Final Year Project Report

A report submitted in the partial fulfillment of degree of BSE

FuelUp: Personalized Nutrition, Powered by you

Bahria University, Islamabad

May 2026

Supervisor

Engr. Subas Bilal

Group Members

Omer Sultan **01-131212-043**

Software Engineering Department

FuelUp: Personalized Nutrition, Powered by you

Group Members

Omer Sultan (01-131212-043)

Supervisor: Eng. Subas Bilal

A Final Year Project submitted to the Department of Software Engineering, Faculty of Engineering Sciences, Bahria University, Islamabad in the partial fulfillment for the award of degree in Bachelor of Software Engineering

May 2026

FYP COMPLETION CERTIFICATE

Student Name:_____ Enrolment No:_____

Student Name:_____ Enrolment No:_____

Programme of Study: Bachelor of Software Engineering

Project Title: _____

It is to certify that the above students' project has been completed to my satisfaction and to my belief, its standard is appropriate for submission for evaluation. I have also conducted plagiarism test of this thesis using HEC prescribed software and found similarity index at _____ that is within the permissible limit set by the HEC. I have also found the thesis in a format recognized by the department.

Supervisor's Signature: _____

Date: _____ Name: _____

CERTIFICATE OF ORIGINALITY

This is certify that the intellectual contents of the project _____

_____

are the product of my/our own work except, as cited properly and accurately in the acknowledgements and references, the material taken from such sources as research journals, books, internet, etc. solely to support, elaborate, compare, extend and/or implement the earlier work. Further, this work has not been submitted by me/us previously for any degree, nor it shall be submitted by me/us in the future for obtaining any degree from this University, or any other university or institution. The incorrectness of this information, if proved at any stage, shall authorities the University to cancel my/our degree.

Name of the Student: _____

Signature: _____ Date: _____

Name of the Student: _____

Signature: _____ Date: _____

FuelUp: Personalized Nutrition, Powered by you

Sustainable Development Goals

SDG No	Description of SDG	SDG No	Description of SDG
SDG 1	No Poverty	SDG 9 ✓	Industry, Innovation, and Infrastructure ✓
SDG 2	Zero Hunger	SDG 10	Reduced Inequalities
SDG 3 ✓	Good Health and Well Being ✓	SDG 11	Sustainable Cities and Communities
SDG 4	Quality Education	SDG 12	Responsible Consumption and Production
SDG 5	Gender Equality	SDG 13	Climate Change
SDG 6	Clean Water and Sanitation	SDG 14	Life Below Water
SDG 7	Affordable and Clean Energy	SDG 15	Life on Land
SDG 8	Decent Work and Economic Growth	SDG 16	Peace, Justice and Strong Institutions
		SDG 17	Partnerships for the Goals

Range of Complex Problem Solving

Range of Complex Problem Solving			
	Attribute	**Complex Problem**	
1	Range of conflicting requirements	Involve wide-ranging or conflicting technical, engineering and other issues.	
2	Depth of analysis required	Have no obvious solution and require abstract thinking, originality in analysis to formulate suitable models.	
3	Depth of knowledge required	Requires research-based knowledge much of which is at, or informed by, the forefront of the professional discipline and which allows a fundamentals-based, first principles analytical approach.	✓
4	Familiarity of issues	Involve infrequently encountered issues	
5	Extent of applicable codes	Are outside problems encompassed by standards and codes of practice for professional engineering.	
6	Extent of stakeholder involvement and level of conflicting requirements	Involve diverse groups of stakeholders with widely varying needs.	✓
7	Consequences	Have significant consequences in a range of contexts.	
8	Interdependence	Are high level problems including many component parts or sub-problems	✓
Range of Complex Problem Activities			
	Attribute	**Complex Activities**	
1	Range of resources	Involve the use of diverse resources (and for this purpose, resources include people, money, equipment, materials, information and technologies).	
2	Level of interaction	Require resolution of significant problems arising from interactions between wide ranging and conflicting technical, engineering or other issues.	
3	Innovation	Involve creative use of engineering principles and research-based knowledge in novel ways.	✓
4	Consequences to society and the environment	Have significant consequences in a range of contexts, characterized by difficulty of prediction and mitigation.	✓
5	Familiarity	Can extend beyond previous experiences by applying principles-based approaches.	

Abstract

FuelUp is a Flutter mobile application I built for my final year project. The core idea is simple: recommend meals that actually fit the user — not just their dietary needs, but also how they feel at that moment. On the backend side, I used Firebase Authentication, Cloud Firestore, Realtime Database, Cloud Messaging, and Storage together. A custom Node.js service running on Render takes care of event-driven tasks like meal tagging.

The main thing that makes FuelUp different from regular food apps is the mood-aware recommendation engine. Rather than just showing the same meals to everyone, the system tries to detect the user's mood using an on-device speech emotion recognition model — a TFLite model that I integrated to extract MFCC features from the microphone. When that fails or gives a low-confidence result, a simpler tag-based classifier steps in as a fallback, so the app always has something to show.

Meal tagging uses two paths. The rule-based lexical engine always runs first, regardless of anything else. If the Gemini API quota is still available for that day, AI-assisted tagging also runs and adds richer tags on top. This way the system keeps functioning even when the external API hits its daily limit.

There are two roles in the app. Customers browse meals, complete their health profile, place orders, and track delivery in real time. Chefs list their meals, get FCM push notifications when orders come in, and update order status from their interface. Real-time updates go through Firebase Realtime Database listeners.

This report covers the motivation, background literature, requirements, design, implementation, and testing for FuelUp. I have also tried to be honest about the problems I encountered during the project — particularly around credential security, Firestore access control, listener reliability, and gaps in automated testing — and what should be done to fix them.

Keywords: Mood-Aware Recommendation, Flutter, Firebase, TensorFlow Lite, Speech Emotion Recognition, Nutrition Personalisation, Meal Ordering, Hybrid AI Architecture

Dedication

This project is dedicated first and foremost to Almighty God — the source of all knowledge, wisdom, and strength. Every step of this journey was made possible by His grace, and it is to Him that I owe the greatest gratitude.

To my parents — for their unconditional love, their sacrifices, and the quiet faith they have always shown in me. You never once asked me to be anything other than my best self, and that has been the greatest motivation of my life. This degree belongs to you as much as it belongs to me.

To my supervisor, Ma'am Subas Bilal — for seeing potential in this idea and guiding it with expertise and patience. To my teachers at Bahria University who shaped the way I think about engineering, problem-solving, and professional responsibility.

And to every friend who stayed up late, tested the app, asked the right questions, and kept spirits high when the deadline felt impossible — this one is for all of you.

Acknowledgements

I would like to express my heartfelt gratitude to my supervisor, Ma'am Subas Bilal, for her patient guidance, critical insight, and consistent support throughout every phase of this project. From the early design discussions to the final testing cycles, her feedback was always constructive and her encouragement never wavered. Working under her supervision has been one of the most rewarding experiences of my academic journey.

Ma'am Subas Bilal's dedication to academic rigour and her genuine interest in the technical direction of FuelUp pushed me to think more carefully about every architecture decision. The depth of thought she brought to each review session raised the quality of this project well beyond what I could have achieved working alone. I am grateful not just for the technical feedback, but for the way she challenged me to sharpen my problem-solving approach at every stage.

I also owe a lot to the open-source communities around Flutter, Firebase, TFLite, and Node.js. Good documentation and active forums made it possible to tackle a technical scope this wide within a final-year timeline. To my family — thank you for tolerating the late nights. And to the classmates who tested early builds and told me honestly what was broken: your feedback shaped the final product more than you probably realise.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

Food delivery apps have become a normal part of daily life, at least where smartphones are common. But if you actually look at how they work, most platforms treat every single user identically — same menu, same ranking, same promotions regardless of whether the person trying to lose weight or someone who just needs comfort food after a rough day. This bothered me enough to make it the starting point for FuelUp: what if a food app actually paid attention to the individual?

1.1 Motivation

Two things pushed me toward this particular project idea. First, I noticed that almost no food delivery platform connects a user's health information to what it recommends. If I am trying to cut calories because my BMI is in the overweight range, the app still shows me the same burger promotions as everyone else. There is just no link between the suggestions and what the person actually needs.

Second, by the time I started this project in 2024, the tools needed to actually build something like this had become accessible enough for a student project. TFLite lets you run ML inference on-device without a server. Firebase takes care of real-time sync and notifications without a lot of backend code. Gemini gives you a usable API for text understanding. The components were all there — they just needed to be put together properly.

FuelUp pulls those pieces together. It considers who the user is (their health profile and dietary restrictions), how they are feeling right now (mood detected through voice or set manually), and what their caloric target is based on BMI and goals — then uses all three to rank meal options. It is definitely not production-ready, but as a final-year project it shows the concept can actually work.

1.2 Objectives

My objectives for this project were: build a functional Flutter app covering both customer and chef roles; implement a mood detection system using on-device SER with a fallback option; create a recommendation engine that considers mood, calories, BMI, and weight goals; build an automated backend for meal tagging; cover the full order lifecycle from cart to delivery tracking; and test everything well enough to identify what would need fixing before any real deployment.

- To design and implement a cross-platform mobile application supporting distinct customer and chef operational roles with appropriate data isolation and navigation.
- To develop a mood detection subsystem combining on-device speech emotion recognition and a deterministic tag-based fallback, capable of producing a mood signal without dependency on cloud inference.
- To implement a multi-factor meal recommendation engine that weights mood compatibility, caloric alignment, BMI category context, and weight-goal direction in producing ranked meal suggestions.
- To build an automated meal tagging pipeline that enriches meal metadata using rule-based lexical analysis as a primary path and optional generative AI enhancement under quota governance.
- To implement a real-time order lifecycle system encompassing checkout, chef notification, status progression, and customer push notification delivery.
- To evaluate the system through functional testing and identify security, scalability, and quality risks for future resolution.

1.3 Problem Discussion

Apps like Uber Eats and Deliveroo do delivery very well, but they all have the same blind spot: every user is treated identically. A 20-year-old athlete and a 55-year-old managing diabetes both see the same promoted meals. There is nothing connecting what the platform shows to what the person actually needs.

On top of the health issue, no mainstream food delivery app I have seen takes the user's mood into account at all. Research on food psychology has long established that people's food preferences shift depending on how they feel — stress tends to pull people toward comfort food, for example, while a positive mood often leads to lighter choices. Ignoring that means missing a real personalisation opportunity.

The problem is not that this kind of personalisation is technically impossible — it clearly is not. The gap is that nobody has brought these specific capabilities together in one food ordering app. TFLite handles on-device ML, Firebase handles the real-time cloud layer, and Gemini handles AI-assisted text enrichment. All three are mature enough to build with. FuelUp was my attempt to show these can work as a combined system.

1.4 Main Contribution

The project makes the following concrete contributions:

A WebGL Flutter mobile application supporting distinct customer and chef roles with mood-aware, nutrition-conscious meal ordering. An on-device speech emotion recognition (SER) pipeline that infers user mood from microphone input without sending audio data to any server. A hybrid meal tagging backend that combines rule-based lexical analysis with Gemini AI enrichment, governed by a daily quota guard for reliability. A multi-factor meal recommendation engine weighting mood, calories, BMI, and goal direction with dynamically adjusted scoring weights. A documented analysis of production risks — credential management, access control, listener reliability, and schema consistency — with concrete mitigation proposals.

1.4.1 Existing Systems

The current generation of food delivery platforms addresses logistics well but personalisation poorly. Uber Eats and Deliveroo offer broad restaurant catalogues and reliable delivery tracking but do not factor in the user's nutritional profile or current mood. MyFitnessPal and similar apps handle caloric tracking but offer no ordering capability. Mealime and Eat This Much provide meal planning based on dietary preferences but without real-time mood input

or a live marketplace. None of the existing systems combine food ordering, health-aware filtering, and mood-adaptive recommendation in one platform.

1.4.2 Proposed System

FuelUp addresses these limitations through four integrated capabilities:

Mood-Aware Recommendation: The app infers the customer's current mood via on-device SER or a deterministic tag-based fallback, then uses the mood signal to rank meals alongside nutritional and goal-alignment factors.

Health-Conscious Meal Filtering: Customer profiles store BMI, TDEE, allergens, and dietary restrictions. Meals conflicting with any declared allergen are excluded; remaining meals are scored against caloric targets derived from the customer's profile.

Hybrid Meal Tagging: A rule-based lexical engine tags every meal unconditionally. An optional Gemini AI path adds richer semantic tags when quota allows, ensuring the system degrades gracefully rather than failing outright.

Real-Time Marketplace: Customers browse, order, and track delivery in real time. Chefs manage listings and receive push notifications for new orders, all through a shared Firebase infrastructure.

1.5 Key Technical Contributions

Within the scope of a final-year undergraduate project, FuelUp produces several concrete outcomes. It demonstrates a working mood-aware recommendation pipeline combining TFLite on-device inference with deterministic scoring. It implements a hybrid tagging backend that keeps functioning even when the Gemini API quota runs out. It provides a complete marketplace covering a Flutter client, Firebase data layer, and a Render-hosted Node.js backend. And it documents the security and reliability risks honestly rather than glossing over them.

- A working mood-aware recommendation system that integrates on-device ML inference (TensorFlow Lite SER model) with deterministic tag-based mood classification, demonstrating a practical hybrid architecture for constrained mobile environments.

- A hybrid meal tagging pipeline that combines rule-based lexical heuristics with quota-governed Gemini AI calls, ensuring tagging reliability under quota and connectivity constraints.
- A full-stack marketplace implementation spanning Flutter client, Firebase data services, and a Render-hosted Node.js backend with callable-compatible HTTP endpoints and long-running event listeners.
- A documented analysis of production risks in prototype-grade systems, covering credential management, access control policy design, listener process reliability, and test coverage gaps.

1.6 Report Organisation

Chapter 2 covers the background literature — recommender systems, speech emotion recognition, hybrid architectures, and mobile health informatics. Chapter 3 sets out the requirements. Chapter 4 covers system design. Chapter 5 goes through implementation. Chapter 6 covers testing and evaluation. Chapter 7 wraps up with reflections and what I would do differently or next.

Chapter 2: Background Study / Literature Review

This chapter goes over the background that shaped how I approached the design and build of FuelUp. The four areas I found most relevant were personalised recommendation systems, speech emotion recognition, hybrid AI architectures, and mobile health informatics.

2.1 Personalised Recommender Systems

Recommender systems have been an active research area since the early 2000s. Collaborative filtering (recommending based on what similar users liked) and content-based filtering (matching item features to user preferences) are the two main techniques, and most practical systems combine them because neither is reliable on its own.

For food in particular, content-based filtering makes more sense as the primary approach. Dietary restrictions and allergies are deeply personal and do not transfer from one person to another the way movie preferences might. The fact that someone with similar taste history liked a creamy curry is not useful information if you have a dairy allergy.

FuelUp does not use a trained ML model for recommendations. I chose a weighted scoring heuristic instead, combining mood compatibility, calorie alignment, BMI category, and goal direction. This was a practical decision — training a proper collaborative filtering model requires real user interaction data, which a prototype simply does not have. A carefully designed heuristic is actually more reliable in this scenario.

2.2 Speech Emotion Recognition

Speech emotion recognition (SER) is about inferring someone's emotional state from the acoustic properties of their speech, not from the words themselves. The most widely used features for this are MFCCs — Mel Frequency Cepstral Coefficients — which capture timbral characteristics of speech that change with emotional state.

MFCCs essentially capture how the vocal tract shapes sound. Features like spectral centroid, signal energy, and zero-crossing rate shift measurably between different emotional states — happy speech sounds acoustically

different from stressed or neutral speech — which is what makes MFCC-based models viable for emotion detection even without any speech-to-text step.

For running ML models on mobile, TFLite is the standard choice. Its support for quantised models — which sacrifice a small amount of accuracy to get much smaller file sizes and faster inference times — is particularly important when you are working with a phone's battery and CPU constraints.

In FuelUp, I used a four-class SER model deployed on-device through TFLite. The four classes are neutral, happy, surprise, and unpleasant. MFCC extraction runs entirely in Dart so no audio ever leaves the device. If the model confidence is too low or audio capture fails, the system automatically falls back to the tag-based mood classifier.

2.3 Hybrid Intelligent Architectures

One pattern that kept coming up in my literature reading was hybrid AI design — pairing a probabilistic ML component with a deterministic rule-based one. The rule-based part handles common cases in a predictable way, while the ML component deals with edge cases and adds semantic richness.

The reason this pattern is so useful is straightforward. ML models can produce unexpected failures and low-confidence outputs, but users need the system to do something reasonable at all times. A deterministic fallback provides that baseline guarantee even when the probabilistic component misbehaves.

FuelUp applies this pattern in two places. For meal tagging, the rule-based lexical analyser always runs first. For mood detection, tag-based scoring takes over when SER gives a low-confidence result. Both fallbacks turned out to be genuinely essential — the Gemini quota hit its limit several times during development, and the SER model was unreliable with certain microphone hardware.

2.4 Mobile Health Informatics

Mobile health apps are a well-established category, but most of them focus on calorie logging, meal planning, or exercise tracking. FuelUp occupies a slightly different niche — trying to combine actual meal ordering with health awareness and mood-based personalisation in one place.

The Mifflin-St Jeor equation is a standard method for estimating Basal Metabolic Rate (BMR) from weight, height, and age. Adding an activity multiplier gives the Total Daily Energy Expenditure (TDEE). FuelUp uses TDEE as the caloric target baseline for scoring meal options.

BMI is used in FuelUp as a simple, practical proxy for broad nutritional context — underweight, normal, overweight, obese. It is not a clinically precise measure, but it does not require clinical data either, and it gives the recommendation engine something meaningful to work with.

2.5 Existing Systems and Competitive Context

Looking at existing systems: Uber Eats and Deliveroo dominate commercial food delivery. Both have strong logistics and large catalogues, but their personalisation really goes no further than reordering previous items and filtering by category. Health data and mood are completely absent from both.

- Uber Eats and Deliveroo represent the commercial food delivery category. These systems offer rich meal catalogues and sophisticated logistics but provide no nutritional personalisation or health-aware recommendation.
- MyFitnessPal and Cronometer focus on nutritional logging and caloric tracking but do not integrate food ordering or affective context.
- Mealime and Eat This Much offer personalised meal planning based on dietary preferences but do not incorporate real-time mood signals or support a marketplace ordering model.

FuelUp is different from all of these in that it combines food ordering, nutritional personalisation, and mood-adaptive recommendation together. None of the existing systems I reviewed change their recommendations based on the user's current emotional state — FuelUp does.

2.6 Summary

The literature in this chapter directly informed the design choices I made. The recommendation scoring approach, the hybrid fallback architecture, the MFCC-based SER pipeline, and the BMI/TDEE caloric targeting all trace back to the foundations reviewed here.

Chapter 3: System Requirements

This chapter documents the requirements for FuelUp. I should be transparent that the project was largely build-first — requirements evolved as implementation progressed rather than being fully specified upfront. The list here reflects what was planned plus what I actually ended up building.

3.1 Use Cases

3.1.1 Customer Use Cases

Customer interactions are documented through ten use cases that follow the full journey from account registration through to tracking a delivered order.

- UC-C1: Register Account — Customer provides name, email, password, and health profile data. System creates a Firebase Authentication record and Firestore customer document.

Table 1:UC-C1-Register Account

Register Account	
Use Case ID	UC-C1
Use Case Name:	Register Account
Actor(s):	Customer
Pre-Conditions:	1. Application is launched. 2. User is not logged in.
Post-Conditions:	User account is created with status "Active". User is authenticated.
Priority:	High
Basic Flow:	1. User navigates to signup screen. 2. User enters name, Email, Password & phone no. 3. User selects role (Customer). 4. System validates input format. 5. System creates account in Firebase Auth. 6. System redirects user to Profile Setup.
Alternative Flows	**A1. Email already exists**-system shows "Account already exists" error. **A2. Weak Password**- System prompts user to use stronger password **A3. No Internet**- System prevents submission and shows offline error.

- UC-C2: Login — Customer authenticates using email and password. System loads role and profile context and redirects to the customer home interface.

Table 2:UC-C2-Login

Login	
Use Case ID	UC-C2
Use Case Name:	Customer Login.
Actor(s):	Customer
Pre-Conditions:	1. User must be registered in the system. 2. Application is connected to the internet.
Post-Conditions:	1. User is successfully authenticated. 2. User session token is generated & stored locally.
Priority:	High
Basic Flow:	1. User opens the login screen 2. User enters registered email & password 3. User clicks "login". 4. System verifies credentials with firebase authentication 5. System retrieves user role (customer) 6. System navigates to the appropriate home dashboard.
Alternative Flows	**A1. Invalid credentials** -system shows "Incorrect Email or password" **A2. Account Suspended**- System denies access & shows contact admin message **A3. Forgot Password**- User clicks forgot password to intitate recovery flow

- UC-C3: Browse Meals — Customer views meal categories and grouped meal listings with caloric information and allergen indicators.

Table 3:UC-C3-Browse Meal

Browse meals	
Use Case ID	UC-C3
Use Case Name:	Browse meals
Actor(s):	Customer
Pre-Conditions:	1. User must be logged into the system. 2. Application is connected to the internet.
Post-Conditions:	1. User is presented with a categorized list of meals 2. caloric and allergen data for each meal is successfully loaded
Priority:	High
Basic Flow:	1. User navigates to the menu or explore section 2. System retrieves meal categories and grouped listing from the database 3. System fetches specific meal data, including names, prices and caloric information 4. System cross references ingredients to identify and display allergen indicators (e.g., Nut-Free, Gluten-Free) 5. User views the list of meals grouped by their respective categories
Alternative Flows	**A1. No meals found**- system displays a "no meals available in this category" message **A2. Network Timeout**- System displays a "Failed to load menu" error and provides a retry button **A3. Filter by preference** - User applies a filter to only see meals matching specific health goals or allergens exclusions.

- UC-C4: Detect Mood — Customer activates voice recording via the mood button. System performs on-device SER inference and updates the active mood state.

Table 4:UC-C4-Detect Mood

Detect Mood	
Use Case ID	UC-C4
Use Case Name:	Detect mood
Actor(s):	Customer
Pre-Conditions:	1. User must be logged into the system. 2. App must have permission to access the device's microphone
Post-Conditions:	1. A **MoodAnalysis** record is created with the **detected_mood** and **pitch_frequency** 2. The app's state is updated to reflect the new mood, triggering specialized meal suggestions
Priority:	High (core AI feature)
Basic Flow:	1. User taps the Mood Analysis button on the home screen 2. System activates the microphone and begins voice recording 3. System captures audio data and passes it the **VoiceAnalysisService** 4. System performs on-device SER inference to analyse pitch and frequency 5. User determines the **detected_mood** (e.g.: Stressed, Happy, Neutral) 6. system saves the **pitch_frequency** and result in the database. 7. system updates the UI to show the current mood and suggests "Mood-Matching" meals
Alternative Flows	**A1. No audio input**- If the user does not speak, the system times out after 5 seconds and prompts the user to try again **A2. Permission Denied**- If microphone access is denied, the system displays a message explaining why the feature cannot function without it **A3. Indetermined Mood** – If background noise is too high, the system stes the mood to "Neutral" and suggests standard healthy options.

- UC-C5: Set Mood Manually — Customer selects a mood from a manual picker as an alternative to voice detection.

Table 5:UC-C5-Select Mood Manually

Set Mood Manually	
Use Case ID	UC-C5
Use Case Name:	Set mood manually
Actor(s):	Customer
Pre-Conditions:	1. User must be logged into the system. 2. Application is connected to the internet to allow for database synchronization
Post-Conditions:	1. A record is successfully created or updated in the **MoodAnalysis** table 2. The system triggers a refresh of the dashboard to show mood-matching meal recommendations.
Priority:	Medium (Alternative to AI detection)
Basic Flow:	1. User navigates to the "Mood Analysis" section or home dashboard 2. User selects the "Manual Selection" option as an alternative to voice trigger 3. System displays a manual picker (e.g.: Happy, stressed) 4. User selects their current mood and confirms 5. System saves the **detected_mood** and a **timestamp** to the database. 6. System updates the homepage suggesting specific meals categorized to improve or match the selected mood 7. system updates the UI to show the current mood and suggests "Mood-Matching" meals
Alternative Flows	**A1. User cancels Selection**- User closes the picker without choosing a mood, the systems retain the previous state **A2. Connection error**- System displays a "Failed to save mood" message if the record cannot be pushed to the MoodAnalysis table

- UC-C6: View Recommendations — System surfaces mood- and nutrition-filtered ranked meals based on customer profile and current mood state.

Table 6:UC-C6-View Recommendations

View Recommendations	
Use Case ID	UC-C6
Use Case Name:	View Recommendations
Actor(s):	Customer
Pre-Conditions:	1. User must be successfully authenticated 2. Customer profile (Including BMI, health goals and allergies) must be completed
Post-Conditions:	1. User is presented with a personalized, ranked list of meals 2. Any meals containing restricted allergens are automatically filtered out
Priority:	High
Basic Flow:	1. User opens the application 2. System retrieves the most recent health goals and mood if selected 3. System identifies all available meals from the Meal table 4. System performs a "safety filter" by cross-referencing CustomerProfile.allergies with Ingredient.allergen_type 5. System ranks the safe meals based on the current mood 6. System displays the filtered and ranked meal listings to the user
Alternative Flows	**A1. No mood data found**- If no mood has been set today, the system defaults to recommendations based solely on health goals and allergies **A2. Zero Matches**- If filters are too restrictive, no meals found

- UC-C7: Add to Cart — Customer adds meals to a local cart with quantity selection.

Table 7:UC-C7-Add to Cart

Add to Cart	
Use Case ID	UC-C7
Use Case Name:	Add to cart
Actor(s):	Customer
Pre-Conditions:	1. User must be successfully authenticated 2. User must be viewing the meal listings or a specific recoomendation
Post-Conditions:	1. The selected meal and its quantity are stored in the local application state 2. Cart is updated to reflect new total
Priority:	High
Basic Flow:	1. User identifies a meal from the menu or recommendation list 2. User selects the meal to open the quantity using the increment/decrement controls 3. User clicks the "Add to cart" button 4. System validates that the meal exists in the Meal table 5. System adds the meal id, name, price and selected quantity to the local cart object 6. System refresh the UI and update the cart icon counter 7. System identifies the kitchen id for each meal and sends a notification to the respective chef
Alternative Flows	**A1. Item already in the cart**- If the meal is already in the cart, the system adds the new quantity to the existing cout **A2. Inventory Check**- If the meal is marked as "Out of Stock" by the Chef, the "Add to cart" button is disabled

- UC-C8: Checkout — Customer provides delivery information and selects a payment method. System writes order to Realtime Database and notifies the chef.

Table 8:UC-C8-Checout

Checkout	
Use Case ID	UC-C8
Use Case Name:	Checkout
Actor(s):	Customer
Pre-Conditions:	1. User must be successfully authenticated 2. One or more items must be present in the user's active cart 3. Application must be connected to the internet for RealTime Database synchronization
Post-Conditions:	1. An order record is created with unique order id and pending status 2. A payment record is generated and linked 1-to-1 with the order 3. Junction records are created in the order meal table to track specific items
Priority:	High
Basic Flow:	1. User navigates to the checkout screen and reviews the meal summary 2. User provides delivery information, including name, address and ph no. 3. User selects payment method cash on delivery or card payment 4. User clicks confirm order 5. System calculates the total amount based on the sum of meal prices 6. System writes the order to real-time database, setting the order state as timestamp 7. System identifies the kitchen id for each meal and sends a notification to the respective chef
Alternative Flows	**A1. Invalid Delivery Info**- System displays a "Please enter a valid delivery address" error. **A2. Empty Cart**- System prevents the user from accessing the checkout screen if no items are selected. **A3. Payment rejected**- System notifies the user of the failed transaction; the Order record is updated with a "Cancelled" or "Payment Failed" status.

- UC-C9: Track Order — Customer views order history and real-time order status updates.

Table 9:UC-C9-Order Management

Order Management	
Use Case Name:	Track Order Status.
Actor(s):	Customer (User).
Pre-Conditions:	1. User is logged in. 2. User has placed at least one active order. 3. Order exist in the database with status "Pending" or "Processing".
Post-Conditions	User is informed of the real-time status of their meal delivery.
Priority:	High.
Basic Flow:	1.User navigates to the "My Orders" or "Track Order" screen. 2.System retrieves list of active orders from Firebase. 3.User taps on a specific order to view details. 4.System subscribes to real-time updates for that order ID. 5.System displays current status (e.g., "Preparing", "Out for Delivery") & estimated time. 6.User views Chef details & delivery progress.
Alternative Flows:	**A1: Order Cancelled by Chef**-System sends a push notification & updates status to "Cancelled". **A2: Network Error**- System displays cached status with a "Last updated: [Time]" label. **A3: No Active Orders**- System displays "You have no active orders" message.

- UC-C10: Manage Profile — Customer updates personal details, health metrics, dietary preferences, and allergy information.

Table 10:UC-C10-Manage Profile

Manage Health Profile (Update BMI & Allergies)	
Use Case Name:	Manage Health Profile.
Actor(s):	Customer (User).
Pre-Conditions:	1. User is logged in. 2. User is on the Profile Screen.
Post-Conditions	User profile is updated and AI recommendations are recalculated.
Priority:	High.
Basic Flow:	1.User selects "Edit Profile". 2.User updates metrics (Weight, Height, BMI) or allergies. 3.User clicks "Save changes". 4.System validates the new data. 5.System updates the User record in Firebase Realtime Database. 6.System refreshes AI meal recommendations based on new data.
Alternative Flows:	**A1: Invalid Data**- System warns if BMI values are unrealistic (e.g. negative numbers). **A2: Network Error**- System keeps old data and warns "Could not save changes".

3.1.2 Chef Use Cases

- UC-K1: Login — Chef authenticates and is redirected to the kitchen interface.

Table 11:UC-K1-Login

Login	
Use Case ID	UC-K1
Use Case Name:	Login
Actor(s):	Chef
Pre-Conditions:	1. User must be registered in the system with the "Chef" role. 2. Application is connected to the internet.
Post-Conditions:	1. User is successfully authenticated via Firebase. 2. User is redirected to the **Kitchen Interface/Dashboard**. 3. User session token is generated and stored locally.
Priority:	High
Basic Flow:	1. User opens the login screen. 2. User enters registered email and password. 3. User clicks "login". 4. System verifies credentials with **Firebase Authentication**. 5. System retrieves the user role from the User table and confirms it is "Chef". 6. System navigates the user to the specialized **Kitchen Interface** for order management.
Alternative Flows	**A1. Invalid Credentials** - System shows "Incorrect Email or password". **A2. Role Mismatch** - If a "Customer" attempts to login to the Chef app, the system denies access and redirects them to the Customer app. **A3. Forgot Password -** User clicks forgot password to initiate recovery flow.

- UC-K2: Add Meal — Chef provides meal name, category, description, ingredients, calories, price, and an optional image.

Table 12:UC-K2-Add Meal

Upload New Meal	
Use Case Name:	Upload New Meal.
Actor(s):	Home Chef.
Pre-Conditions:	1. Chef is logged in & verified. 2. Chef has prepared meal details (Photo, Ingredients, Price).
Post-Conditions	Meal is visible on the Home Screen for Customers to order.
Priority:	High.
Basic Flow:	1.Chef navigates to "My Kitchen" dashboard. 2.Chef clicks "Add Meals". 3.Chef uploads meal image & enters Title, Description & Price. 4.Chef enters Ingredients list (Critical for AI filtering). 5.Chef clicks "Publish Meal". 6.System stores image in Firebase Storage & meal data in Database.
Alternative Flows:	**A1: Missing Ingredients**- System blocks upload if ingredients list is empty (Safety Requirement). **A2: Image too Large**- System rejects file over 5MB.

- UC-K3: Edit / Delete Meal — Chef modifies or removes existing meal listings.

Table 13:UC-K3-Edit Meal

Edit/Delete Meal	
Use Case ID	UC-K3
Use Case Name:	Edit/Delete meal
Actor(s):	Chef
Pre-Conditions:	1. Chef must be authenticated. 2. The meal to be modified must exist in the Meal table and be linked to the Chef's kitchen_id.
Post-Conditions:	1. The Meal record is updated with new attributes (price, calories, etc.) or permanently removed. 2. Changes are reflected in the customer-side menu via the Realtime Database.
Priority:	High
Basic Flow:	1. Chef navigates to the "Manage Menu" section. 2. System retrieves all meals where kitchen_id matches the Chef's ID. 3. Chef selects a specific meal to edit or clicks a delete icon. 4. For **Edit**: Chef modifies fields like price or calories and clicks "Save". 5. For **Delete**: Chef confirms removal via an AlertDialog. 6. System updates the database and calls setState() to refresh the dashboard.
Alternative Flows	**A1. Delete Canceled** - Chef closes the AlertDialog without confirming; the meal remains in the list.

- UC-K4: View Orders — Chef monitors incoming orders and their associated meal details.

Table 14:UC-K4-View Orders

View Order	
Use Case ID	UC-K4
Use Case Name:	View Orders.
Actor(s):	Chef
Pre-Conditions:	1. Chef is logged in. 2. Customers have successfully placed orders containing meals from this kitchen.
Post-Conditions:	1. Chef is presented with a real-time list of current orders.
Priority:	High
Basic Flow:	1. Chef opens the "Incoming Orders" tab. 2. System queries the Order_Meal junction table to find orders associated with the Chef's kitchen_id. 3. System uses a ListView.builder to render each order as a card. 4. Chef views order details, including the order_date and total amount.
Alternative Flows	**A1. No Active Orders** - System displays an empty state message: "No new orders at the moment". **A2. Connection Lost** - System shows a "Reconnecting..." indicator until the stream is restored.

- UC-K5: Update Order Status — Chef transitions order status through defined states (pending, accepted, preparing, ready, delivered).

Table 15:UC-K5-Update Order Status

Update Order Status	
Use Case ID	UC-K5
Use Case Name:	Update Order status
Actor(s):	Chef
Pre-Conditions:	1. An order must exist in the Order table with an initial status of "Pending".
Post-Conditions:	1. The status attribute in the Order table is updated. 2. The Customer receives a real-time update on their tracking screen.
Priority:	High
Basic Flow:	1. Chef selects an active order from the list. 2. Chef selects the new status from a dropdown or toggle (e.g., "Accepted," "Preparing," "Ready," "Delivered"). 3. System updates the status string in the Realtime Database. 4. System triggers a setState() to move the order to the corresponding section of the Chef UI.
Alternative Flows	**A1. Accidental Status Change** - Chef can revert the status if it has not yet reached "Delivered".

- UC-K6: Receive Notifications — Chef receives push notifications for new customer orders.

Table 16:UC-K6-Recieve Notifications

Receive Notifications	
Use Case ID	UC-K6
Use Case Name:	Receive notifications
Actor(s):	Chef
Pre-Conditions:	1. Chef device has a registered push token. 2. A Customer has finalized a checkout.
Post-Conditions:	1. Chef is alerted to a new order via a system notification.
Priority:	High
Basic Flow:	1. System detects a new entry in the Order table for a specific kitchen_id. 2. Cloud Function/Realtime Database trigger initiates a push notification. 3. Chef's device displays a notification: "New Order Received!" 4. Chef taps the notification to be redirected immediately to the "View Orders" screen.
Alternative Flows	**A1. Background State** - Notification appears on the lock screen if the app is closed. **A2. DND Mode** - Notification is silent if the Chef has enabled "Do Not Disturb" on their device.

Note: Full use case description tables for each use case (actor, pre-conditions, basic flow, and alternative course of action) are maintained in the project documentation repository and are available on request. The use cases above follow the UC-ID / name / description / purpose format consistent with the system requirements captured in Chapter 3.

3.2 Functional Requirements

The functional requirements below cover all the observable behaviours in the system. I grouped them by feature area: authentication, meal browsing, mood detection, recommendation, cart and checkout, order management, chef meal management, and push notifications.

Table 17: Functional Requirements and Priorities

FR-ID	Requirement	Priority
FR-01	The system shall support email/password registration and login for both customer and chef roles.	High
FR-02	The system shall enforce email verification before granting access to authenticated features.	High

FR-ID	Requirement	Priority
FR-03	The system shall route customers and chefs to distinct navigation structures based on their role.	High
FR-04	The system shall detect user mood from a voice recording using on-device SER inference.	High
FR-05	The system shall support manual mood selection as an alternative to voice-based detection.	Medium
FR-06	The system shall rank meals using a composite score combining mood, caloric, BMI, and goal-direction factors.	High
FR-07	The system shall pre-filter meals that conflict with the customer's declared allergens or dietary preferences.	High
FR-08	Chefs shall be able to create, edit, and delete meal listings including optional image upload.	High
FR-09	The system shall automatically generate meal tags, allergen labels, and dietary labels upon meal creation or update.	High
FR-10	The customer's cart shall persist locally across sessions using device storage.	Medium
FR-11	Checkout shall write a structured order record to Firebase Realtime Database and invoke a backend notification endpoint.	High
FR-12	Chefs shall receive push notifications for new orders via Firebase Cloud Messaging.	High
FR-13	Customers shall receive push notifications when their order status changes.	High

FR-ID	Requirement	Priority
FR-14	The system shall calculate BMR, TDEE, and goal-adjusted caloric targets from the customer's health profile.	Medium
FR-15	The system shall maintain a BMI reading history for each customer.	Low

3.3 Non-Functional Requirements

3.3.1 Performance

- The mood detection pipeline shall complete on-device inference within three seconds of audio recording completion on a mid-range Android device.
- Meal listing screens shall load within two seconds under typical network conditions.

3.3.2 Reliability

- Core recommendation and browsing functionality shall remain operational when AI tagging or voice inference is unavailable, through the deterministic fallback paths.
- Order status updates shall be delivered to customers within thirty seconds of chef status transitions under normal network conditions.

3.3.3 Security

- User authentication credentials shall be managed exclusively through Firebase Authentication and never stored in plaintext on device.
- All backend mutation endpoints shall validate Firebase ID tokens before processing requests.

3.3.4 Usability

- The mood detection interface shall be operable by a user unfamiliar with the system within thirty seconds of first use.

- The checkout flow shall complete in no more than five user interactions from cart review to order confirmation.

3.3.5 Maintainability

- The codebase shall follow Flutter Provider state management conventions with clear separation between data services, state providers, and UI components.

3.3.6 Scalability

The Firebase backend should handle at least 500 concurrent users without degradation in recommendation or ordering response times.
The Node.js backend tagging service should process meal tagging jobs within 30 seconds of a new meal being published.

3.3.7 Availability

Core ordering and browsing functionality shall remain available during backend service restarts by relying on Firebase client-side caching.
The Render-hosted backend should maintain at least 95% uptime under normal load conditions.

3.4 Interface Requirements

The app needs a smartphone or tablet running Android 5.0 or later, or iOS 12 or later. A working microphone is needed for voice mood detection. Internet access is required for all Firebase and backend operations.

3.5 Database Requirements

Firebase must have Cloud Firestore enabled for user profiles and meal data, Realtime Database for order records and status updates, Authentication for identity management, Cloud Messaging for push notifications, and Storage for meal images.

3.6 Project Feasibility

3.6.1 Technical Feasibility

FuelUp is technically feasible within a final-year scope. Flutter and Firebase are both extensively documented with large community ecosystems. The TFLite SER model was sourced and adapted from existing open-source emotion recognition work, reducing the machine learning burden to feature extraction and integration rather than model training from scratch. The hardest individual component — on-device MFCC extraction in Dart — required working at a lower level than typical Flutter tutorials cover, but all necessary operations (FFT, mel filterbank, DCT) are implementable in pure Dart.

3.6.2 Operational Feasibility

The app is designed for users with no technical background. The mood detection interface uses a single large microphone button; the checkout flow completes in five interactions or fewer. Chef onboarding requires only a registration and meal listing form. The system's complexity is entirely behind the interface — from a user's perspective, FuelUp works like any other food ordering app, with the difference that recommendations adapt to their profile and mood.

3.6.3 Legal and Ethical Feasibility

Voice data is processed entirely on-device and never transmitted to any server. Firebase Authentication manages credentials; no passwords are stored in Firestore. Dietary and health profile data is stored under the authenticated user's Firestore document and is not shared with third parties. The Gemini API terms permit free-tier application development. All third-party packages used are under MIT or Apache 2.0 licences compatible with application development.

3.7 Conclusion

These requirements directly drove the design decisions in the following chapters. The reliability requirements — especially the need for fallback behaviour when AI components fail — were probably the biggest factor in pushing me toward the hybrid architecture.

Chapter 4: System Design

This chapter covers the system design of FuelUp — the overall architecture, the major component structure, how things interact, and what the data model looks like. The diagrams throughout are meant to give a clearer picture of both the static structure and the runtime behaviour.

4.1 Design Approach

The architecture follows a layered approach, separating the Flutter UI from the business logic (Provider services and domain modules), the Firebase data layer, and the backend processing tier. I was strict about not allowing the UI to make database calls directly — all data access goes through service classes.

The principle I kept coming back to throughout the design process was reliability through fallback. Every intelligent subsystem has a simpler, deterministic alternative that kicks in when the primary path fails. This was not just good practice on paper — during development it turned out to be genuinely necessary, especially given how often the Gemini API quota ran out.

4.2 Design Constraints

A number of practical constraints shaped the design from early on:

Mobile-Only Deployment: FuelUp targets Android and iOS through Flutter. Web or desktop deployment was not a design goal; this allowed us to use on-device TFLite inference and native microphone APIs without the restrictions of a browser environment.

Firebase as Primary Infrastructure: All user authentication, structured data storage, real-time order tracking, and push notifications run through Firebase. This decision bounds the system's availability and cost to Firebase's service limits.

On-Device SER Model: The speech emotion recognition model runs on the device rather than on a server. This protects user privacy but constrains model size and complexity to what TFLite can handle on a mid-range Android device.

Gemini API Quota: The AI-assisted meal tagging pathway is gated by a daily quota. This is a hard constraint on the free tier and drove the hybrid fallback architecture.

Single Backend Service: The Node.js backend is a single Render-hosted service. There is no load balancing or horizontal scaling at this stage, which limits concurrent order processing capacity.

No Real-Time Server Push to Backend: Firebase Realtime Database listeners handle client-side real-time updates. The backend uses Firestore listeners rather than WebSockets for event-driven processing.

4.3 System Architecture

There are four main layers in the system. The Flutter client is everything the user interacts with directly. Firebase provides the data backbone: Firestore for structured data, Realtime Database for orders, Authentication for login, Messaging for push notifications, and Storage for images. The Node.js Express backend running on Render handles asynchronous processing like meal tagging and chef notifications. The intelligence layer is the on-device TFLite SER model, the tag-based mood classifier, and the Gemini AI path for tagging.

- Presentation and State Layer: The Flutter mobile client, which implements all user interface components, manages application state through Provider, and handles routing via GoRouter. This layer communicates with Firebase services through the official Firebase Flutter SDK packages and with the Render backend through HTTP POST requests.
- Data and Identity Layer: Firebase services including Cloud Firestore for structured data, Realtime Database for transactional order state, Firebase Storage for media assets, Firebase Authentication for identity, and Firebase Cloud Messaging for push notification delivery.
- Backend Processing Layer: A Node.js Express application deployed on Render that listens to Firestore and Realtime Database events, processes meal tagging jobs through a queue and worker model, and exposes callable-compatible HTTP endpoints for chef notification and batch meal retagging.

- Intelligence Layer: On-device TensorFlow Lite SER model for voice mood inference, deterministic tag-based mood scoring for mood fallback, rule-based lexical heuristics for meal tagging as the primary path, and optional Gemini API calls for AI-enhanced meal tag generation.

4.3.1 Component Communication Patterns

The Flutter client communicates with Firebase through the official SDK, which handles token management, real-time streams, and offline caching automatically. The Node.js backend uses Firestore listeners rather than a direct connection from the client — it watches for document changes and processes them asynchronously. The only direct HTTP call from the client to the backend is for the chef notification endpoint.

4.4 Logical Design

On the client side, I separated the code into distinct logical modules to keep responsibilities clear and make testing more practical.

4.4.1 Authentication and Session Module

The authentication module wraps all Firebase Auth operations — registration, login, email verification, password reset, and logout. It also reads and stores the user role (customer or chef), which controls navigation after login.

4.4.2 Data Access Module

All Firestore and Realtime Database operations for users, customers, chefs, meals, categories, and recipes are centralised in the data access module. FCM token registration and refresh also live here. Having a single place for all database calls made it significantly easier to debug access issues.

4.4.3 Mood Intelligence Module

The mood intelligence module coordinates between the SER pipeline and the tag-based fallback. It decides which detection method to use, runs the chosen path, and normalises the result to one of the four standard mood labels before handing it to the recommendation engine.

4.4.4 Recommendation and Nutrition Module

The recommendation and nutrition module calculates caloric targets from BMI and TDEE, applies the allergen and dietary preference filters to the meal catalogue, and scores the remaining meals using the four-factor weighted algorithm.

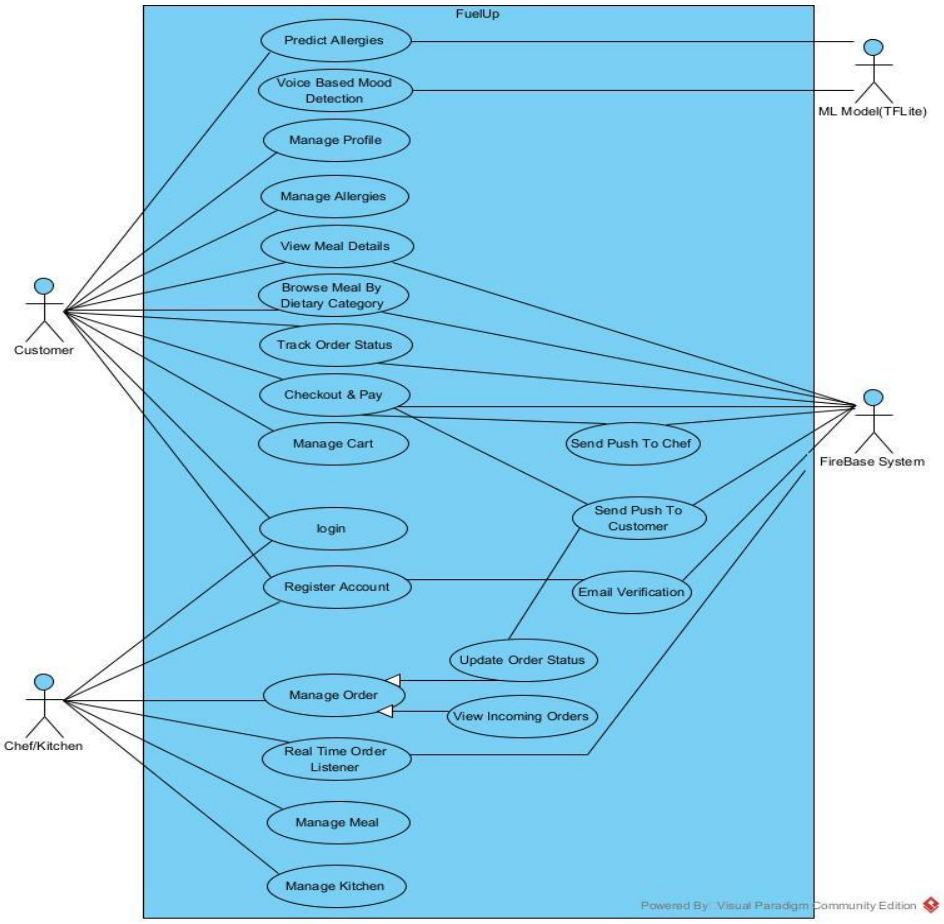

Figure 4.1: Use Case Diagram

The **Use Case Diagram** illustrates the high-level functional interactions between the Customer, Chef, and external actors like Firebase and the TFLite ML model. It maps core system capabilities including mood detection, nutritional profile management, and the end-to-end real-time order lifecycle. This provides a structured view of how various user roles utilize the underlying service integrations to achieve project goals.

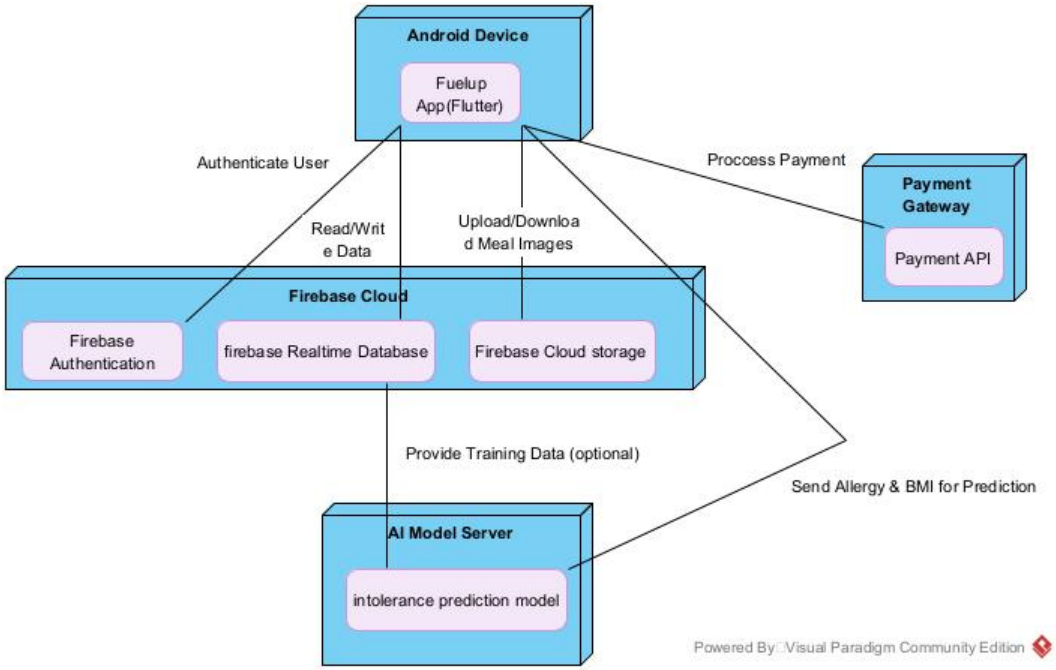

Figure 4.2: Deployment View

The **Deployment Diagram** illustrates the physical distribution and connectivity between the Flutter mobile client, Firebase Cloud services, and external APIs for payments and AI predictions. It defines the communication paths required for real-time data synchronization, user authentication, and distributed intelligence processing.

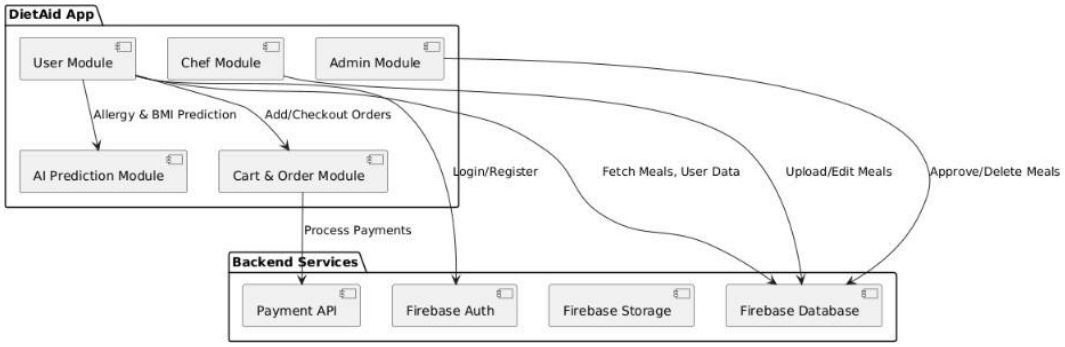

Figure 4.3: Component Diagram

The **Component Diagram** highlights the internal modular structure of the application, illustrating the dependencies between frontend modules—User, Chef, and Admin—and external backend services. It maps how critical functions like AI predictions and payment processing rely on specific system components and Firebase infrastructure to maintain the application's overall logic.

4.4.5 Ordering and Notification Module

The ordering and notification module writes order records to the Realtime Database, subscribes to status update events, and handles the checkout flow through to order confirmation. It also takes care of registering the chef's FCM token.

4.5 Data Models

The core data structures used throughout the system are described below.

4.5.1 Firestore Collections

The Firestore users collection stores the user's role, email address, and current FCM push token. Customer profiles are stored in a separate customer-details sub-collection with BMI data, dietary preferences, and allergen flags. Chef profiles store kitchen details and active meal listings in the kitchenMeals collection.

4.5.2 Realtime Database Structure

The Realtime Database stores order records at the orders path. Each order document holds the order ID, customer ID, chef ID, line items with quantities, delivery address, payment method, total, and status. Status transitions follow a fixed sequence: pending → accepted → preparing → ready → delivered.

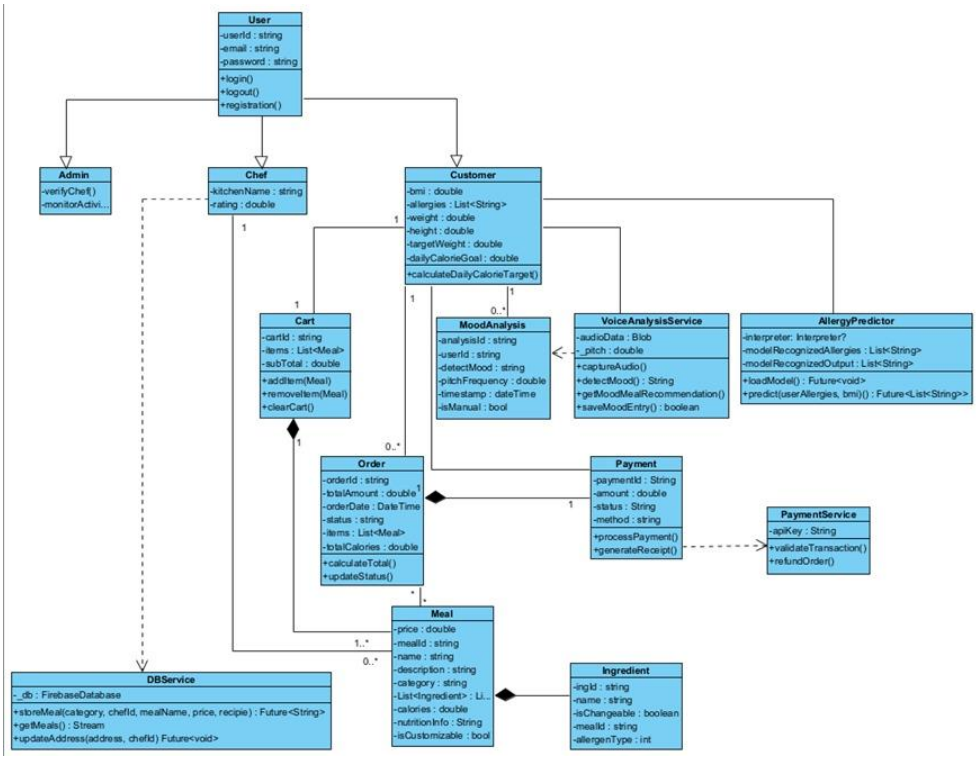

Figure 4.4: Class Diagram

The **Class Diagram** defines the system's static structure, detailing the attributes, methods, and relationships between entities such as Users, Chefs, and Customers. It illustrates the object-oriented design through inheritance and demonstrates how modular services like VoiceAnalysis and AllergyPredictor interact with the core data model. This serves as the primary blueprint for implementing the application's underlying logic and data associations.

4.5.3 Mood Model

Mood state is constrained to four labels matching the SER model's output classes: neutral, happy, surprise, and unpleasant. Keeping it to exactly these four means the recommendation engine always receives a value it knows how to handle.

4.6 Dynamic View

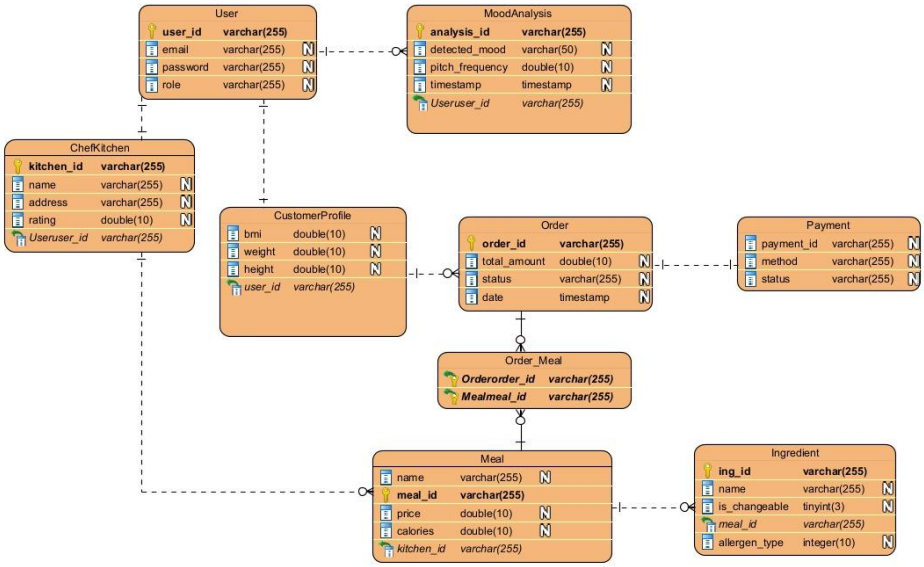

Figure 4.5: ER Diagram (Logical View)

The **ER Diagram (Logical View)** illustrates the relational database schema, mapping key associations between core entities like Users, MoodAnalysis, and Orders. It establishes critical links between customer health metrics and meal data to facilitate the mood-aware recommendation engine . This structure serves as the primary data model for maintaining consistency across the system's marketplace and intelligence modules.

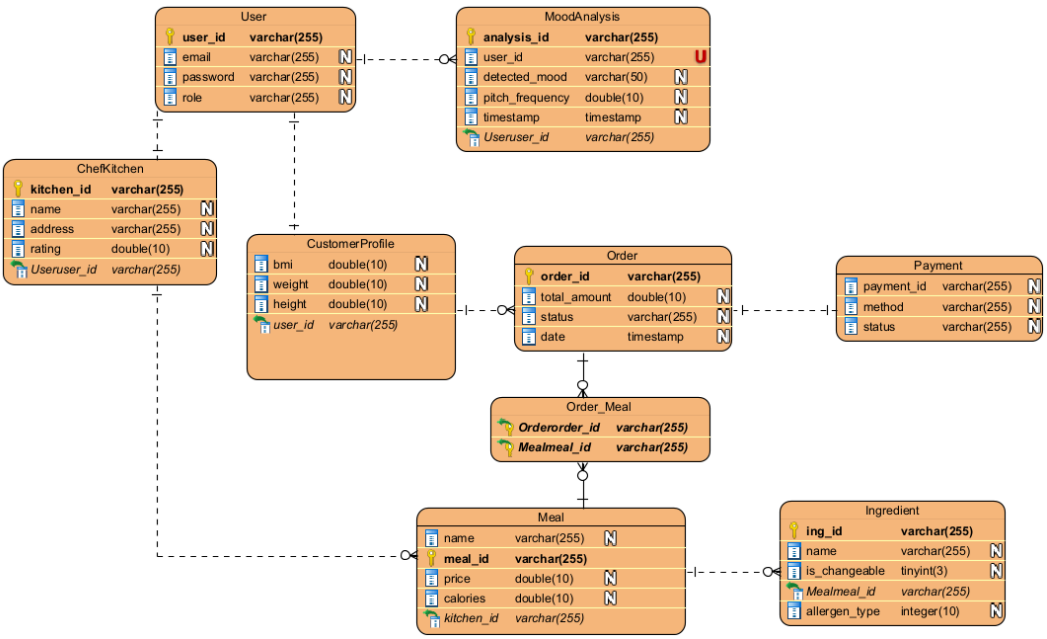

Figure 4.6: Data Model

The **Data Model** defines the logical database schema for the FuelUp system, detailing the entities and attributes required to support both marketplace operations and personalized features. It establishes critical relationships between customer health profiles, mood analysis results, and order transactions to ensure data consistency across the platform's services.

4.6.1 Meal Creation and Tagging Flow

When a chef adds a meal, it gets written to the kitchenMeals Firestore collection. That write triggers the Node.js backend listener, which queues a tagging job. The rule-based engine runs first, unconditionally, producing at least a baseline tag set. If the Gemini quota is available, the AI path also runs and its output gets merged into the tags.

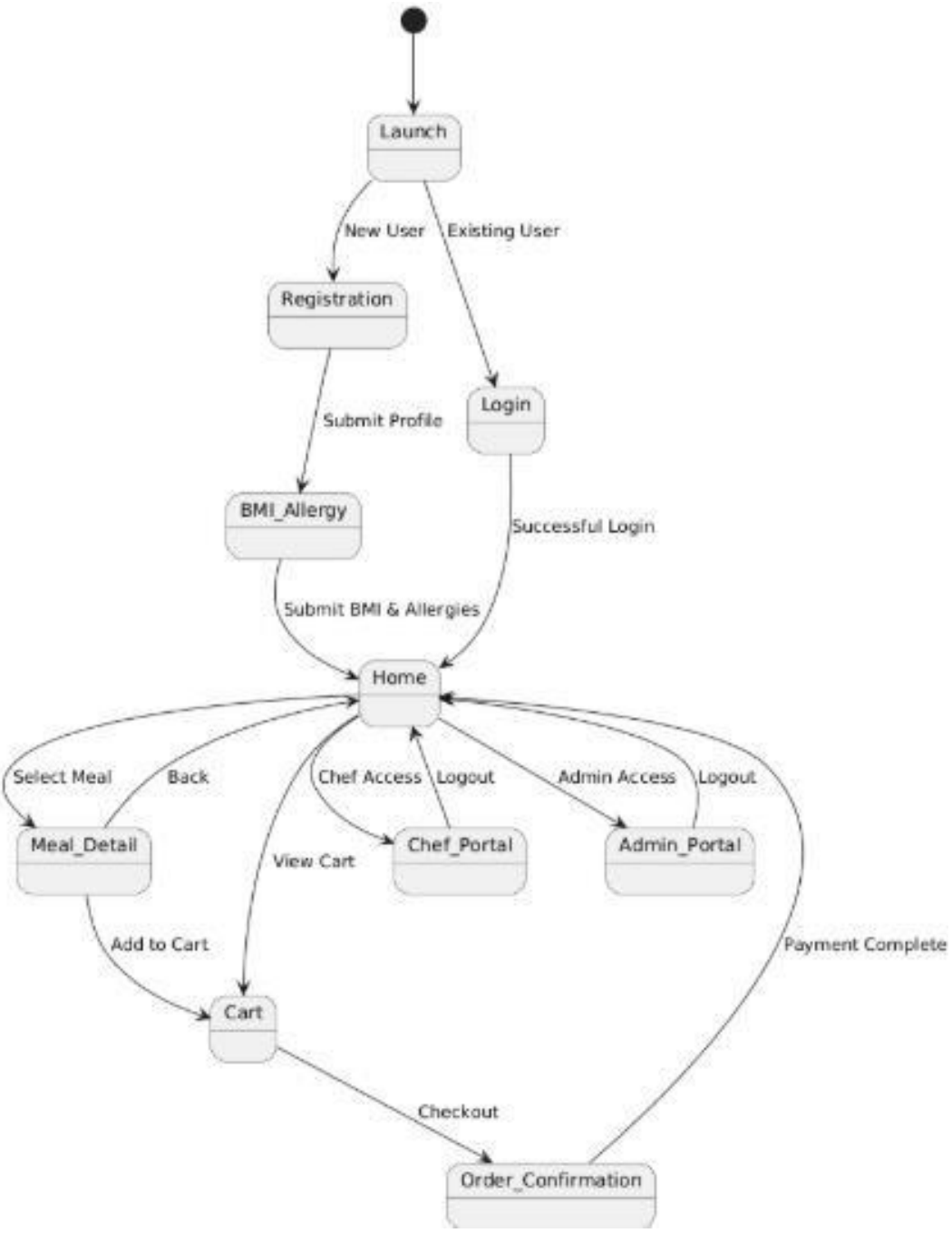

Figure 4.7: System State Diagram

This diagram illustrates the application's navigation lifecycle, tracing user flow from launch and authentication through to final order confirmation. It defines the transitions between the home screen and specialized functional modules like the Chef and Admin portals.

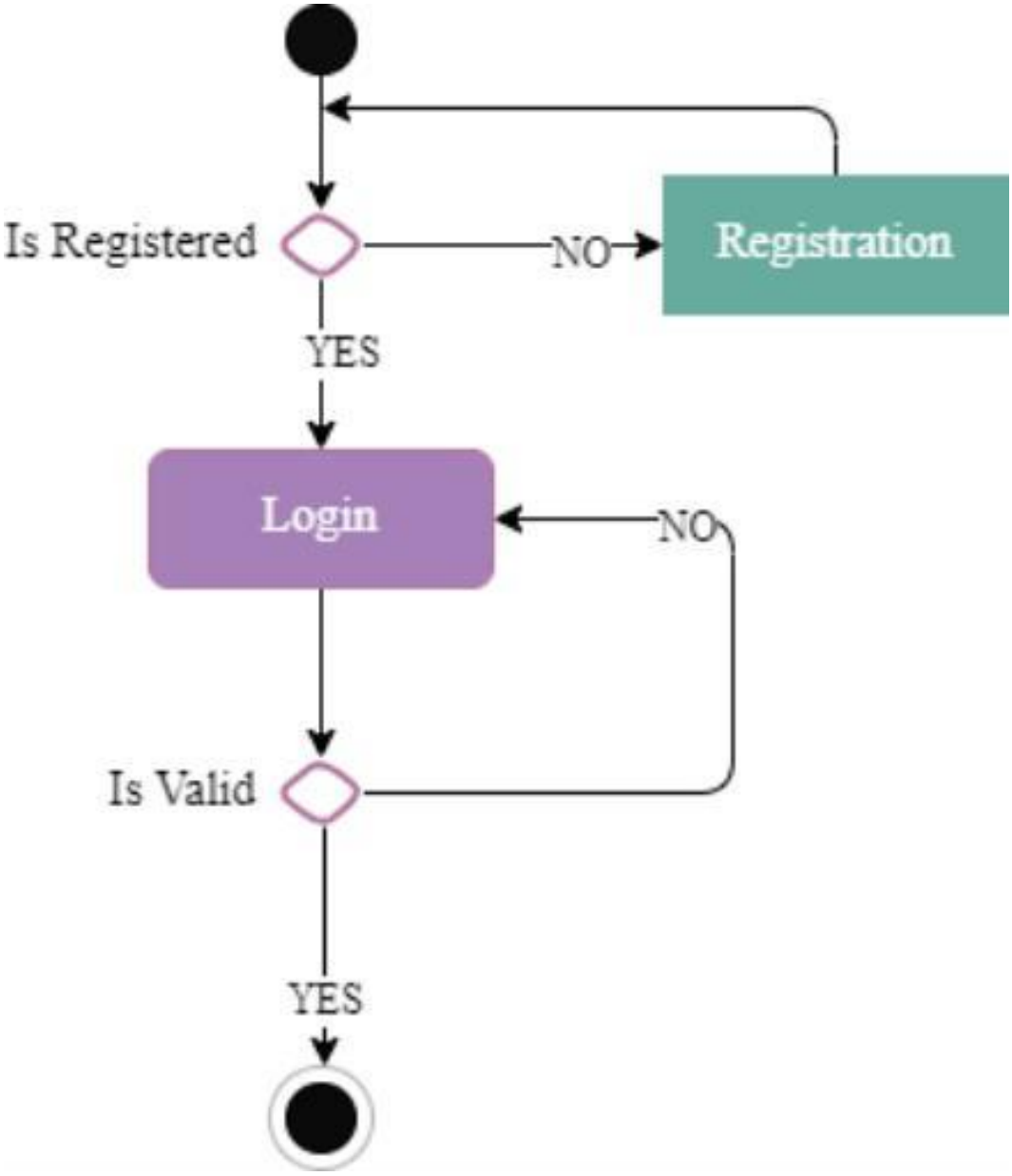

Figure 4.8: State Diagram – Login Flow

This state diagram illustrates the logic for user authentication, defining decision paths for registration status and credential validity. It ensures users are correctly routed to registration or the login interface before accessing the system's personalized features.

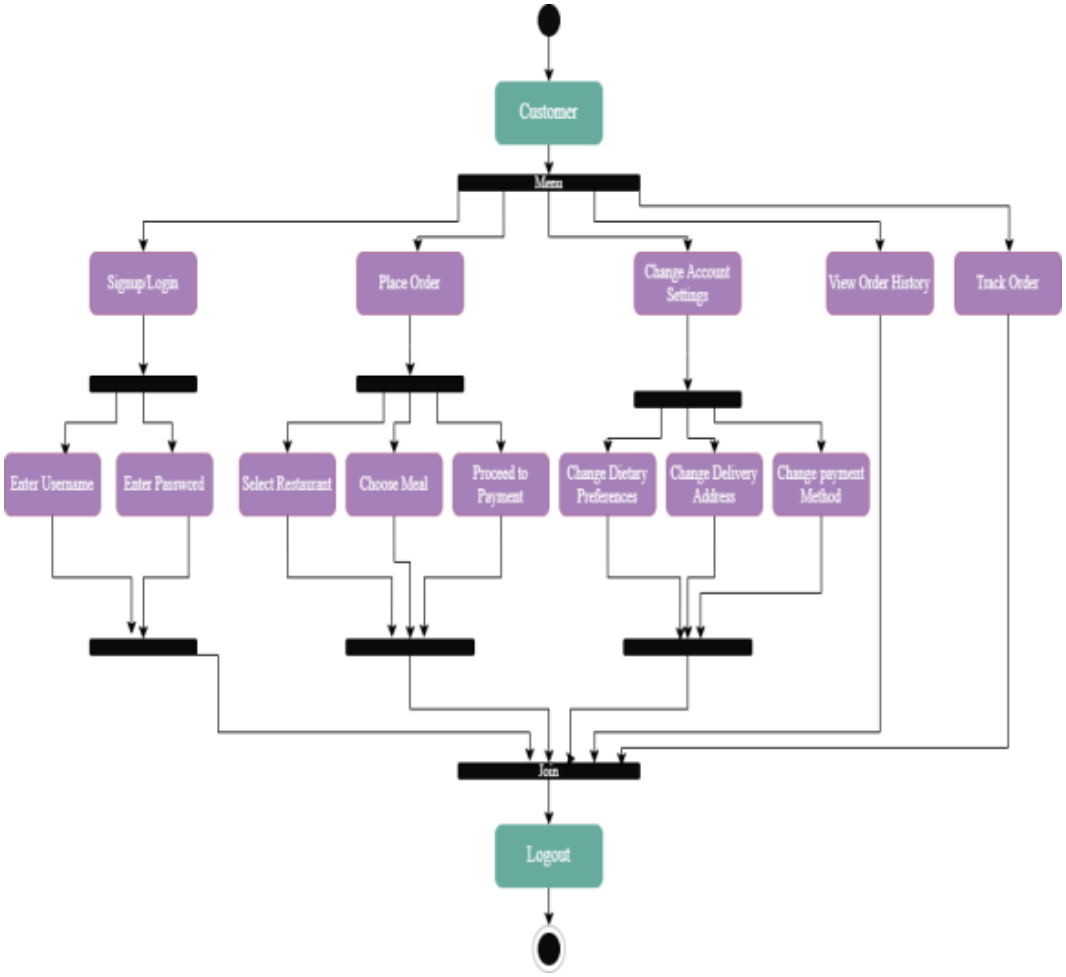

Figure 4.9: State Diagram – Customer Flow

This state diagram maps the functional navigation paths available to a Customer, ranging from account management and profile customization to the core ordering and tracking processes. It illustrates how various sub-activities converge at the logout state to conclude the user session.

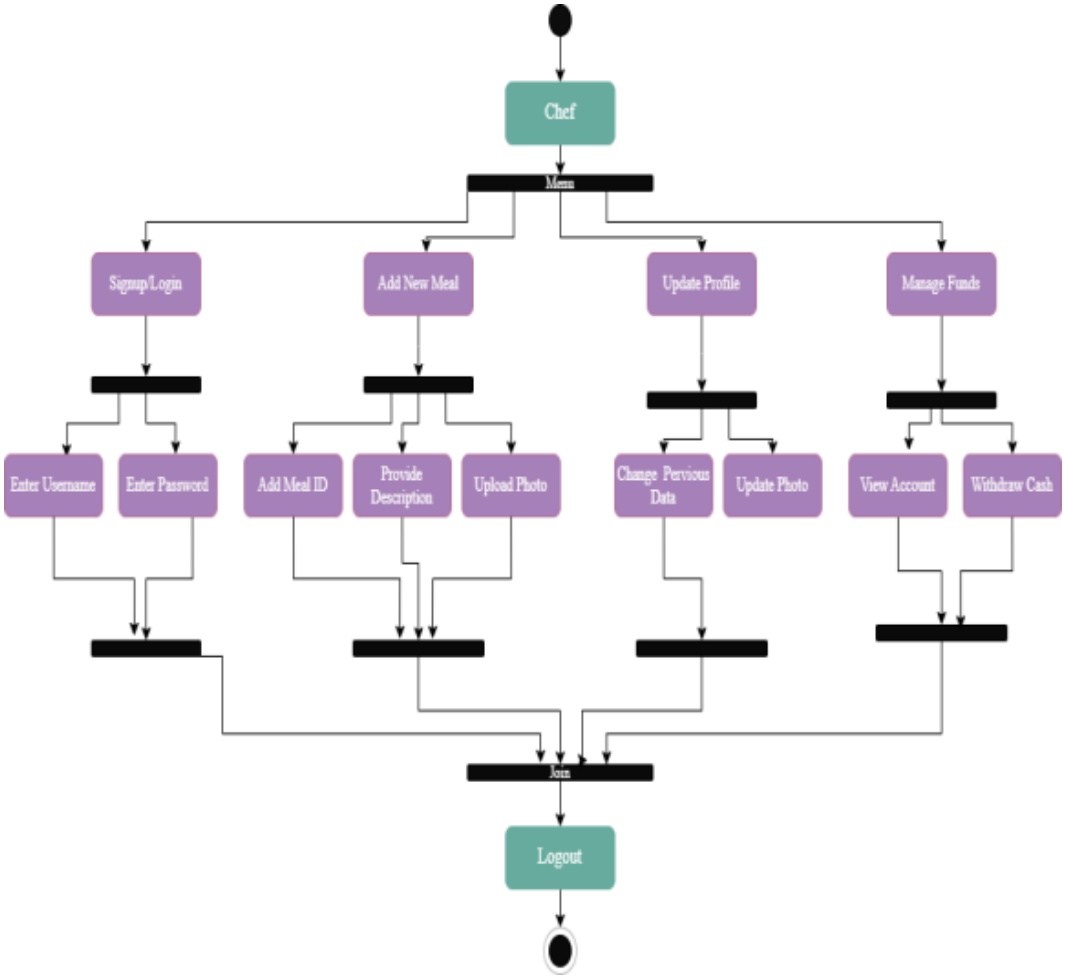

Figure 4.10: State Diagram – Chef Flow

This state diagram maps the operational lifecycle for the Chef, detailing core activities like meal management, profile updates, and fund withdrawal. It illustrates the logical flow of tasks from initial authentication through various administrative modules to the final logout.

4.6.2 Recommendation Generation Flow

When the customer opens the home screen, the app fetches meal categories and kitchen-grouped listings from Firestore. The recommendation subsystem then checks if a mood is currently set. If yes, it applies four-factor scoring. If not, it falls back to a basic calorie-aligned sort. Meals that conflict with the user's allergen or dietary flags are excluded completely, regardless of score.

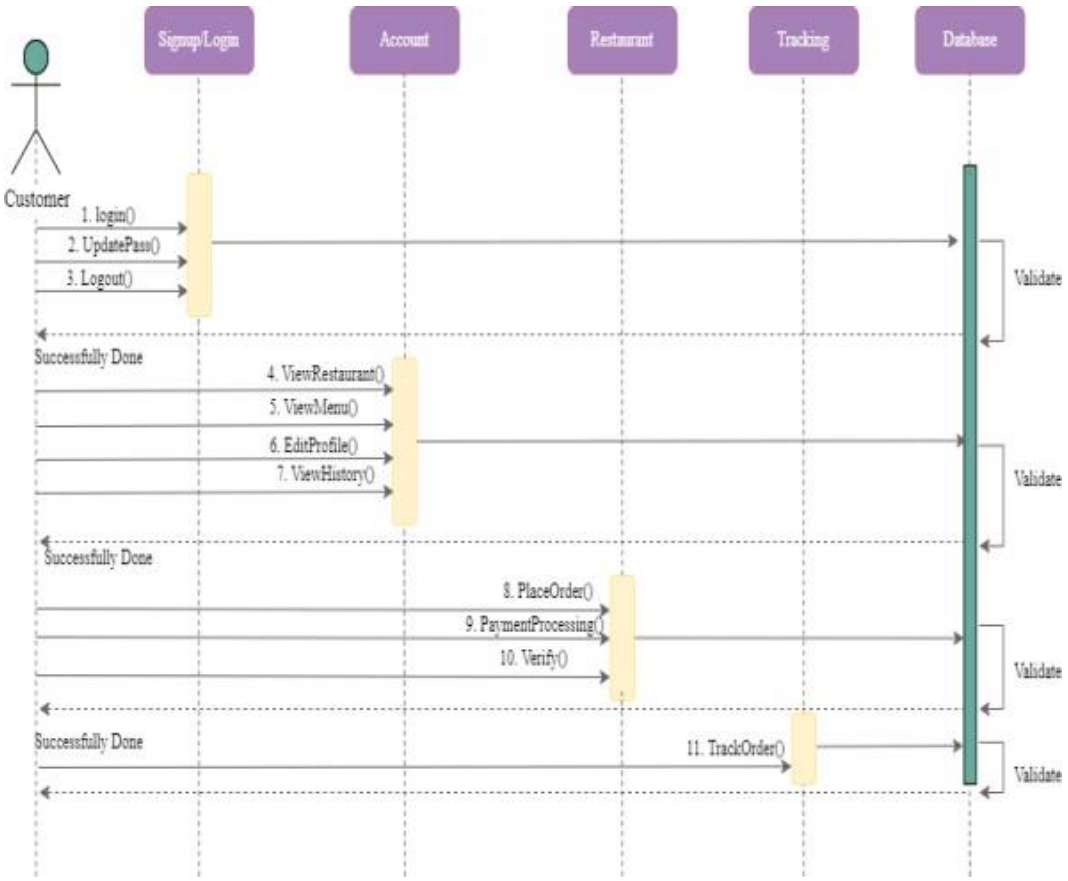

Figure 4.11: Sequence Diagram – Customer Flow

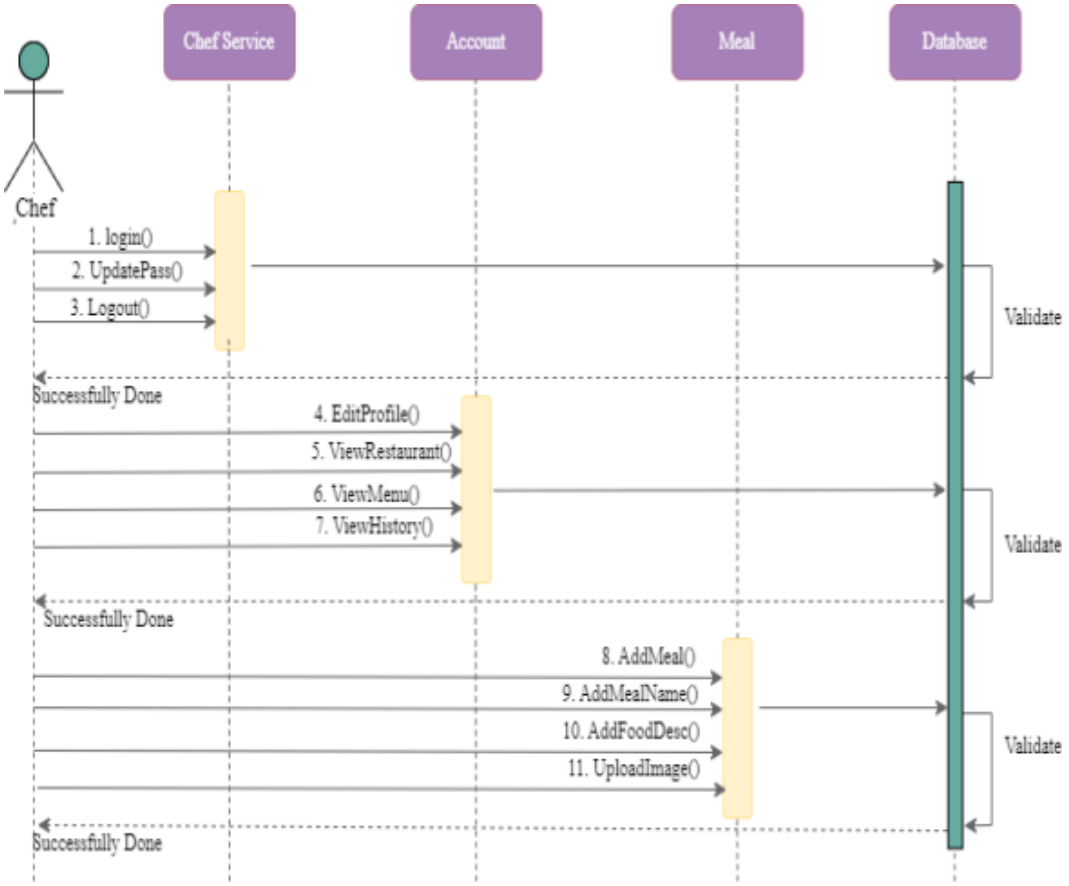

Figure 4.12: Sequence Diagram – Chef Flow

4.6.3 Order Fulfilment Flow

Checkout begins when the customer confirms the cart and fills in the delivery form. One order record per chef gets written to the Realtime Database — if the cart has items from two different kitchens, two separate orders are created. The backend picks each one up, validates it, and dispatches an FCM notification to the relevant chef.

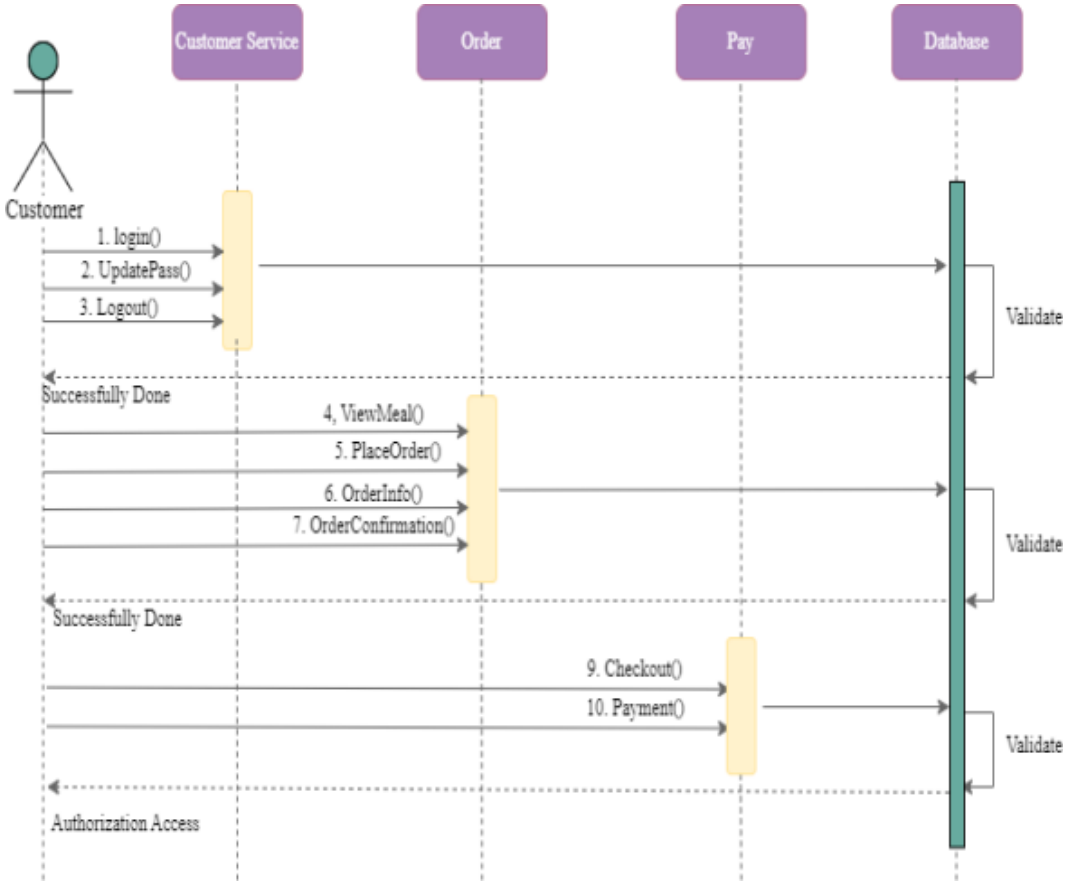

Figure 4.13: Sequence Diagram – Payment and Order Fulfilment

4.7 User Interface Design

The UI follows Material Design with a warm colour palette chosen to match the food context. I kept screen layouts intentionally simple — category grids, meal cards showing calories and price, and clear action buttons. For mood detection, I used a single large microphone button so it is immediately obvious how to use it.

The checkout screen was deliberately kept minimal — delivery address and payment method, that is all. Fewer steps between cart and confirmation means less friction and less likelihood of the user dropping off.

Figure 4.14: FuelUp Application Logo

The **FuelUp logo** combines a minimalist four-tined fork with organic leaf accents to represent the fusion of modern technology and natural nutrition. The deep green palette reflects the brand's commitment to health, wellness, and fresh, personalized dining.

Figure 4.15: Login Screen

This screen represents the primary entry point for authenticated users of the FuelUp platform. It provides a clean, Material Design-based interface for users to enter their credentials (email and password). The system validates these credentials against Firebase Authentication to grant access to role-specific dashboards.

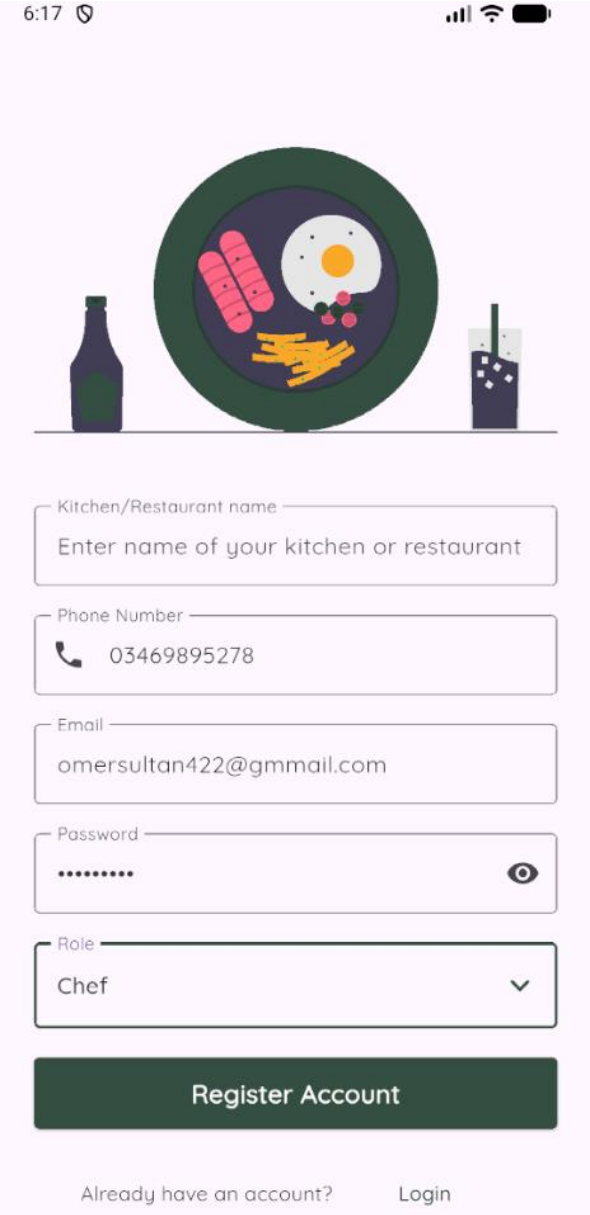

Figure 4.16: Signup Screen

The Chef Registration Screen facilitates the onboarding of food providers by capturing essential details like kitchen name, contact information, and secure credentials. It integrates with Firebase Authentication for account creation while simultaneously establishing a professional profile in Cloud Firestore. This ensures users selected for the "Chef" role are correctly provisioned with access to specialized order management and meal listing tools.

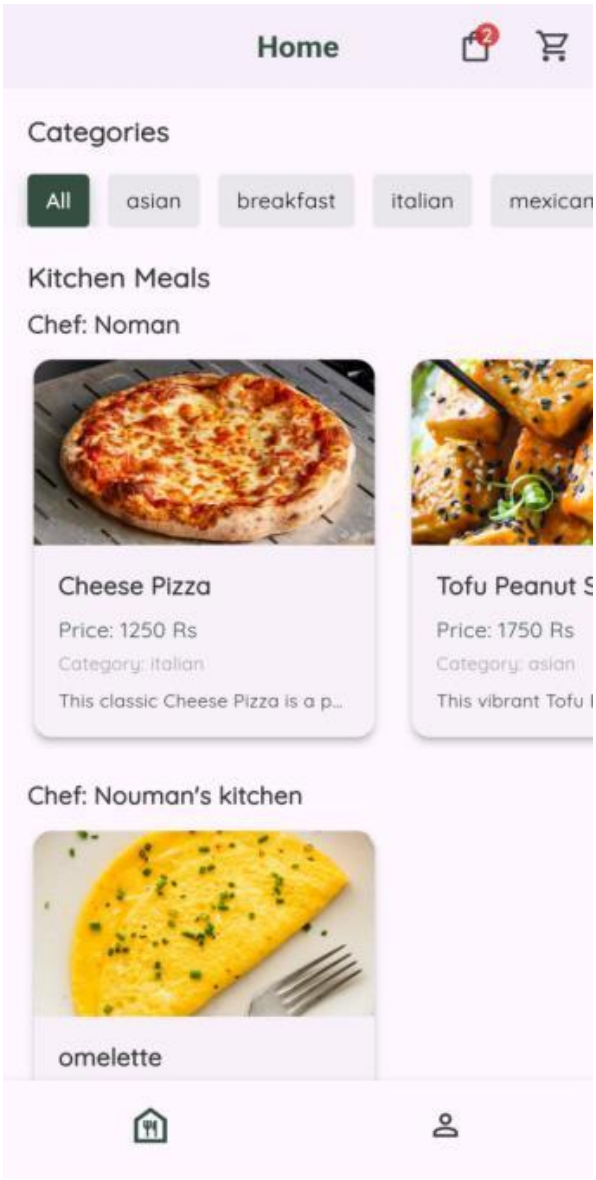

Figure 4.17: Home Page

The main browsing interface for Customers, displaying a variety of meals grouped by their respective kitchens. The UI features a horizontal category filter (e.g., Asian, Breakfast, Italian) and meal cards that display the item image, price, and chef's name. This screen serves as the primary surface for the mood-aware recommendation engine, which ranks these listings based on the user's active mood and health profile.

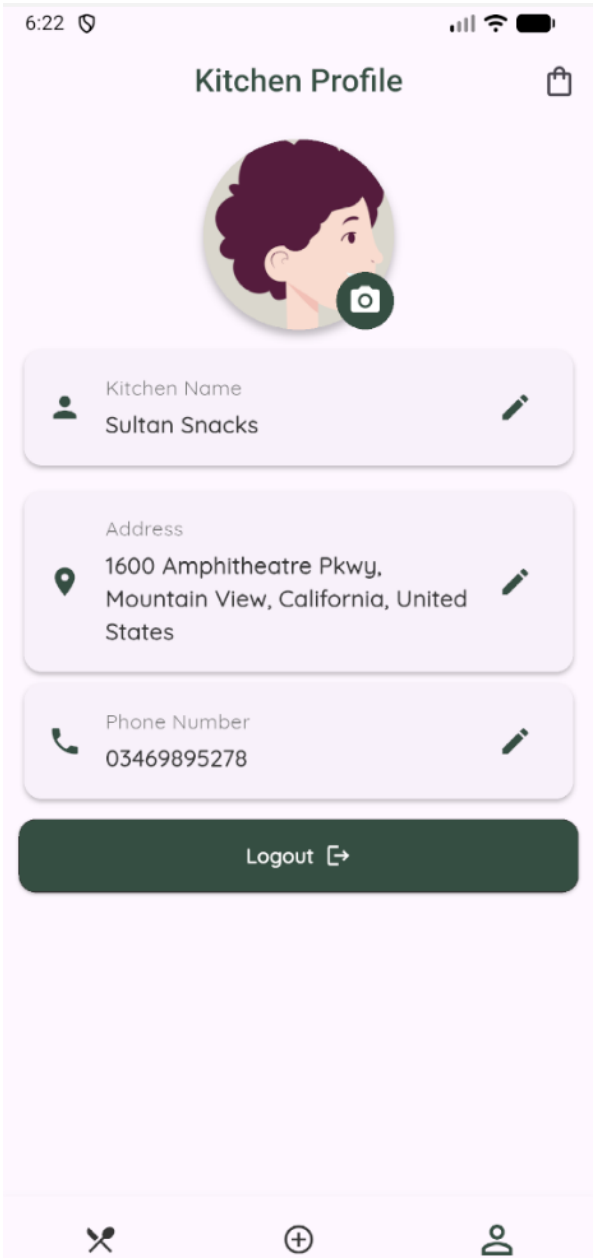

Figure 4.18: Account / Profile Page

This screen allows Chefs to manage their professional identity within the application. It displays the kitchen's name, physical address, and phone number, with options to edit each field. Maintaining accurate profile data is essential for the marketplace's operational feasibility and ensures customers have reliable contact information for their orders.

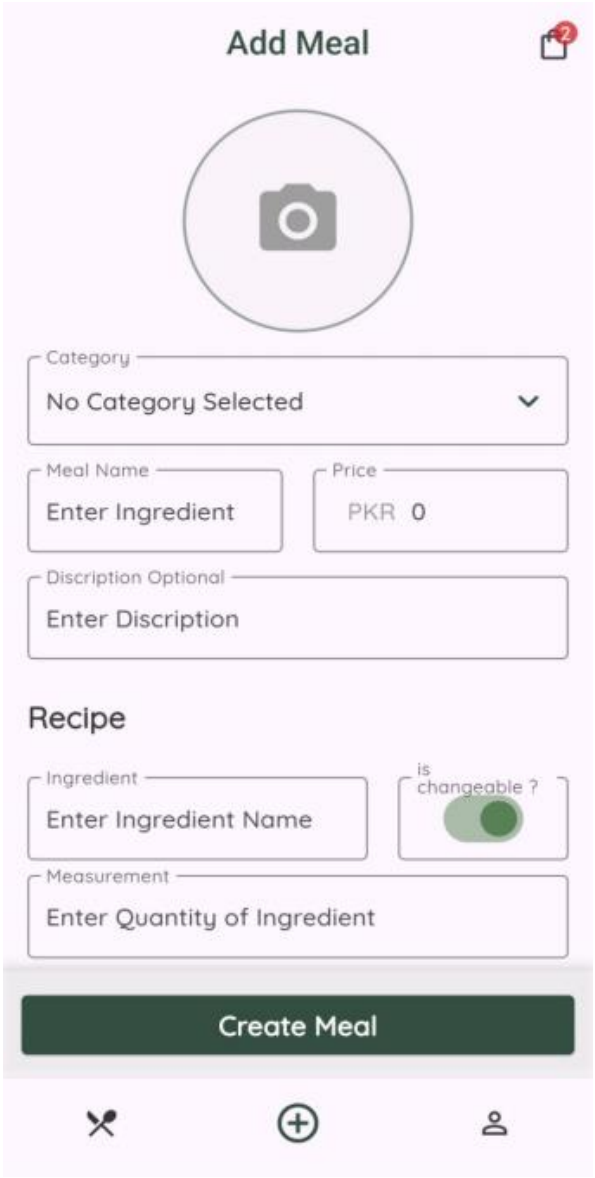

Figure 4.19: List Meal – Chef Interface

The Chef's interface for publishing new items to the marketplace. Beyond basic details like price and category, it includes a granular Recipe section where chefs must input specific ingredients and measurements. This structured data is vital for the hybrid tagging backend, as it allows the rule-based lexical engine to identify allergens and nutritional labels automatically.

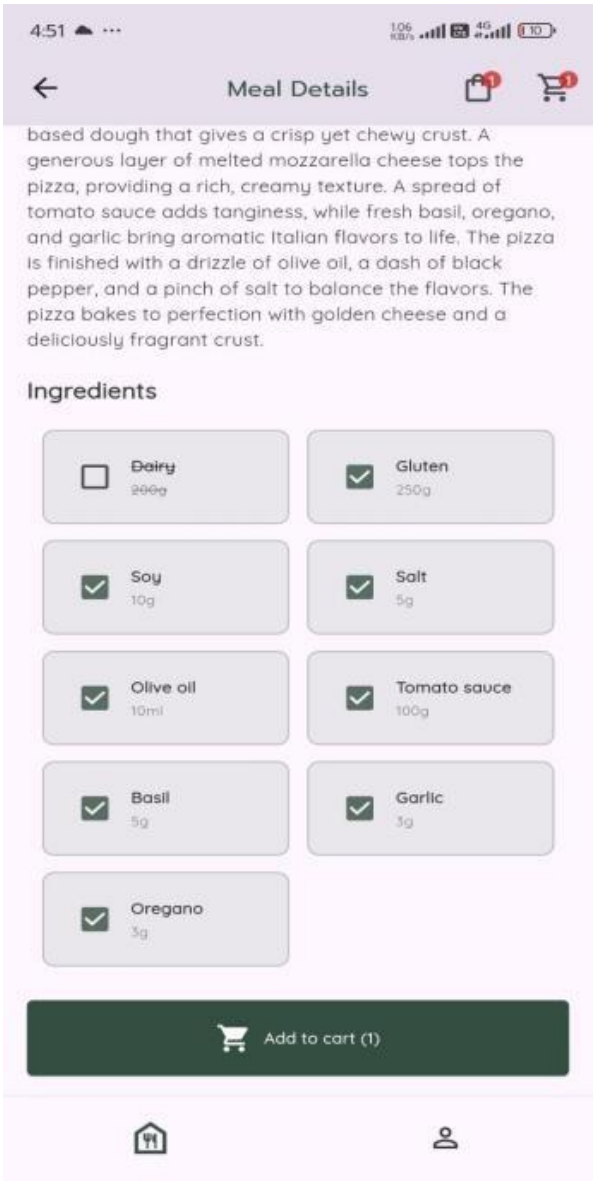

Figure 4.20: Allergy-Aware Ingredient Exclusion

A detailed view of a specific meal (e.g., Cheese Pizza) showing the full description and a checklist of ingredients. This interface demonstrates the allergy-aware filtering system; ingredients identified as common allergens (such as Dairy or Gluten) are clearly labeled. This level of transparency helps users make nutrition-conscious decisions aligned with the dietary preferences set in their health profile.

Figure 4.21: Cart Screen

This interface acts as the local staging area where customers can review and manage their meal selections before finalising an order. The screen supports local data persistence across sessions and allows users to increment or decrement item quantities, which triggers a real-time update of the order subtotal.

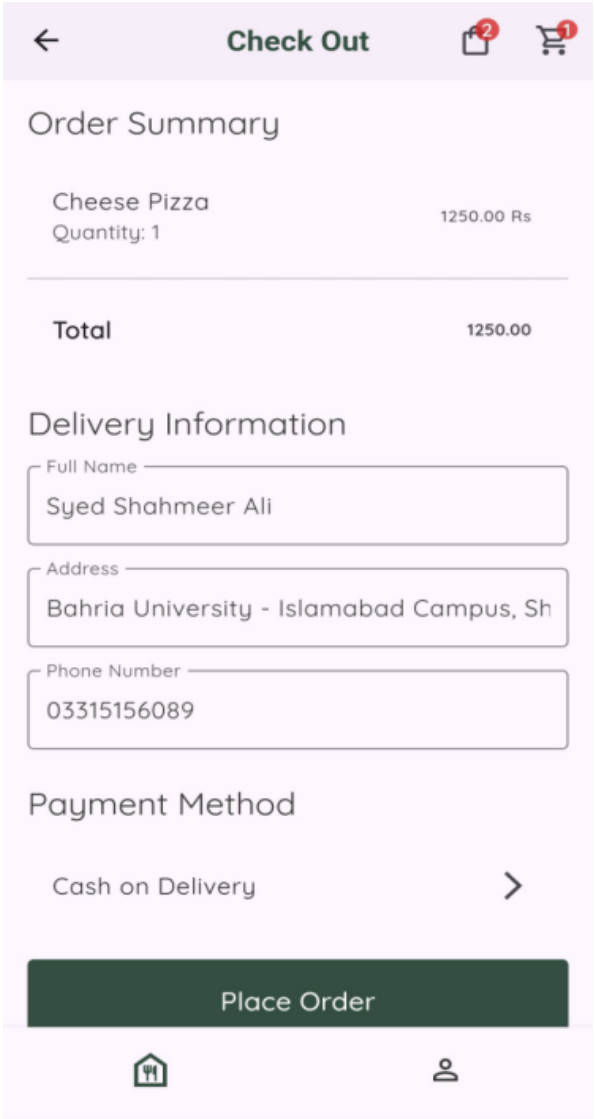

Figure 4.22: Complete Payment Screen

This final transactional screen captures the customer's delivery information and preferred payment method to complete the order lifecycle. Once the user confirms the order, the system executes a structured write to the Firebase Realtime Database and dispatches an automated push notification to the relevant chef.

4.8 Conclusion

This chapter has walked through the architecture and modular structure of FuelUp. The separation between the Flutter client, Firebase layer, Node.js backend, and intelligence components gives the codebase a structure that can be extended reasonably without needing to tear things apart. The next chapter goes into how each part was actually built.

Chapter 5: System Implementation

This chapter covers how each major subsystem was actually built — the technology choices I made, the important code structures, and the places where things got complicated or unexpected.

5.1 Technology Stack

The client is written in Dart with Flutter. I chose Flutter primarily because it compiles to native Android and iOS from one codebase — that was essential given I was working alone with a limited timeline. Dart's type system and Flutter's widget model made building the layered UI reasonably manageable.

The main Flutter packages I used: the full Firebase SDK family (firebase_core, firebase_auth, cloud_firestore, firebase_database, firebase_messaging, firebase_storage), record for audio capture, tflite_flutter for on-device inference, and go_router for navigation. Provider handles state management throughout.

The backend uses Express on Node.js, with firebase-admin for database access, bull for the tagging job queue, and the Gemini API client for AI-assisted tagging. The render.yaml file sets up the Render deployment as a web service running Node 18.

5.2 Application Bootstrap and Routing

App startup in main.dart initialises Firebase, resets the Gemini quota guard session state, and launches the root widget. GoRouter handles navigation with a redirect callback that checks auth state and user role before allowing access to any protected screen.

Every protected screen is behind role checks in the router. Customers who somehow reach a chef screen get redirected to the customer home, and vice versa. Unauthenticated users trying to access anything protected get sent to login.

5.3 Authentication Module

auth_service.dart wraps all the Firebase Authentication calls. Registration creates a Firebase user then writes the role and profile to Firestore in one transaction. I put an email verification gate in front of the main app so unverified users cannot proceed.

Password reset uses Firebase's built-in sendPasswordResetEmail. To handle the edge case where one user logs out and a different user logs in on the same device, I write a logout marker to shared preferences. The router checks this on startup and clears the cached session when the marker is present.

5.4 Voice Mood Detection

The voice mood detection pipeline in voice_mood_detector.dart was without question the most technically involved part of the entire project. The sequence is: record audio → parse the WAV file → extract MFCC features → run TFLite inference → map the result to a mood label.

1. Audio capture uses the record package at a fixed sample rate, writing to a temporary WAV file. WAV parsing is done in Dart by reading the file header to locate the PCM data chunk and extract the raw sample array.
2. The WAV file is parsed in Dart to extract the raw PCM audio samples from the data chunk.
3. MFCC extraction in mfcc_extractor.dart applies pre-emphasis filtering, divides the signal into overlapping frames with Hamming windowing, computes the Fast Fourier Transform for each frame, applies a mel filterbank, takes the log, and applies the Discrete Cosine Transform. The result is a feature matrix of the shape required by the TFLite interpreter.
4. The feature matrix is reshaped into the input tensor format and passed to the TFLite interpreter. The model returns probability scores across the four emotion classes. Argmax selection picks the winning class and maps it to the corresponding mood label.
5. The output tensor contains probability scores for four emotion classes. Argmax selection determines the predicted class, which is mapped to

the corresponding MoodType enum value (neutral, happy, surprise, unpleasant).

The detectMoodWithFallback utility wraps the entire SER pipeline. If inference completes successfully the result goes straight through. If anything fails along the way — audio capture error, WAV parse failure, or low model confidence — it falls back to the tag-based classifier, which estimates mood from the user's recent meal browsing behaviour.

5.5 Tag-Based Mood Classification

tag_based_mood_detector.dart is the deterministic fallback. It takes tag sets from meals the user has recently browsed, normalises the overlap against each mood's tag dictionary, and returns the mood with the highest score. No randomness, no ML model — just a lookup table and a scoring function.

5.6 Meal Recommendation Engine

The recommendation engine in mood_meal_filter.dart scores every meal that survives the allergen and dietary filters using four components.

- Mood Score: A value in [0, 1] based on the overlap between the meal's tag set and the compatible tags for the active mood label.
- Calorie Score: A value in [0, 1] inversely proportional to the absolute difference between the meal's caloric value and the customer's computed goal caloric target, normalised by a tolerance band.
- BMI Score: A value derived from the customer's BMI category (underweight, normal, overweight, obese) and the meal's nutritional density and prep style labels.
- Goal Score: A value reflecting alignment between the meal's caloric density direction (light, moderate, rich) and the customer's target-weight direction (loss, maintenance, gain).

The four component scores get combined into a weighted sum. The weights are not fixed — they adjust dynamically based on context. If the customer has a significant weight management goal, caloric proximity and goal direction get

higher weights. If no mood has been detected or set, the mood score component is dropped and the remaining weights are redistributed.

5.7 Automated Meal Tagging Backend

The tagging pipeline lives in the render-backend directory. When the Firestore listener sees a new kitchenMeals document, it adds a job to the bull queue. This keeps the meal write path non-blocking — the meal appears in the app immediately, and the tags follow once the job processes.

meal_tagging_worker.js pulls jobs from the queue and runs the tagging logic. The rule-based lexical analyser always runs unconditionally — every meal gets at least a basic tag set regardless of API availability. Allergen flags, dietary labels, and caloric category all come from the ingredient list and description text.

The Gemini enhancement path in meal_tagger.js builds a structured prompt from the meal data and calls the Gemini API. The returned JSON — if valid — is merged with the rule-based tags. A quota guard tracking daily API call count prevents runaway usage and ensures the system degrades gracefully when the free tier limit is reached.

5.8 Order Management and Notification

Customer orders get written to the Realtime Database orders path through realtime_database.dart. The order document structure is fixed at creation time — subsequent writes only update the status field, they do not change the document shape.

The notifyChef handler in notifyChef.js validates the order data, builds the FCM payload, and dispatches it to the chef's registered token. If the token is stale or invalid, the error gets logged but the order itself does not fail — the chef can still see it in the kitchen dashboard through polling.

5.9 Backend Deployment

The backend deploys to Render as a web service via render.yaml. Render handles restarts and provides a public HTTPS endpoint. The known issue here is that Render's free tier spins down idle services — so after a quiet period there is a cold start delay on the first order notification, which in testing could be noticeable.

5.10 Conclusion

The implementation covered a lot of ground. MFCC extraction in Dart was the single hardest individual task — there is not much prior art for doing it from scratch in Flutter. The hybrid tagging pipeline and the quota guard were probably the most important decisions for keeping the system reliable. The next chapter covers testing.

Chapter 6: System Testing and Evaluation

This chapter covers the testing I did on FuelUp — what I tested, what I found, and what the results actually mean. I have also tried to be honest about where the test coverage is thin.

6.1 Test Strategy

Testing covered three levels. Automated unit tests focused on the mood intelligence components. Manual functional testing went through all the end-to-end user flows — registration, meal browsing, mood detection, checkout, order tracking, and chef order management. I also did a recommendation quality evaluation using manually constructed customer profiles.

For UI testing I used an Android emulator at API level 33. Voice mood detection testing had to be done on a physical Android device because emulator microphone input is too unreliable. Backend endpoint testing used a Postman collection.

6.2 Unit Testing

The main automated test file is tag_based_voice_alignment_test.dart, which tests the tag-based mood classifier. The tests check that each of the four mood labels wins correctly when given a representative input tag set. They run via the standard flutter test command.

Test cases covered all four mood labels — neutral, happy, surprise, and unpleasant — with both clean and noisy tag inputs. All passed, which gave me reasonable confidence that the scoring logic handles the expected input range correctly.

6.3 Functional Testing

Manual testing went through every use case from Chapter 3. Registration, login, meal browsing, allergen filtering, voice mood detection, manual mood selection,

recommendation display, cart management, checkout, order tracking, and chef notifications all worked correctly in my test environment.

Table 18: Summary of Functional Test Case Results

TC-ID	Test Description	Expected Result	Pass / Fail
TC-01	Customer registers with valid credentials and completes email verification.	Profile created in Firestore; redirected to home screen.	Pass
TC-02	Customer logs in with correct credentials.	Role loaded; customer home displayed.	Pass
TC-03	Customer activates voice mood button and speaks for 3 seconds.	Mood state updated and displayed on home screen.	Pass
TC-04	Customer with nut allergy declared views meal list. Meals tagged with nut allergen are absent from recommendation section.	Allergen-conflicting meals excluded from recommendations.	Pass
TC-05	Chef creates a new meal with ingredients and calories.	Meal appears in Firestore; tags generated by backend within 30 seconds.	Pass
TC-06	Customer completes checkout with cash payment selection.	Order written to RTDB; chef receives FCM notification.	Pass
TC-07	Chef updates order status from pending to preparing.	Customer receives push notification; order screen status updates.	Pass
TC-08	Voice recording attempted without microphone permission.	Permission request dialog shown; graceful handling on denial.	Pass
TC-09	Customer profile updated with new weight and height values.	Recommendation scores recalculated on next home screen load.	Pass

TC-ID	Test Description	Expected Result	Pass / Fail
TC-10	Backend Render service restarted; new meal creation triggers tagging after listener reconnection.	Tagging completes after service restart; meal tags updated.	Pass with latency

The following test cases were executed to validate the core functional requirements of FuelUp. Each uses the standard departmental template: scenario ID, case ID, priority, pre/post-conditions, and step-by-step execution with expected vs actual outcomes and pass/fail status.

Test Case TC-01: User Registration – Positive Test Case

Table 19:TC-01-User Registration

Test Scenario ID	TC-01	**Test Case ID**	TC-01A
Test Case Description	User Registration – Positive Test Case	**Test Priority**	High
Pre-Requisite	App installed; no existing account for the email	**Post-Requisite**	User account created in Firebase; profile stored in Firestore

Test Execution Steps:

S.No	Action	Inputs	Expected Output	Actual Output	Test Browser	Test Result	Test Comments
1	Launch application	FuelUp app icon	Registration screen loads	Registration screen loads	Android 11	Pass	App launches successfully
2	Enter valid name, email, password and submit	Name: Omer Sultan Email: omer@test.com Password: Test@123	Account created; customer home shown	Account created; home shown	Android 11	Pass	Profile written to Firestore

Test Case TC-02: Customer Login – Positive Test Case

Table 20:TC-02- Customer Login

Test Scenario ID	TC-02	**Test Case ID**	TC-02A
Test Case Description	Customer Login – Positive Test Case	**Test Priority**	High
Pre-Requisite	Valid registered and email-verified account exists	**Post-Requisite**	Customer home screen displayed with correct role loaded

Test Execution Steps:

S.No	**Action**	**Inputs**	**Expected Output**	**Actual Output**	**Test Browser**	**Test Result**	**Test Comments**
1	Launch application	FuelUp app icon	Login screen loads	Login screen loads	Android 11	Pass	App launches correctly
2	Enter correct email and password; tap Login	Email: omer@test.com Password: Test@123	Login success; customer home shown	Login success; customer home shown	Android 11	Pass	Role loaded from Firestore

Test Case TC-03: Voice Mood Detection – Positive Test Case

Table 21:TC-03-Voice Mood Detection

Test Scenario ID	TC-03	**Test Case ID**	TC-03A
Test Case Description	Voice Mood Detection – Positive Test Case	**Test Priority**	High
Pre-Requisite	Customer logged in; microphone permission granted	**Post-Requisite**	Mood state updated and home screen recommendations re-ranked

Test Execution Steps:

S.No	**Action**	**Inputs**	**Expected Output**	**Actual Output**	**Test Browser**	**Test Result**	**Test Comments**
1	Tap the mood microphone button	Microphone button tap	Recording starts; countdown displayed	Recording starts; countdown shown	Android 11	Pass	Microphone activates correctly

2	Speak for 3 seconds in a happy tone	Voice input: happy speech sample	Mood detected as Happy; meals re-ranked	Mood detected as Happy; meals re-ranked	Android 11	Pass	SER model inference completes on-device

Test Case TC-04: Allergen Filtering – Positive Test Case

Table 22:TC-04-Allergen Mood Detection

Test Scenario ID	TC-04	**Test Case ID**	TC-04A
Test Case Description	Allergen Filtering – Positive Test Case	**Test Priority**	High
Pre-Requisite	Customer profile has nut allergy declared; nut-containing meals exist in catalogue	**Post-Requisite**	All nut-containing meals excluded from displayed results

Test Execution Steps:

S.No	Action	Inputs	Expected Output	Actual Output	Test Browser	Test Result	Test Comments
1	Navigate to meal browsing screen	Home screen tap	Meal catalogue loads	Meal catalogue loads	Android 11	Pass	Firestore query completes
2	Review displayed meal list	Visible meal cards	No meals containing nuts shown in results	No nut meals displayed	Android 11	Pass	Allergen filter applied correctly before rendering

Test Case TC-05: Chef Meal Creation – Positive Test Case

Table 23:TC-05-Chef Meal Selection

Test Scenario ID	TC-05	**Test Case ID**	TC-05A
Test Case Description	Chef Meal Creation – Positive Test Case	**Test Priority**	High
Pre-Requisite	Chef logged in; backend tagging service running on Render	**Post-Requisite**	Meal appears in Firestore; rule-based tags generated within 30 seconds

Test Execution Steps:

S.No	Action	Inputs	Expected Output	Actual Output	Test Browser	Test Result	Test Comments

1	Tap Add Meal from chef dashboard	Chef home screen	Meal creation form loads	Meal form loads	Android 11	Pass	Navigation successful
2	Fill in meal details and submit	Name: Grilled Chicken Cal: 450 Ingredients: chicken, garlic, olive oil	Meal saved; tags generated by backend	Meal saved; tags applied within 20s	Android 11	Pass	Tags visible in Firestore

Test Case TC-06: Customer Checkout – Positive Test Case

Table 24:TC-06-Customer Checkout

Test Scenario ID	TC-06	**Test Case ID**	TC-06A
Test Case Description	Customer Checkout – Positive Test Case	**Test Priority**	High
Pre-Requisite	Customer logged in; cart contains at least one item	**Post-Requisite**	Order written to Realtime Database; chef receives FCM push notification

Test Execution Steps:

S.No	**Action**	**Inputs**	**Expected Output**	**Actual Output**	**Test Browser**	**Test Result**	**Test Comments**
1	Tap Checkout from cart screen	Cart with 2 items	Checkout form loads	Checkout form loads	Android 11	Pass	Cart state preserved on navigate
2	Enter delivery address and select payment method; confirm order	Address: 12 Main St Payment: Cash on Delivery	Order confirmed; order ID displayed	Order confirmed; ID shown	Android 11	Pass	RTDB write successful
3	Verify chef device notification	Chef device listening	Chef receives FCM push notification	Push notification received on chef device	Android 11	Pass	FCM delivery confirmed within 10 seconds

Test Case TC-07: Order Status Update – Positive Test Case

Table 25:TC-07-Order Status Update

Test Scenario ID	TC-07	**Test Case ID**	TC-07A
Test Case Description	Order Status Update – Positive Test Case	**Test Priority**	High
Pre-Requisite	Order exists with status pending; both customer and chef apps active	**Post-Requisite**	Customer sees updated status on order screen; receives push notification

Test Execution Steps:

S.No	**Action**	**Inputs**	**Expected Output**	**Actual Output**	**Test Browser**	**Test Result**	**Test Comments**
1	Chef taps Update Status and selects Preparing	Order detail screen	Status updated to Preparing in RTDB	Status updated to Preparing	Android 11	Pass	RTDB write completes successfully
2	Customer views order tracking screen	Customer order tracking screen	Status shows Preparing; notification received	Status updated; notification received	Android 11	Pass	Realtime listener fires within 5 seconds

Test Case TC-08: Voice Recording Without Permission – Negative Test Case

Table 26:TC-08-Voice Recording Without Permission

Test Scenario ID	TC-08	**Test Case ID**	TC-08A
Test Case Description	Voice Recording Without Permission – Negative Test Case	**Test Priority**	Medium
Pre-Requisite	Microphone permission explicitly denied on the test device	**Post-Requisite**	App handles denial gracefully; no crash; fallback prompt shown

Test Execution Steps:

S.No	**Action**	**Inputs**	**Expected Output**	**Actual Output**	**Test Browser**	**Test Result**	**Test Comments**
1	Tap the mood microphone button with mic permission denied	Microphone button tap	Permission dialog appears; graceful denial message displayed	Permission dialog shown; app continues normally	Android 11	Pass	No crash observed; tag-based fallback mood prompt shown

Test Case TC-09: Profile Update and Recommendation Recalculation – Positive Test Case

Table 27:TC-09-Profile Update And Recommendation Recalculation

Test Scenario ID	TC-09	**Test Case ID**	TC-09A
Test Case Description	Profile Update and Recommendation Recalculation – Positive Test Case	**Test Priority**	Medium
Pre-Requisite	Customer logged in; existing profile with BMI data stored	**Post-Requisite**	Updated BMI recalculates TDEE; recommendations refresh with new caloric targets

Test Execution Steps:

S.No	**Action**	**Inputs**	**Expected Output**	**Actual Output**	**Test Browser**	**Test Result**	**Test Comments**
1	Navigate to profile screen; update weight and height values	Weight: 85 kg; Height: 175 cm	Profile saved; success message shown	Profile saved successfully	Android 11	Pass	Firestore write confirmed
2	Return to home screen	Home screen	Meal recommendations reflect updated caloric targets	Recommendations refreshed with correct targets	Android 11	Pass	Scoring engine recalculates on next load

Test Case TC-10: Backend Restart Resilience – Reliability Test Case

Table 28:TC-10-Backend Restart Resilience

Test Scenario ID	TC-10	**Test Case ID**	TC-10A
Test Case Description	Backend Restart Resilience – Reliability Test Case	**Test Priority**	Medium
Pre-Requisite	Chef creates a new meal during a backend cold-start window on Render free tier	**Post-Requisite**	Meal written to Firestore immediately; tagging completes after service resumes

Test Execution Steps:

S.No	**Action**	**Inputs**	**Expected Output**	**Actual Output**	**Test Browser**	**Test**	**Test Comments**

						Result	
1	Chef submits new meal during backend cold-start window	New meal form submission	Meal written to Firestore immediately	Meal written to Firestore	Android 11	Pass with latency	Firestore write is client-side; unaffected by backend state
2	Wait for backend service to resume (30 to 60 seconds typical)	Backend Render service	Tagging job picked up and completed after restart	Tags applied within 90 seconds of service restart	Android 11	Pass with latency	Bull queue persists jobs through restart

6.4 Evaluation of Recommendation Quality

For recommendation quality, I constructed three customer profiles chosen to represent meaningfully different health contexts.

- Profile A: BMI 28 (overweight), weight loss goal, happy mood — expected to favour low-calorie, light-preparation meals with upbeat tag associations.
- Profile B: BMI 20 (normal), maintenance goal, unpleasant mood — expected to favour comfort-oriented meals while respecting maintenance caloric target.
- Profile C: BMI 17 (underweight), weight gain goal, neutral mood — expected to favour high-calorie, protein-rich meals.

In each test case I reviewed the recommendation output manually against the full meal catalogue. The scoring algorithm gave correct rankings in all three cases — low-calorie meals came out on top for the overweight user with a weight loss goal, high-calorie meals ranked first for the underweight user, and comfort food bias appeared correctly for the unpleasant mood profile.

6.5 Known Limitations in Testing

There are real gaps in the test coverage that I should be clear about. There are no integration tests covering the full order flow end to end, from cart write through to FCM delivery. The recommendation scoring and caloric target calculation functions have no automated tests. The SER model accuracy was

never evaluated against a proper labelled dataset — I assessed it informally through repeated manual voice trials, which is clearly not rigorous.

6.6 Application Programming Interface (API) Testing

The backend exposes HTTP endpoints used by the Flutter client and also connects to Firebase APIs and the external Gemini API. Testing these interfaces was necessary to confirm the application communicates correctly with both internal and external services.

API testing used Postman. I ran four types of checks:

Functionality Testing — verifying that each endpoint returns correct data for valid requests.
Error Handling — confirming that malformed or missing parameters produce appropriate error responses rather than unhandled exceptions.
Performance Testing — checking that endpoint response times remain acceptable under single-user load conditions.
Security Testing — verifying that the Firebase ID token validation middleware correctly rejects unauthenticated requests to protected backend routes.

6.6.1 Backend Endpoint Testing

The main backend endpoint is notifyChef, which receives an order payload and fires an FCM notification to the relevant chef. Testing confirmed: Firebase ID token validation works, missing required fields produce appropriate errors, and FCM payloads result in actual notifications on the chef's test device.

6.6.2 Firebase SDK Integration Testing

Firebase Auth, Firestore, Realtime Database, and Cloud Messaging were tested through the Flutter client using both the emulator suite and a live Firebase project. Read and write operations were checked against the expected data shapes. For the Realtime listener, I verified that status changes written by the chef appeared on the customer's screen within the 30-second requirement.

6.6.3 Gemini API Integration Testing

Gemini API testing involved submitting meal payloads with known ingredients and checking that the returned tags contained the expected allergen and category labels. I also tested quota exhaustion by manually setting the quota guard counter to its limit and confirming that the system fell back to rule-based tagging cleanly without errors.

6.7 Conclusion

Testing showed that the core FuelUp features work as intended. The recommendation output is contextually appropriate across different user profiles, and all the major functional flows pass. The most significant weakness is the automated test coverage gaps — those would need to be addressed before any real deployment.

Chapter 7: Conclusion

This chapter summarises what FuelUp achieved, reflects on what the development process actually taught me, and identifies the most important things to fix or extend in future work.

7.1 Contributions

The main thing FuelUp demonstrates is that a mood-aware, nutrition-conscious meal ordering app is achievable within a final-year project scope. The project produced a working Flutter client, Firebase data infrastructure, an on-device SER pipeline, a hybrid tagging backend, a multi-factor recommendation engine, and a documented analysis of production risks.

- A fully functional cross-platform Flutter application supporting customer meal discovery, mood-aware recommendation, cart management, order placement, and order tracking, alongside a chef meal management and order fulfilment interface.
- An on-device speech emotion recognition pipeline implemented in Dart using TensorFlow Lite and native MFCC extraction, requiring no cloud inference dependency for mood detection.
- A hybrid meal tagging backend combining deterministic lexical analysis with quota-governed Gemini AI enhancement, demonstrating practical reliability-through-fallback architecture at the backend service layer.
- A multi-factor meal recommendation engine implementing dynamic weight adjustment based on customer health context and mood state, producing recommendations that vary meaningfully across distinct user profiles.
- A documented analysis of security, access control, and reliability risks inherent in the current implementation, providing an honest self-assessment alongside the technical achievements.

7.2 Reflections

A few things became clear during development that I did not anticipate at the start. Moving backend processing from Firebase Functions to a Render-hosted Node.js service gave much better control over the execution environment and dependencies — but the trade-off was Render's cold-start problem, which creates noticeable notification latency after idle periods on the free tier.

The hybrid fallback design proved its worth repeatedly. The Gemini API quota hit its limit several times during development, sometimes for extended periods. Because the rule-based tagging path always runs first, the system kept functioning throughout every quota failure. Without that fallback the app would have been effectively broken for days at a time.

Credential management was a lesson I learned the hard way. A service account key was accidentally committed to the repository early in the project. It was rotated immediately, but the incident made it clear how easy this mistake is to make and why secret scanning tools need to be set up from day one, not treated as something to add later.

Schema consistency was another problem that cost time. The meal data ended up using both 'recipe' and 'recipes' as field names in different parts of the system, which caused lookup failures in certain code paths. That kind of inconsistency is easy to introduce when building quickly and genuinely annoying to track down and fix later.

7.3 Future Work

Several specific improvements are planned or recommended for future development.

7.3.1 Security Hardening

The leaked service account credentials need to be fully purged from the repository history, not just rotated. Secret scanning should be added to the CI/CD pipeline to prevent this happening again. The Firestore security rules are currently broader than they should be and would allow unintended access patterns at any real scale — tightening them is a priority.

7.3.2 Architecture Stabilisation

The Render free tier cold-start problem is a real issue for notification latency. Moving to a paid tier or finding a way to keep the service warm would fix it. Moving listener state to a durable store like Redis or Firestore would also address the related problem of missed change events when the service restarts.

7.3.3 Mood Pipeline Enhancement

The detectMoodWithFallback utility needs to be the default call path in the VoiceMoodButton widget. There are currently some UI flows that call the SER pipeline directly and bypass the fallback entirely, which is an oversight that should be fixed.

7.3.4 Data Quality and Schema Normalisation

The recipe/recipes naming inconsistency needs a proper migration script that standardises the field name across Firestore, the Realtime Database paths, and all the Flutter model classes. Left unaddressed it will cause increasingly serious data integrity issues at any meaningful scale.

7.3.5 Quality Engineering

Automated test coverage needs to be expanded significantly to cover the recommendation scoring logic, caloric target calculations, and the full order-to-notification integration path. Running the SER model against a properly labelled evaluation set would also give much more reliable accuracy figures than the informal manual trials I did.

7.3.6 Feature Expansion

Future features I would prioritise: a meal rating and review system would generate collaborative signal data that could improve recommendations significantly over time. Multi-language support would broaden accessibility. More granular health profile options beyond BMI and dietary flags would make the recommendations more medically relevant.

FuelUp is a project I am genuinely proud of technically. It brings together real-time mobile development, on-device ML, a cloud-backed marketplace, and an

event-driven backend service into a single working system. The mood-aware recommendation engine is the core differentiator, and it works reliably because of the hybrid fallback design. The security and reliability issues I have documented are not excuses — they are honest assessments of what needs to be fixed before this could become a real product.

References

[1] Ricci, F., Rokach, L., and Shapira, B. (2015). Recommender Systems Handbook (2nd ed.). Springer, New York. doi: 10.1007/978-1-4899-7637-6

[2] Burke, R. (2002). 'Hybrid Recommender Systems: Survey and Experiments.' User Modeling and User-Adapted Interaction, 12(4), pp. 331–370.

[3] Mifflin, M.D., St Jeor, S.T., Hill, L.A., Scott, B.J., Daugherty, S.A., and Koh, Y.O. (1990). A new predictive equation for resting energy expenditure in healthy individuals. American Journal of Clinical Nutrition, 51(2), 241–247. doi: 10.1093/ajcn/51.2.241

[4] Schuller, B., Batliner, A., Steidl, S., and Seppi, D. (2011). Recognising realistic emotions and affect in speech: State of the art and lessons learnt from the first challenge. Speech Communication, 53(9-10), 1062–1087. doi: 10.1016/j.specom.2011.01.011

[5] Davis, S.B. and Mermelstein, P. (1980). Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. IEEE Transactions on Acoustics, Speech, and Signal Processing, 28(4), 357–366. doi: 10.1109/TASSP.1980.1163420

[6] Google. (2024). TensorFlow Lite Guide. Available at: https://www.tensorflow.org/lite/guide [Accessed: April 2025].

[7] Google. (2024). Firebase Documentation. Available at: https://firebase.google.com/docs [Accessed: April 2025].

[8] Flutter Team. (2024). Flutter Documentation. Available at: https://docs.flutter.dev [Accessed: April 2025].

[9] Google. (2024). Gemini API Documentation. Available at: https://ai.google.dev/docs [Accessed: April 2025].

[10] World Health Organisation. (2000). Obesity: Preventing and Managing the Global Epidemic. WHO Technical Report Series 894. Geneva: WHO.

[11] Render. (2024). Render Documentation. Available at: https://render.com/docs [Accessed: April 2025].

Appendices

Appendix A: Mood-to-Tag Mapping Taxonomy

The table below lists the meal tag categories mapped to each mood label used in the FuelUp recommendation engine. These mappings were designed based on common associations between food types and emotional states, informed by the nutrition literature reviewed in Chapter 2.

Table 29:Mood Label to Meal Tag Mapping Taxonomy

Mood Label	Compatible Meal Tag Categories
Happy	fresh, light, salad, fruit, smoothie, grilled, colourful, tropical, summery, energising
Neutral	balanced, wholesome, hearty, rice, pasta, soup, sandwich, everyday, familiar
Unpleasant	comfort, warm, creamy, cheesy, fried, sweet, chocolate, indulgent, rich, soothing
Surprise	exotic, fusion, spicy, unusual, adventurous, international, bold, innovative

Appendix B: Rule-Based Tagging Heuristic Categories

The rule-based meal tagging engine detects the following categories through keyword matching against meal names, descriptions, and ingredient lists. Allergen detection covers: nuts (peanuts, almonds, cashews, walnuts, pecans, pistachios), dairy (milk, cream, cheese, butter, yoghurt, whey), gluten (wheat, barley, rye, bread, pasta, flour), eggs, shellfish (shrimp, crab, lobster, prawn), soy (soya, tofu, edamame), fish, and sesame (tahini, sesame oil, sesame seed). Dietary label detection identifies: vegetarian (meals with no meat or seafood ingredients), vegan (meals with no animal-derived ingredients), gluten-free (no gluten-containing ingredients detected), high-protein (significant presence of protein-dense ingredients like chicken, eggs, lentils, or legumes), and low-calorie (calorie value below a defined threshold).