3B1A Inc.

Mashallah Eats
Software Requirements Specification
Phase II Report

Ahmed Huossein, Maaz Qasim, Mahir Shahriar, Hurera Ranjha

Version 2.0

Revision History

Date	Version	Description	Author
11/11/2025	2.0	Created initial report	Ahmed, Hurera, Maaz, Mahir
11/12/2025	2.1	Added psuedocode	Ahmed, Hurera, Maaz, Mahir
11/15/2025	2.2	Added ER Diagram/Class Collaboration	Ahmed, Hurera, Maaz, Mahir
11/17/2025	2.3	Added Use Case diagrams and documentation	Ahmed, Hurera, Maaz, Mahir

Table of Contents

Table of Contents

Software Requirements Specification

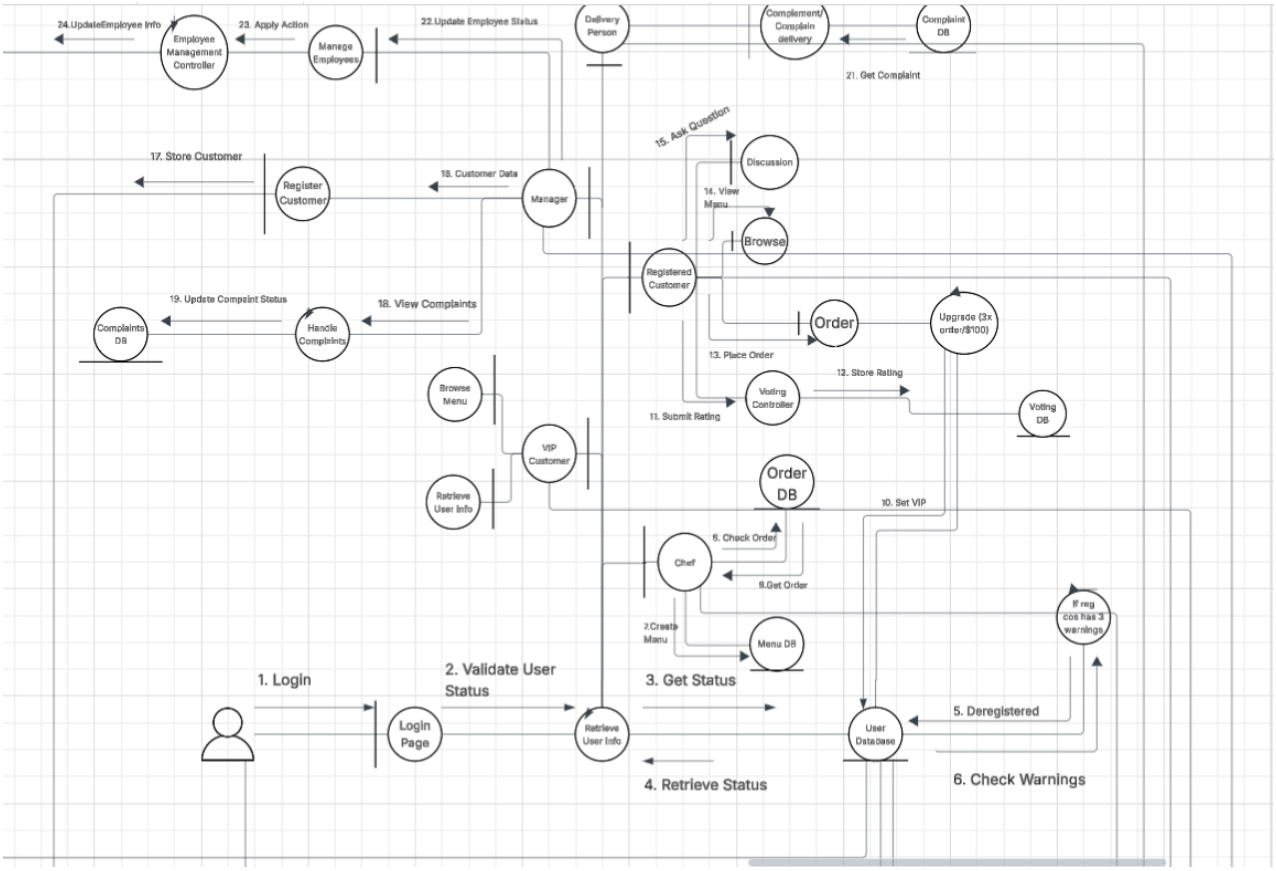
1. Introduction

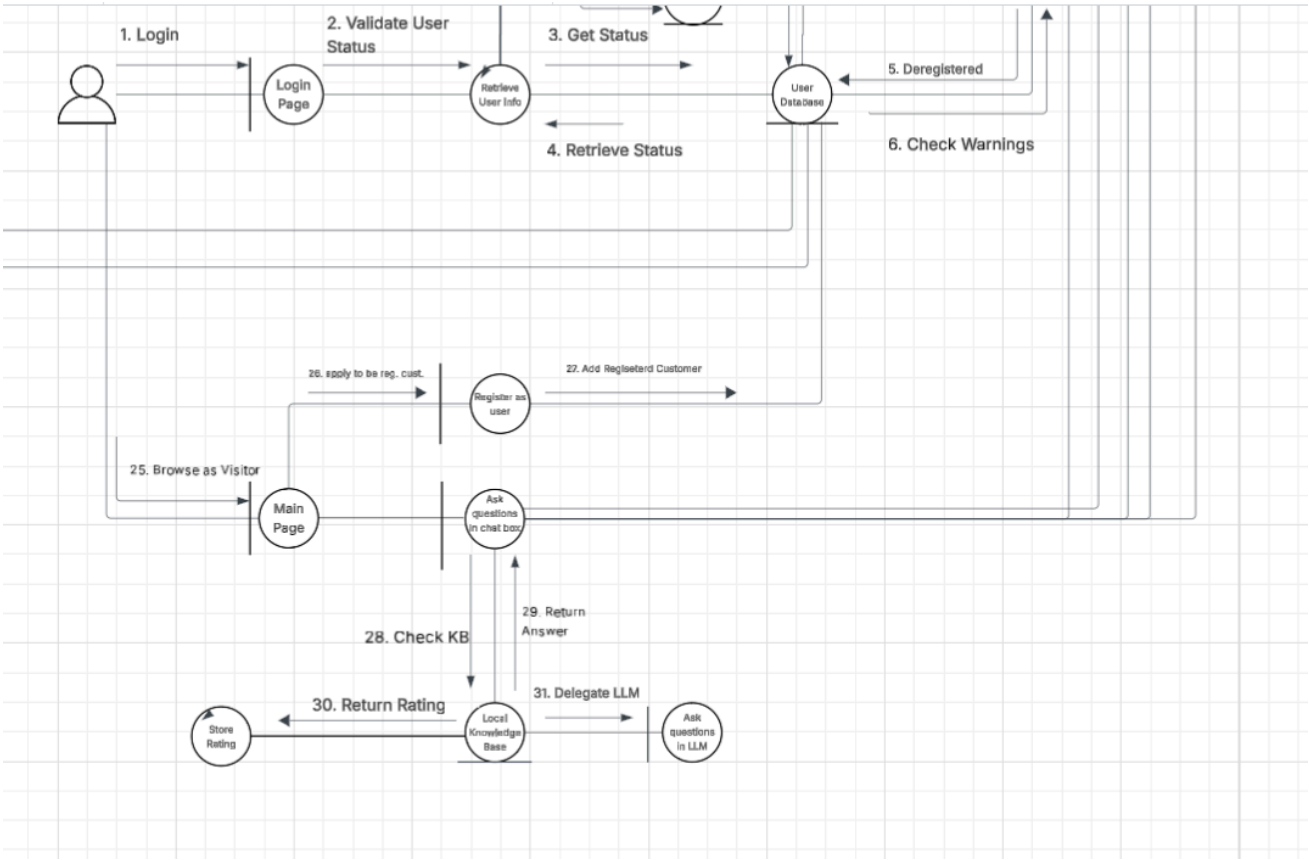
MashallahEats is an AI-powered online restaurant ordering and delivery platform inspired by services like Uber Eats and DoorDash. The system enables users to browse restaurant menus, place orders, and have their meals delivered directly to their homes with ease and reliability.

The primary objective of MashallahEats is to deliver an engaging, user-friendly web experience. Visitors can explore a dynamic homepage showcasing popular and highly rated dishes or access personalized pages tailored to their unique food preferences. The platform emphasizes user interaction through customer reviews, feedback, and an integrated AI assistant that remains available at all times to answer questions and guide users throughout the site.

MashallahEats is designed as a multi-user platform capable of supporting at least 100 simultaneous users who can interact through discussion forums and place concurrent orders. The system ensures a secure environment where all financial data remains private and protected according to high security standards. Additionally, the AI chatbot is incorporated to provide reliable, helpful responses that reflect the company’s mission.

2. Introduction: Collaboration Class Diagram





3. All use cases

Use Cases: Place Order

Actors: Registered Customer/VIP, Chef, Manager (for assignment), Delivery Person
Preconditions: User is authenticated; menu is up-to-date; user has positive balance or valid payment method.

Main Flow:

1. The customer browses the menu and selects items.
2. The system calculates total and applicable VIP discounts.
3. Customer confirms order and delivery/pickup option.
4. If delivery, the order enters the delivery bidding pool.
5. Manager assigns delivery per bidding policy.
6. Chef prepares dish; status updates are emitted.
7. Customers can track and receive notifications.

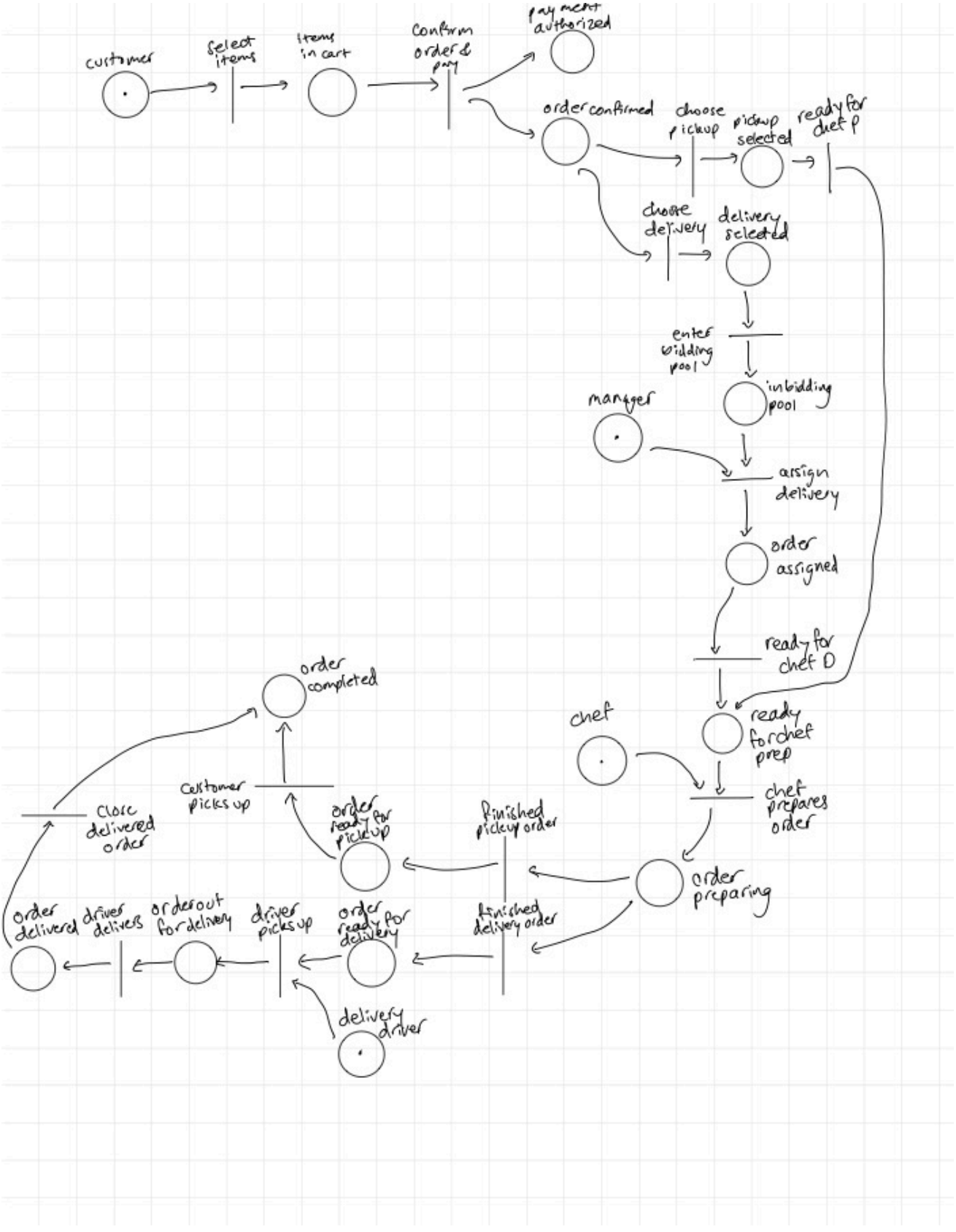
Postconditions: Order is created; payment authorized; status set to Preparing or Assigned.

Functional Requirements:

- Support in-store, pickup, and delivery flows.
- Apply VIP discounts and free delivery rules.
- Emit order status updates (Preparing, Out for Delivery, Delivered).

Non-Functional Requirements:

- Latency for status updates < 2s in most cases.
- Secure checkout; PCI-compliant gateway integration.


customer
select items
items in cart
confirm order & pay
payment authorized
order confirmed
choose pickup
pickup selected
ready for chef p
choose delivery
delivery selected
enter bidding pool
in bidding pool
manager
assign delivery
order assigned
ready for chef D
order completed
chef
ready for chef prep
chef prepares order
customer picks up
close delivered order
order ready for pickup
finished pickup order
order preparing
order delivered
driver delivers
order out for delivery
driver picks up
order ready for delivery
finished delivery order
delivery driver

Use Case: Process Payment

Actors: Registered Customer/VIP, Payment Gateway

Preconditions: Order initiated; basket total computed.

Main Flow:

1. The system checks the deposit balance.
2. If insufficient balance, reject and add a warning.
3. If sufficient, authorize and capture via gateway.

Postconditions: Transaction recorded and receipt issued.

Functional Requirements:

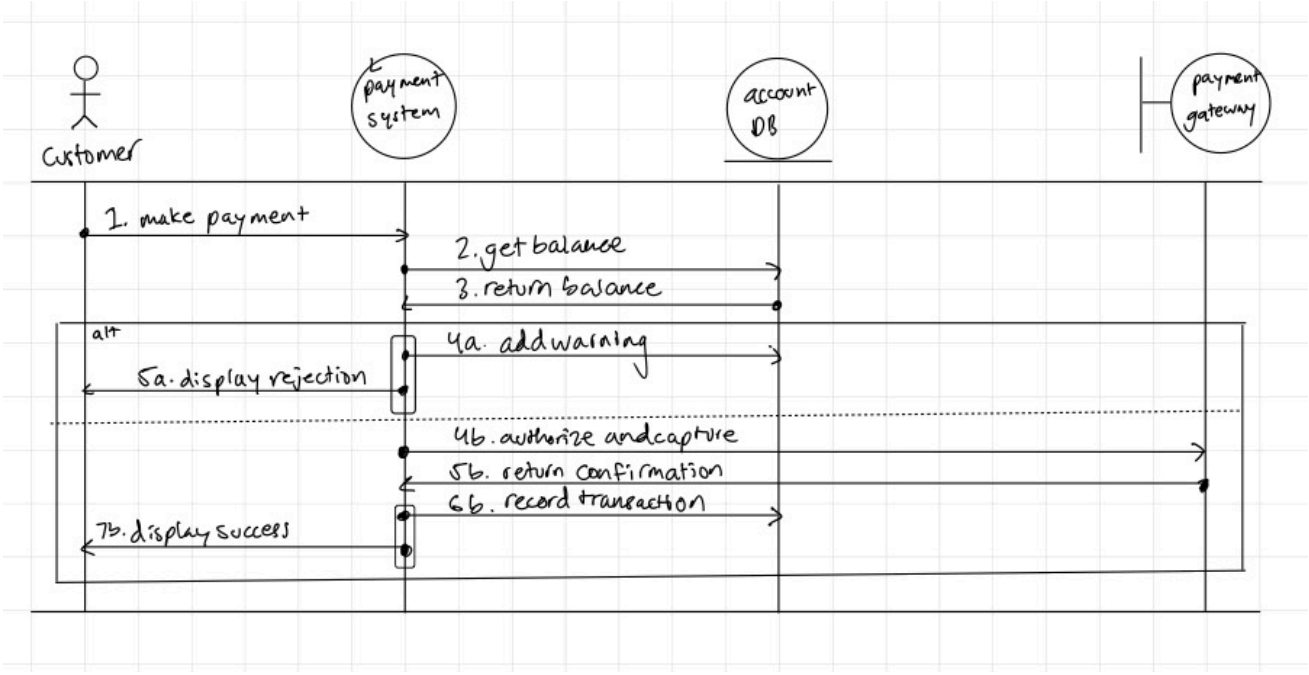
• Deposit model with auto‑freeze if price exceeds balance.

• Refund path for canceled orders.

Non‑Functional Requirements:

• Payment data encrypted in transit and at rest.

• Idempotent charge operations (prevent duplicate billing)



Use Case: Delivery Bidding & Assignment

Actors: Delivery Person, Manager

Preconditions: New delivery task available.

Main Flow:

1. Delivery persons submit bids (price/ETA).
2. System ranks bids; manager reviews and chooses the winner.
3. If a higher price is chosen, the manager must write a justification note.
4. Assignment pushed to the chosen delivery person.

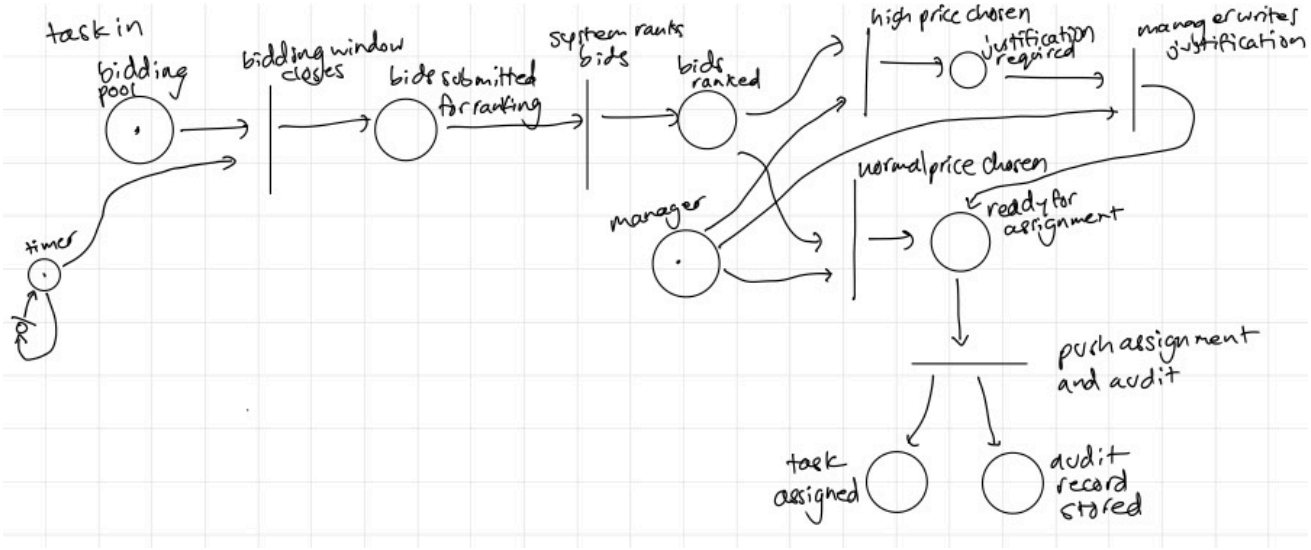
Postconditions: Delivery task assigned; audit record stored.

Functional Requirements:

• Bid submission

Non-Functional Requirements:

• Real-time push within seconds; durable audit logs.



Use Case: Track Order

Actors: Customer, Delivery Person

Preconditions: Order accepted and in progress.

Main Flow:

1. Show order status and ETA.
2. Delivery updates (progress)
3. Customer receives notifications on status changes.

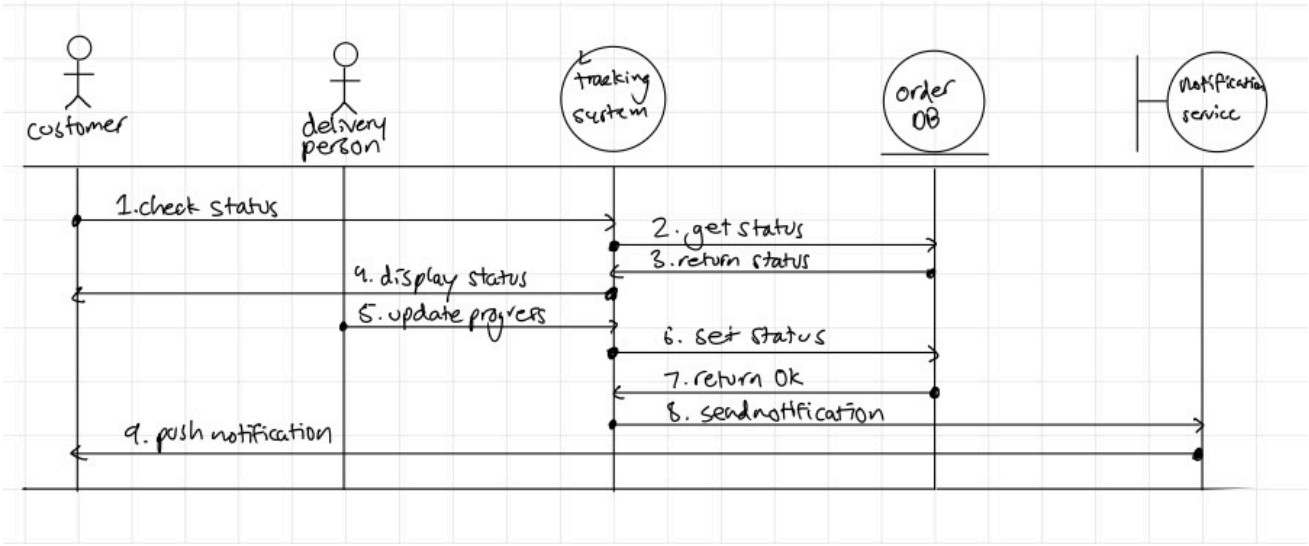
Postconditions: Order marked Delivered when completed.

Functional Requirements:

• Real-time status feed from delivery app/panel.

Non-Functional Requirements:

• High availability for tracking channel.



Use Case: Feedback & Complaints

Actors: Customer, Delivery Person, Manager

Preconditions: Order completed or forum interaction occurred.

Main Flow:

1. Customer rates dish and delivery separately (1–5).
2. Customer or Delivery person files a complaint/compliment.
3. The accused party may dispute.
4. Manager reviews and decides; warnings updated; compliments can cancel complaints.

Postconditions: Decision applied; parties notified; warnings surfaced on login.

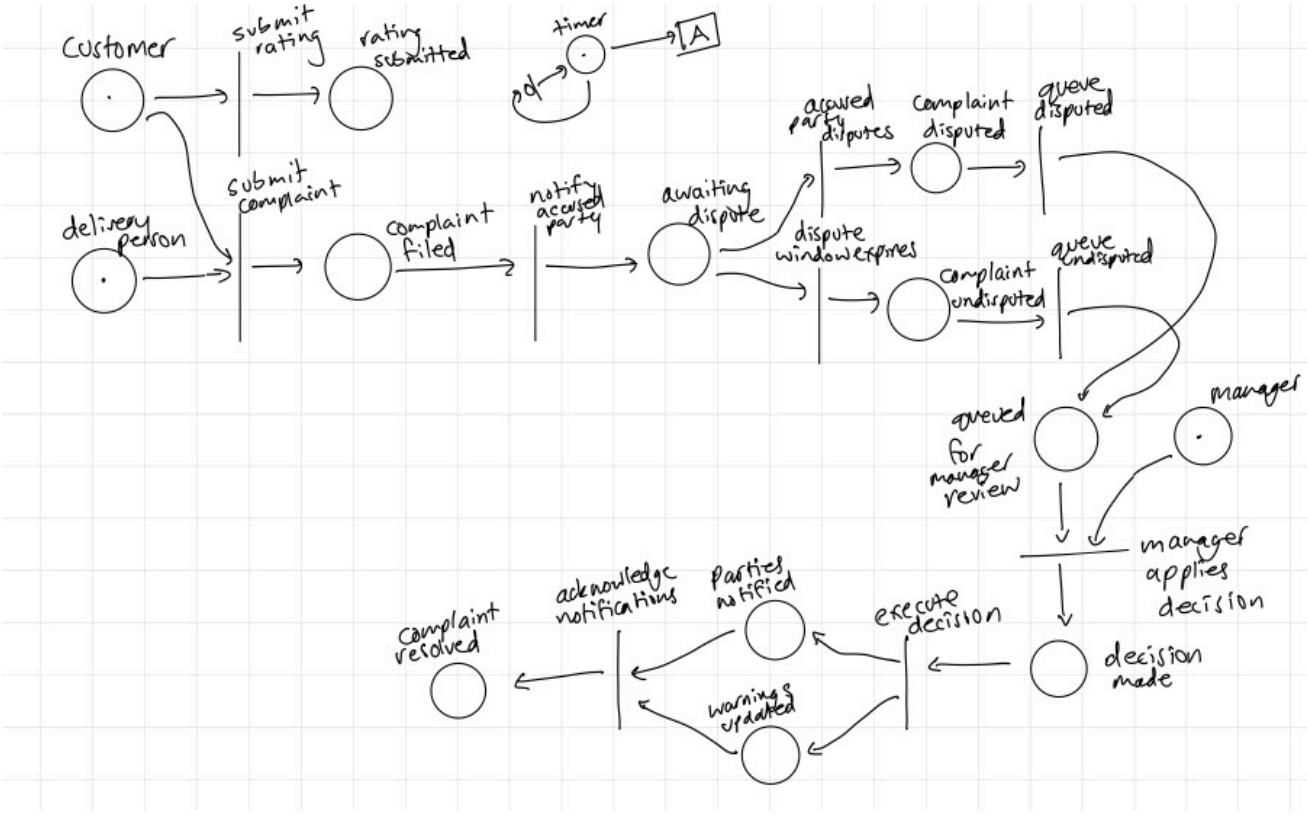
Functional Requirements:

• Store separate ratings for food and delivery.

• Complaint lifecycle with dispute and decision states.

Non-Functional Requirements:

• Moderated forums; audit trails; notification reliability.



Use Case: Account Management & VIP

Actors: Customer, Manager

Preconditions: Customer is registered; thresholds computed.

Main Flow:

1. Customer views status, warnings, and VIP eligibility.
2. VIP upgrade/downgrade per policy.

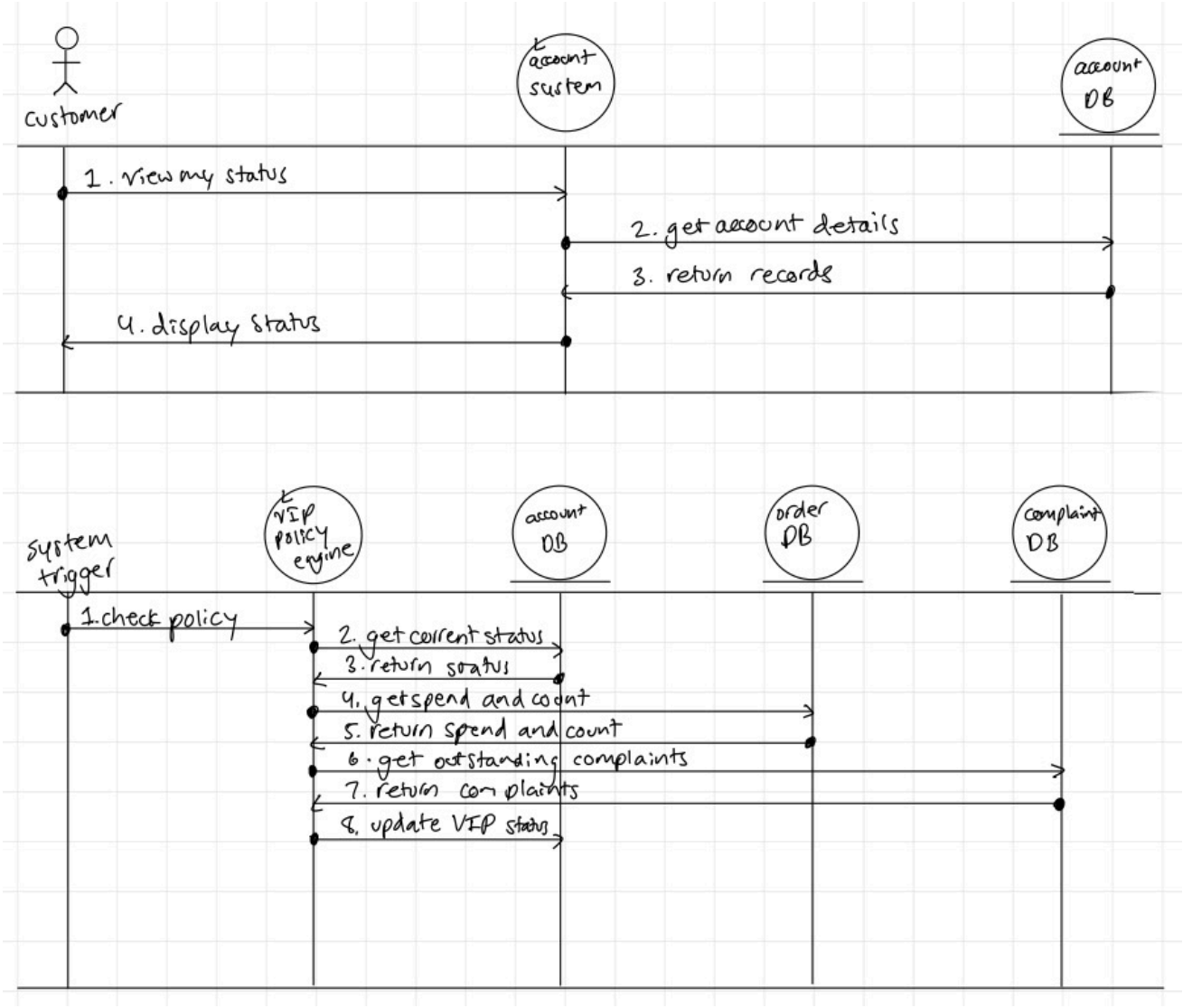
Postconditions: VIP status updated; benefits applied.

Functional Requirements:

• Policy: VIP after >$100 spend or 3 orders without outstanding complaints; 5% discount and 1 free delivery per 3 orders.

Non-Functional Requirements:

• consistent presentation



Use Case: Deregistration & Blacklist

Actors: Manager

Preconditions: User has accumulated warnings or requested quit.

Main Flow:

1. Manager clears deposit and closes account.
2. If kicked, add to blacklist; prevent future registration.

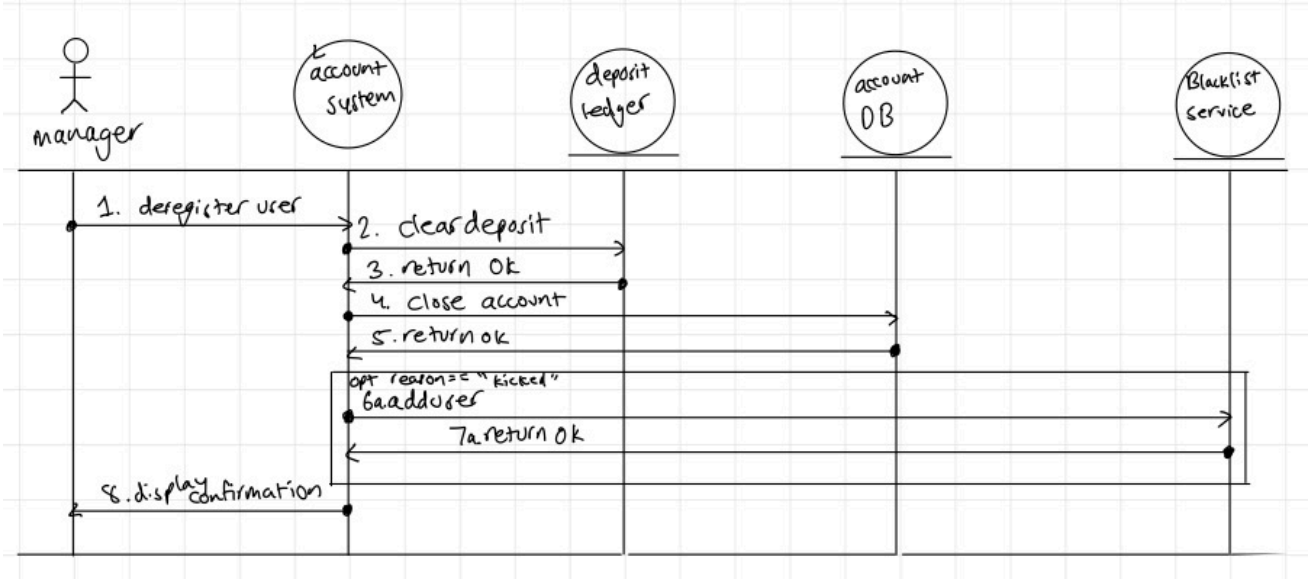
Postconditions: Account closed; funds settled; blacklist enforced.

Functional Requirements:

• Refund rules and irreversible blacklist flag.

Non-Functional Requirements:

• Financial reconciliation integrity; access control.

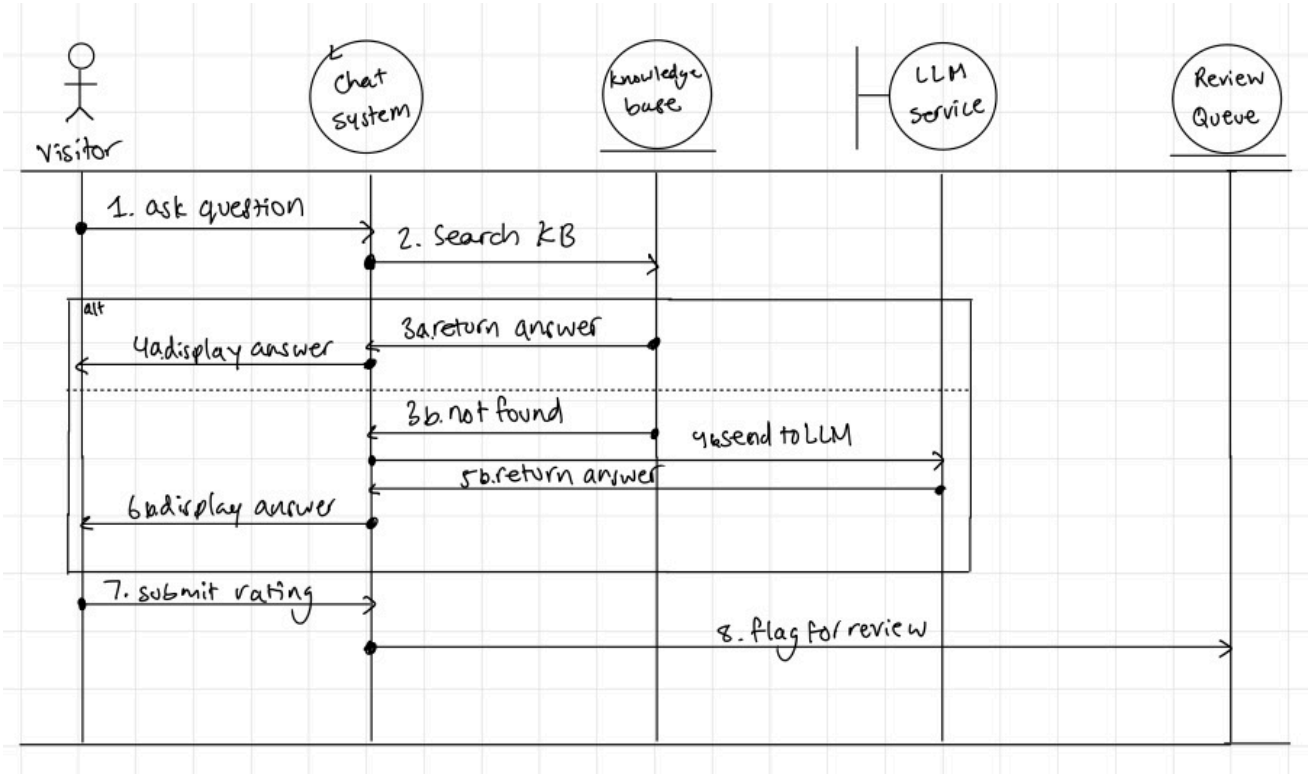


Use Case: AI Customer Service (KB + LLM Fallback)

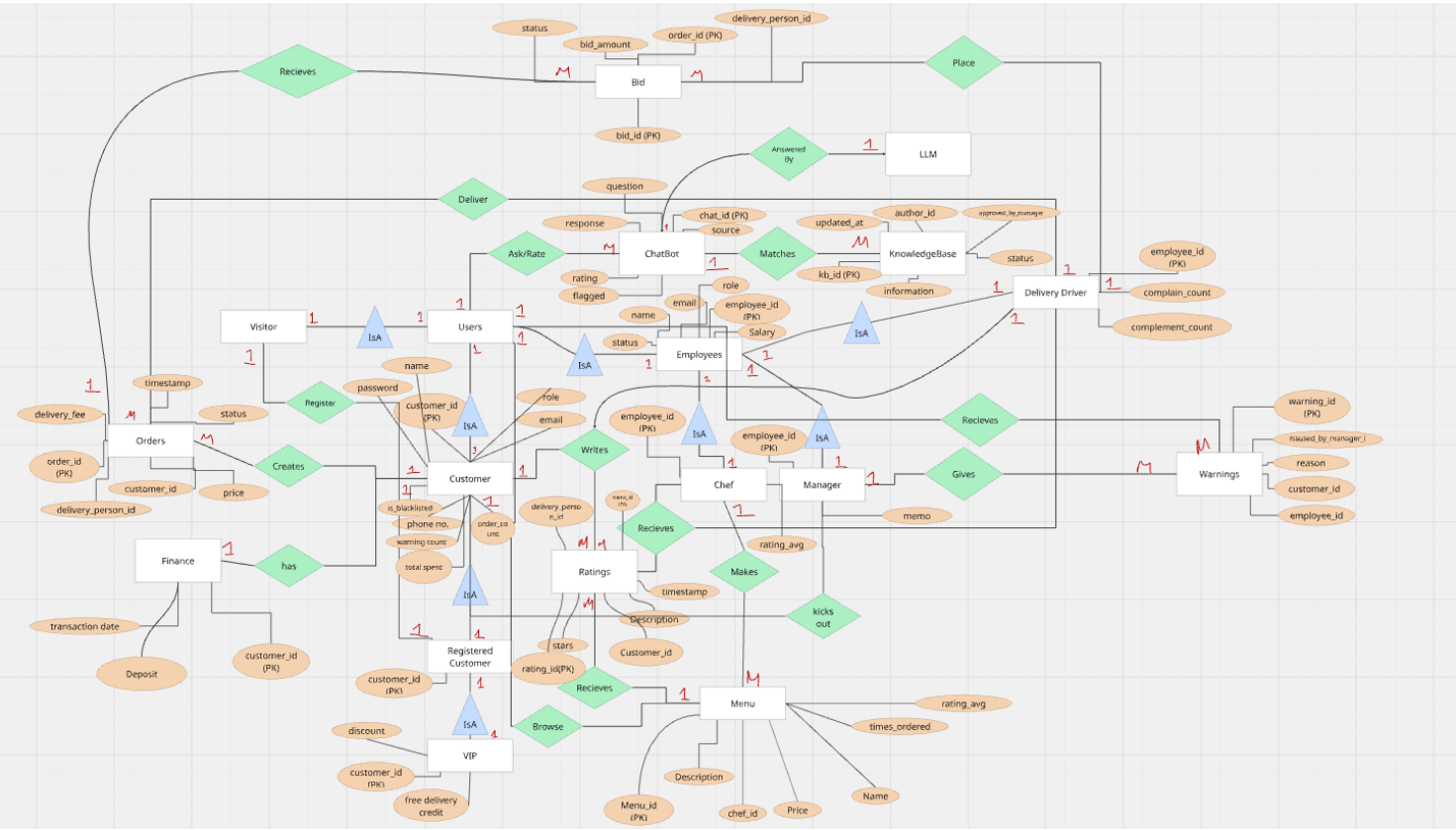
Actors: Visitor, Customer, Manager, AI Assistant
Preconditions: Chat channel available; KB populated.

Main Flow:
1. User asks a question in chat.
2. System searches local KB first.
3. If not found, delegate to LLM and return answer.
4. User rates answer; rating=0 flags item for manager review.

Postconditions: Answer delivered; feedback captured; flagged items queued.
Functional Requirements:
• KB CRUD; LLM connector; flagging workflow.
Non-Functional Requirements:
• Content safety filters; response time targets.



4. E-R diagram for the entire system



Link for better view: https://ibb.co/fzgkS4c6

5. Detailed Design

Pseudocode for every method:

login(username, password)	Input: Username, Password
	
	Output: Success or Error
	
	If (username, password) exists in UserDatabase
	
	Return “success”
	
	Else
	
	Return “Error”

getUserInfo(userID)	Input: userID Output: User Role If userID exists in UserDatabase: return userID.role Else: return "Error: User not found"
registerUser(userData)	**Input:** first_name, last_name, birthday, email, password, deposit_amount **Output:** “Success” or “Error” If email exists in UserDatabase: return "Error: Email already exists" If deposit_amount == 0: return "Error: Must put deposit" Else: User_Data = { all input data } Add User_Data to UserDatabase return "Success"

deregisterUser(userID)	Input: userID Output: “Success” or “Error” If userID exists in UserDatabase: UserDatabase.remove(User_Id) return "Success" Else: return "Error: User not found"
checkWarnings(userID)	Input: userID Output: warning count or Error If userID exists in UserDatabase: Return UserDatabase[userID].warnings Else Return “Error: User not found”
updateWarnings(userID, action)	Input: userID Output: “Success” or “Error” If userID exists in UserDatabase: UserDatabase[userID].warnings++ Return “Success” Else Return “Error: User not found”
browseMenu(userID)	Input: userID Output: Menu If userID not found in UserDatabase: return getVisitorMenu() If getUserInfo(userID) == "Registered": return getRegisteredMenu(userID) If getUserInfo(userID) == "VIP": return getVIPMenu(userID) Else: return "Error: Invalid role"

getRecommendedItems(userID)	Input: userID Output: Recommended items or Error If userID found in UserDatabase: Return UserDatabase[userID].mostOrderedItems Else Return “Error: User not found”
getVisitorMenu()	Input: no input Output: Visitor Menu Return PopularDishes + HighestRatedDishes
getRegisteredMenu(userID)	Input: userID Output: Registered Menu If userID not found in UserDatabase: return "Error: User not found" return StandardMenu + getRecommendedItems(userID)
getVIPMenu(userID)	Input: userID Output: VIP Menu If userID not found in UserDatabase: return "Error: User not found" return StandardMenu + VIPDishes + getRecommendedItems(userID)
placeOrder(userID, orderData)	Input: userID, orderData Output: “Success” or “Error” If userID not found in UserDatabase: return "Error: User not found" If orderData is invalid: return "Error: Invalid order" Else: Add orderData to UserDatabase[userID].orders return "Success"
checkDeposit(userID, amount)	Input: userID, amount Output: “Enough” or “Error”

	checkDeposit(userID, amount): If userID not found in UserDatabase: return "Error: User not found" deposit = UserDatabase[userID].deposit If deposit >= amount: return "Enough" Else: return "Not enough"
deductDeposit(userID, amount)	
recordOrderHistory(userID, orderID)	Input: userID, orderID Output: “Success” or “Error” If userID not found in UserDatabase: return "Error: User not found" Append orderID to UserDatabase[userID].orderHistory return "Success"
rateFood(userID, orderID, rating)	Input: userID, orderID, rating Output: “Success” or “Error” If userID not found in UserDatabase: return "Error: User not found" If orderID not in UserDatabase[userID].orders: return "Error: Invalid order" Add rating to FoodRatingsDB[orderID].ratings return "Success"
rateDelivery(userID, deliveryPersonID, rating)	Input: userID, deliveryPersonID, rating Output: “Success” or “Error” If userID not found in UserDatabase: return "Error: User not found" If deliveryPersonID not in DeliveryPersonDatabase: return "Error: Delivery person not found" Add rating to DeliveryPersonDatabase[deliveryPersonID].ratings return "Success"

fileComplaint(userID, targetID, type)	Input: userID, targetID, type Output: “Success” or “Error” If userID not found in UserDatabase: return "Error: User not found" Complaint = { userID, targetID, type } Add Complaint to ComplaintDatabase return "Success"
fileCompliment(userID, targetID, type)	Input: userID, targetID, type Output: “Success” or “Error” If userID not found in UserDatabase: return "Error: User not found" Compliment = { userID, targetID, type } Add Compliment to ComplimentDatabase return "Success"

disputeComplaint(userID, complaintID)	Input: userID, complaintID
	Output: “Success” or “Error”
	
	If complaintID not found in ComplaintDatabase:
	  return "Error"
	
	If ComplaintDatabase[complaintID].targetID != userID:
	  return "Error"
	
	ComplaintDatabase[complaintID].disputed = True
	return "Success"

handleComplaint(complaintID)	Input: complaintID Output: “Success” or “Error” If complaintID not found in ComplaintDatabase: return "Error" ComplaintDatabase[complaintID].handled = True return "Success"
submitDeliveryBid(deliveryPersonID, price)	Input: deliveryPersonID, price Output: “Success” or “Error”

	If deliveryPersonID not found in DeliveryPersonDatabase: return "Error" Bid = { deliveryPersonID, price } Add Bid to DeliveryBidDatabase return "Success"

assignDelivery(orderID)	Input: orderID
	Output: deliveryPersonID or “Error”
	
	If orderID not found in OrderDatabase:
	  return "Error"
	
	Find the bid with the lowest price in DeliveryBidDatabase
	
	If no bids exist:
	  return "Error"
	
	Assign deliveryPersonID to orderID
	return deliveryPersonID

assignDelivery(orderID)	Input: orderID
	Output: “Success” or “Error”
	
	If orderID not found in OrderDatabase:
	  return "Error"
	
	If DeliveryBidDatabase is empty:
	  return "Error"
	
	lowestBid = MIN(DeliveryBidDatabase.price)
	
	OrderDatabase[orderID].deliveryPersonID =
	lowestBid.deliveryPersonID
	return "Success"

recordDeliveryRating(deliveryID, rating)	Input: deliveryID, rating Output: “Success” or “Error” If deliveryID not found in DeliveryDatabase: return "Error" Add rating to DeliveryDatabase[deliveryID].ratings return "Success"
updateMenu(chefID, dishInfo)	Input: chefID, dishInfo Output: “Success” or “Error”

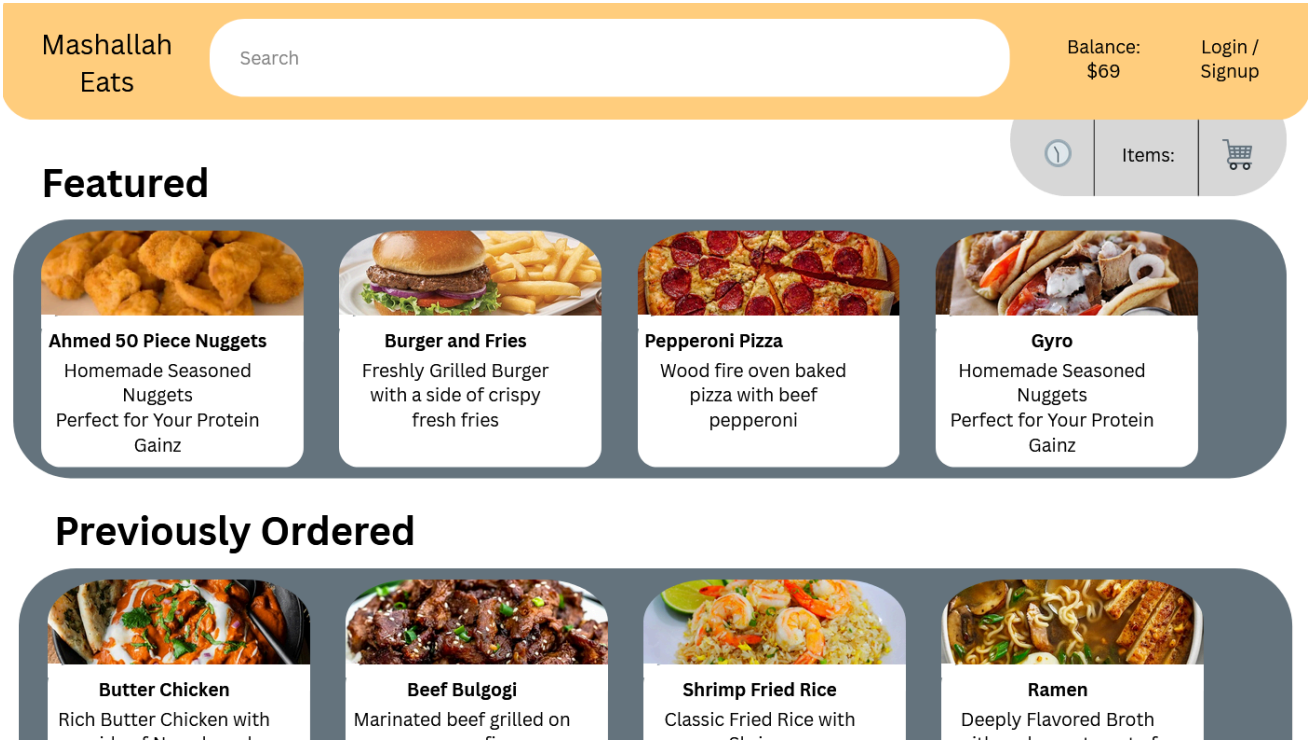
	If chefID not found in ChefDatabase: return "Error" Add dishInfo to MenuDatabase return "Success"
prepareDish(orderID)	Input: orderID Output: “Success” or “Error” If orderID not found in OrderDatabase: return "Error" OrderDatabase[orderID].status = "Prepared" return "Success"
updateChefRating(chefID, rating)	Input: chefID, rating Output: “Success” or “Error” If chefID not found in ChefDatabase: return "Error" Add rating to ChefDatabase[chefID].ratings return "Success"
addChefComplaint(chefID)	Input: chefID Output: “Success” or “Error” If chefID not found in ChefDatabase: return "Error" ChefDatabase[chefID].warnings++ return "Success"
hireEmployee(employeeData)	Input: employeeData Output: “Success” or “Error” Add employeeData to EmployeeDatabase return "Success"
fireEmployee(employeeID)	Input: employeeID Output: “Success” or “Error” If employeeID not found in EmployeeDatabase: return "Error"

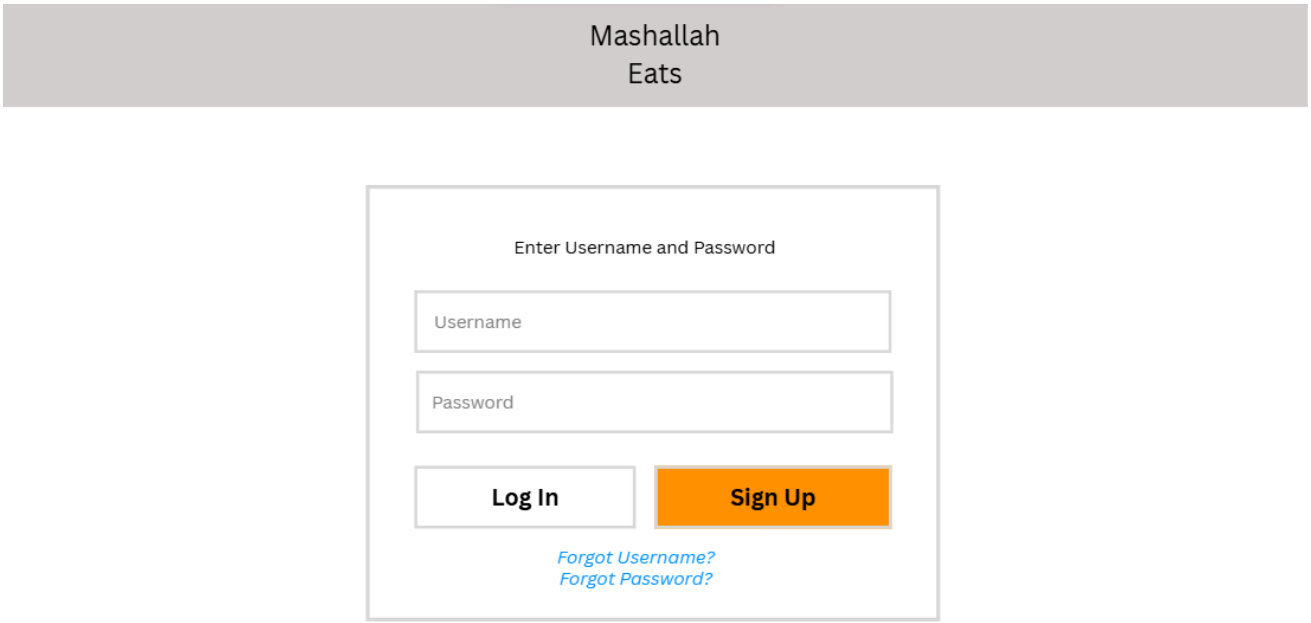
	Remove employeeID from EmployeeDatabase return "Success"
adjustSalary(employeeID, amount)	Input: employeeID, amount Output: “Success” or “Error” If employeeID not found in EmployeeDatabase: return "Error" EmployeeDatabase[employeeID].salary += amount return "Success"
blacklistUser(userID)	Input: userID Output: “Success” or “Error” If userID not found in UserDatabase: return "Error" UserDatabase[userID].status = "Blacklisted" return "Success"
askQuestion(userID, query)	Input: userID, query Output: answer or “Error” If userID not found in UserDatabase: return "Error" Return forwardToLLM(query)
searchLocalKnowledge(query)	Input: query Output: results or “Error” results = LocalKnowledgeDatabase.search(query) return results
forwardToLLM(query)	Input: query Output: answer answer = LLM.generate(query) return answer
rateResponse(userID,	Input: userID, answerID, rating

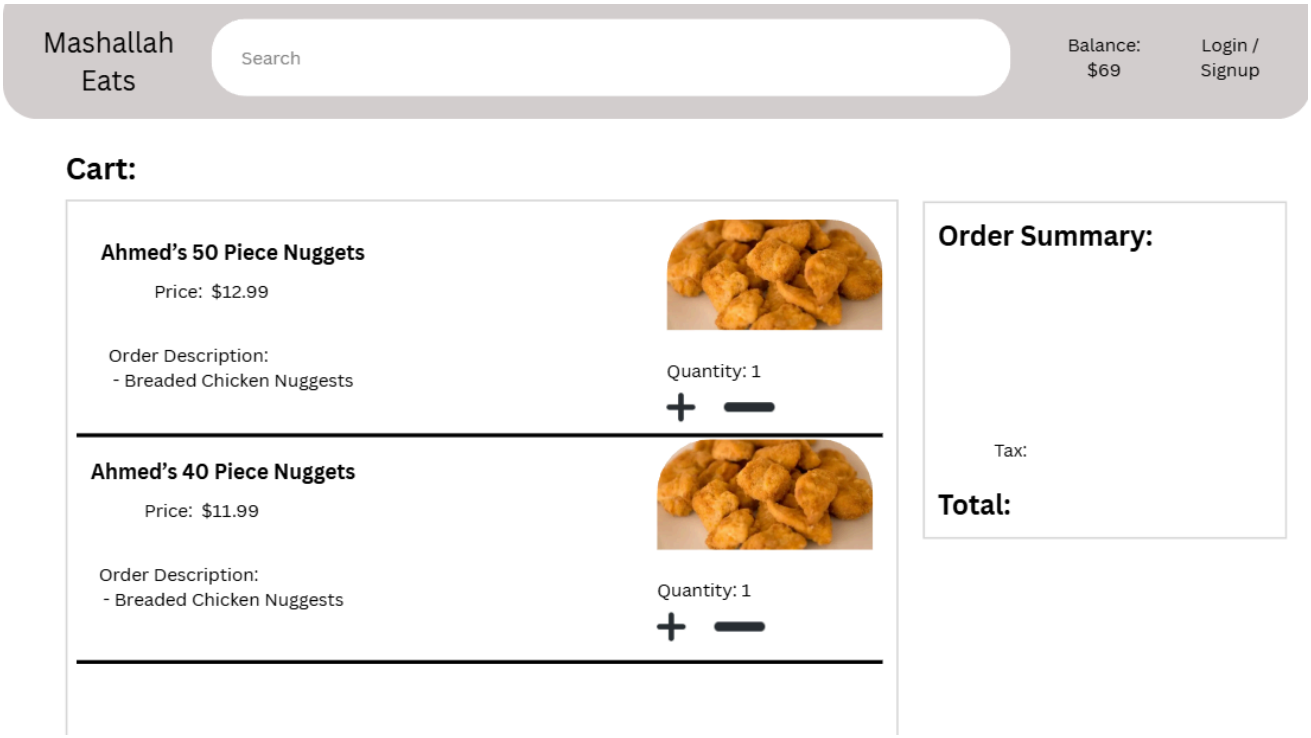
answerID, rating)	Output: “Success” or “Error” If userID not found: return "Error" Add rating to AnswerDatabase[answerID].ratings return "Success"
flagBadAnswer(answerID)	Input: answerID Output: “Success” or “Error” If answerID not found: return "Error" AnswerDatabase[answerID].flagged = True return "Success"
dbGetUser(userID)	Input: userID Output: userData or “Error” If userID not found: return "Error" return UserDatabase[userID]

6. System screens:

demonstrate major GUI screens of the system and a sample prototype of one functionality of your own choice.







https://www.figma.com/proto/LbjZshtOhe0NDUEHLLeCUX/Untitled?node-id=0-1&t=D234xSNkBVgjxjLo-1

The prototype demonstrates a basic user navigation flow: Login → Home → Cart → Home. After logging in, the user is directed to the Home page. From there, selecting the cart icon in the top-right corner opens the Cart page. The user can then return to the Home page using the back navigation.

7. Memos of group meetings and possible concerns of team work

Date	Attendance	Discussion
11/11/2025	Ahmed, Maaz, Mahir, Hurera	Made the document for phase 2. Assigned individual tasks
11/12/2025	Ahmed, Maaz	Wrote use-case scenarios and checked numbering.
11/13/2025	Ahmed, Hurera	Set up backend dependencies.
11/14/2025	Maaz, Mahir, Hurera	Set up React Frontend and connected Supabase DB
11/15/2025	Ahmed, Maaz, Mahir,	Added pseudocode for login, register, and menu functions.
11/16/2025	Hurera	Fixed DaisyUI component library and compatibility with Tailwind
11/17/2025	Ahmed, Maaz, Mahir, Hurera	Finished pseudocode for delivery, bidding, and complaints.
11/18/2025	Ahmed, Maaz, Mahir, Hurera	Combined all sections, reviewed ERD, finalized SRS draft.

8. Address of the git repo:

https://github.com/HureraRanjha/Mashallah-Eats