrates a content-based**

face decision fatigue when choosing where to eat. Mainstream discovery platforms such as map applications and food delivery services rely heavily on keyword matching, popularity sorting, and static category filters [1], [2]. These approaches rarely capture an individual user's taste history, dietary requirements, or the underlying intent behind dish level queries such as "chicken chop" or "ramen." The Malaysian context adds a further constraint: a large share of diners requires halal compliant options, and reviews of Kuala Lumpur restaurants confirm that halal certification is a recurring consumer concern [3].


ERNATIONAL CONFERENCE ON
Home
About
Important Dates
Call for Papers
Keynotes
Committee
Sponsors
Fees
Authors
Artificial Intelligence
& Data Sciences
(AiDAS2026)
Foundations and Frontiers in Artificial Intelligence
and Data Sciences:
"From Models to Meaning"
Hybrid presentation
In-person at Universiti Malaysia Sabah (UMS), Sabah, Malaysia
Online presentation – Google Meet
2 – 3 September 2026
Learn More
Call For Papers
STARTING IN
124
DAYS
11
HOURS
17
MINUTES
40
SECONDS

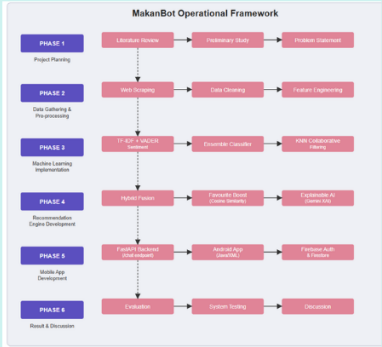
POSTER

Faculty of Information Communication and Technology
Lew Yun Cheng
Supervisor: Dr. Muhammad Husaini Bin Nadri

Development of A Mobile Chatbot for Restaurant Recommendation

INTRODUCTION

Kuala Lumpur is home to thousands of restaurants spanning a wide variety of cuisines, yet diners frequently face decision fatigue when choosing where to eat. Existing platforms rarely account for individual taste history, halal certification, or dish level queries such as "chicken chop" or "ramen". MakanBot is proposed as a mobile chatbot that delivers personalised, halal aware, and explainable restaurant recommendations through natural language interaction.



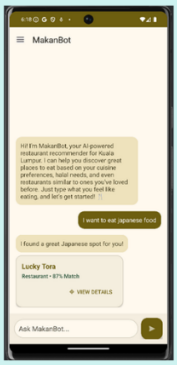
METHOD

The MakanBot system follows a six phase development workflow covering planning, data preprocessing, machine learning, recommendation engine integration, mobile application development, and evaluation. The hybrid recommendation engine fuses three signals through adaptive weighting: a content based ensemble classifier (SVM + Random Forest with soft voting), an item based K Nearest Neighbours collaborative filter, and a favourite based cosine similarity boost. The Google Gemini API powers natural language understanding and generates explainable AI sentences, while the Google Places API enriches each recommendation with live photos, addresses, and reservation links.

RESULT

Machine Learning Model	Accuracy	Precision	Recall	F1-Score
SVM (linear kernel, C=1)	0.8721	0.9611	0.8782	0.9178
Random Forest (100 estimators)	0.8860	0.9434	0.9147	0.9288
Ensemble (SVM + RF, Soft Voting)	0.9063	0.9300	0.9568	0.9432

Metric	Value
Best k (neighbourhood size)	40
Best similarity metric	pearson_baseline
Matrix orientation	Item-based (user_based = False)
Best RMSE (3-fold GridSearchCV)	1.0861
Test Set RMSE	1.0822
Test Set MAE	0.8210



DISCUSSION AND CONCLUSION

The MakanBot system delivers strong results across all evaluation methods. The ensemble classifier achieved an F1 score of 0.9432 and an ROC AUC of 0.94, while functional testing recorded a 100% pass rate and a pilot user study reported 90% favourable satisfaction. These outcomes confirm that hybrid recommendation, large language models, and halal aware filtering can be successfully integrated into a mobile dining assistant for Malaysian users. Future enhancements include dataset expansion, multilingual support, cloud deployment, and broader user testing.