AI Eats

Design Report
For AI-enabled Online Restaurant
Version 2.0

Revision History

Date	Version	Description	Author
11/18/25	2.0	Phase 2: Design Report	Sajid Patwary, Mohmed Bemat, Kritan Baniya, Amit Howlader

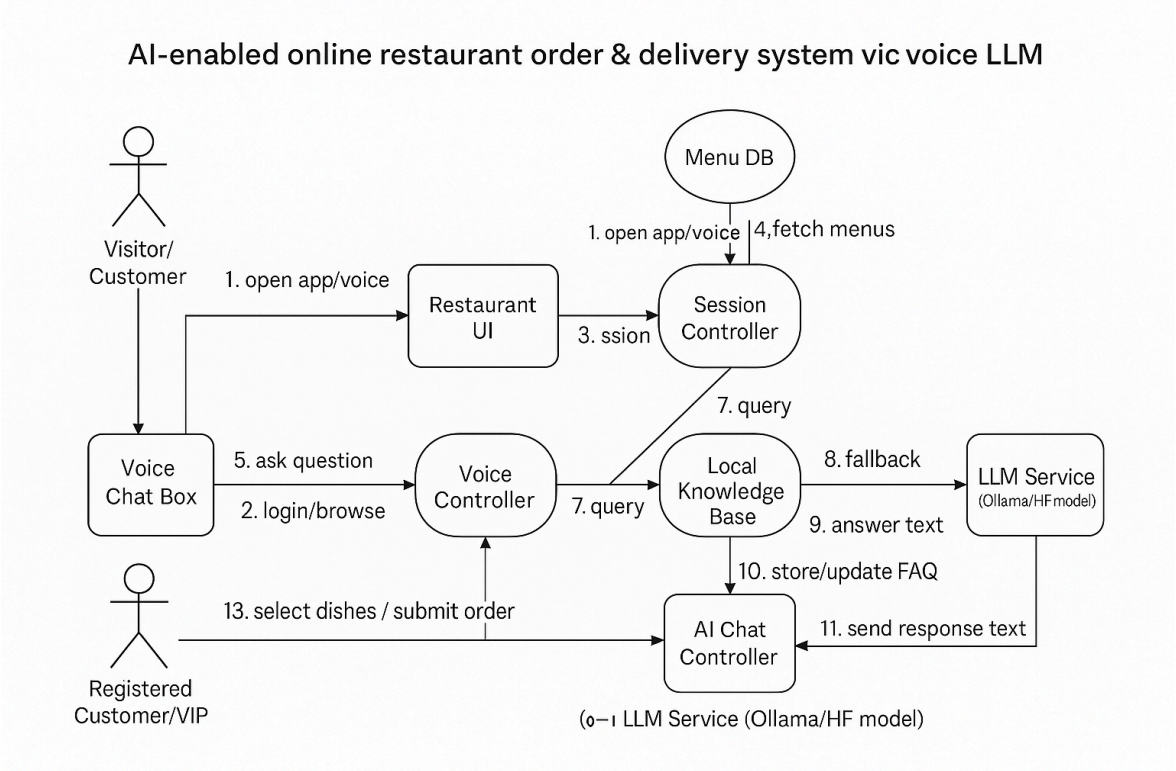
Table of Contents

1. Introduction

1.1 Overview of the System

The AI-enabled Online Restaurant Order and Delivery System is a full-stack platform that supports the entire workflow of a modern restaurant. It allows customers to browse menus, place orders, use AI-powered assistance, and receive deliveries, while giving employees and managers tools for backend operations. The system supports three user groups—Visitors, Customers, and Employees—each with different access levels. Visitors can explore the menu and ask basic questions; Customers can register, order food, manage deposits, and earn VIP perks; and Employees such as Chefs, Delivery Personnel, and Managers handle food preparation, delivery, complaints, and administrative tasks. Customers interact through a GUI with images, descriptions, and pricing, and their orders enter a delivery bidding system. A key feature is the AI chat, which uses both a local knowledge base and an external LLM to answer questions and guide users. Managers receive tools for HR functions, system oversight, and reputation management. The system also includes space for a creative feature like image-based search or route optimization. Overall, the platform unifies ordering, delivery, management tools, and AI-driven interaction into one integrated restaurant ecosystem.

1.2 Collaboration Class Diagram



2. All Use Cases

2.1 Use-Case Scenarios

Use Case ID: UC-01
Use Case Name: Order Food
Actor(s): Customer, VIP Customer

Description: A logged-in Customer selects food items from the menu, adds them to a cart, and confirms the order.

Preconditions:

1. The user is logged in as a Customer or VIP.
2. Users have a positive deposited balance in their account.

Main Flow:

1. The Customer browses or searches the menu.
2. The Customer selects a dish and adds it to their order.
3. The Customer proceeds to checkout.
4. The System displays the order summary and total price, including any applicable VIP discounts.
5. The Customer confirms the order.
6. The System verifies the Customer's balance is sufficient for the total price.
7. The System deducts the order price from the Customer's deposit.
8. The System sends the order to the Chef(s) for preparation and initiates the delivery bidding process.

Alternate Flows (Exceptions):

- **A1: Insufficient Funds**
  1. At step 6, if the total price is greater than the Customer's deposited amount, the order is rejected.
  2. The System issues one warning to the Customer for being "reckless."
  3. The System displays an error message to the Customer.
  4. The use case ends.
- **A2: Become VIP**
  1. A Customer becomes a VIP after spending more than $100 OR making 3 orders (without outstanding complaints).
- **A3: Warnings**
  1. Customers receive one automatic warning for a rejected order due to insufficient funds.
  2. A party whose complaint is dismissed as "without merit" receives one warning.
  3. Registered Customers with 3 warnings are deregistered.
  4. VIPs with 2 warnings are demoted to Registered Customer status (with warnings cleared).

Postconditions:

- The order is logged in the system.

- The Customer's account balance is updated.
- The order is in the queue for fulfillment.

Use Case ID: UC-02
Use Case Name: Cook Food
Actor(s): Chef

Description: Chef prepares dishes for an order assigned by the system after customer confirmation.

Preconditions:

1. The customer has placed and paid for an order.

2. The order has been assigned to the Chef.

Main Flow:

1. Chef logs into the system and views the "Pending Orders" list.
2. The chef selects an order to view details (dish list, special notes, delivery info).
3. Chef updates order status to "In Preparation."
4. The chef prepares the dishes.
5. Once completed, Chef marks the order as "Ready for Delivery."
6. The system notifies the Manager and available Delivery People that the order is ready.

Alternate Flows (Exceptions):

- **A1: Ingredient Shortage**
    1. Chef marks the order as "On Hold" and sends a shortage notification to the Manager.
    2. The manager decides whether to cancel or substitute the order.
- **A3: Complaints**
    1. A Chef with consistently low ratings (<2 stars) or 3 net complaints is demoted. A second demotion results in termination.
    2. A Chef with high ratings (>4 stars) or 3 net compliments receives a bonus.

Postconditions:

- Order status is "Ready for Delivery."
- Delivery bidding process begins.

Use Case ID: UC-03
Use Case Name: Deliver the Food

Actor(s): Delivery Person

Description: After being assigned a delivery, the Delivery Person picks up the order, delivers it to the customer, and records completion.

Preconditions:

1. Delivery Person has been assigned to a specific order.
2. Order status is “Ready for Delivery.”

Main Flow:

1. Delivery Person logs in and views “Assigned Deliveries.”
2. Delivery Person confirms pickup from the restaurant (status: “Out for Delivery”).
3. The system updates order tracking information for the customer.
4. Delivery Person delivers the food to the customer.
5. Customer confirms receipt via the system or delivery app.
6. Delivery Person marks the order as “Delivered.”
7. The system updates the order as completed, releases payment to the Delivery Person, and prompts both sides for feedback.

Alternate Flows (Exceptions):

- **A1: Delivery Failure**
    1. If delivery cannot be completed (e.g., wrong address), Delivery Person marks “Delivery Failed” and adds a note.
    2. System alerts the Manager for manual resolution.
- **A3: Complaints**
    1. A delivery person with consistently low ratings (<2 stars) or 3 net complaints is demoted. A second demotion results in termination.
    2. A delivery with high ratings (>4 stars) or 3 net compliments receives a bonus.

Postconditions:

- Order status = “Delivered” or “Delivery Failed.”
- Payment and reputation metrics are updated.

Use Case ID: UC-04

Use Case Name: Manage System and Personnel
Actor(s): Manager

Description: Manager oversees restaurant operations, handles disputes, manages users and employees, and ensures system stability.

Preconditions:

- The manager is authenticated in the system.

Main Flow:

1. Manager logs in and accesses the admin dashboard.
2. Manager reviews pending registrations from new Customers or Delivery People.
3. The manager approves or rejects applications.
4. The manager monitors open complaints and compliments from Customers or Employees.
5. Manager resolves disputes, marking outcomes as “valid,” “invalid,” or “escalated.”
6. Manager manages Employees (hire, fire, promote, or pay bonuses).
7. Manager reviews delivery bids and assigns orders to Delivery People.
8. If a higher bid is accepted, the manager must submit a justification memo.
9. Manager reviews AI responses flagged as “outrageous” by users and updates the Knowledge Base accordingly.

Alternate Flows:

- **A1: Invalid Application:** Manager rejects registration and provides reason.
- **A2: System Alert:** If multiple unresolved complaints exist, the system notifies the Manager to prioritize action.

Postconditions:

- System data is updated (registrations, disputes, AI flags, etc.).
- Reports are logged for accountability.

Use Case ID: UC-05
Use Case Name: View Menu
Actor(s): Visitor

Description: An unauthenticated visitor browses the restaurant’s menu and interacts with the AI chat for general inquiries.

Preconditions:

1. The visitor has access to the system interface.

Main Flow:

1. The visitor enters the AI System (possibly for the first time)

2. The visitor can browse menus.
3. The visitor can use the AI chatbot to ask questions.
4. The visitor can choose to register themselves and become a customer.

Alternate Flows:

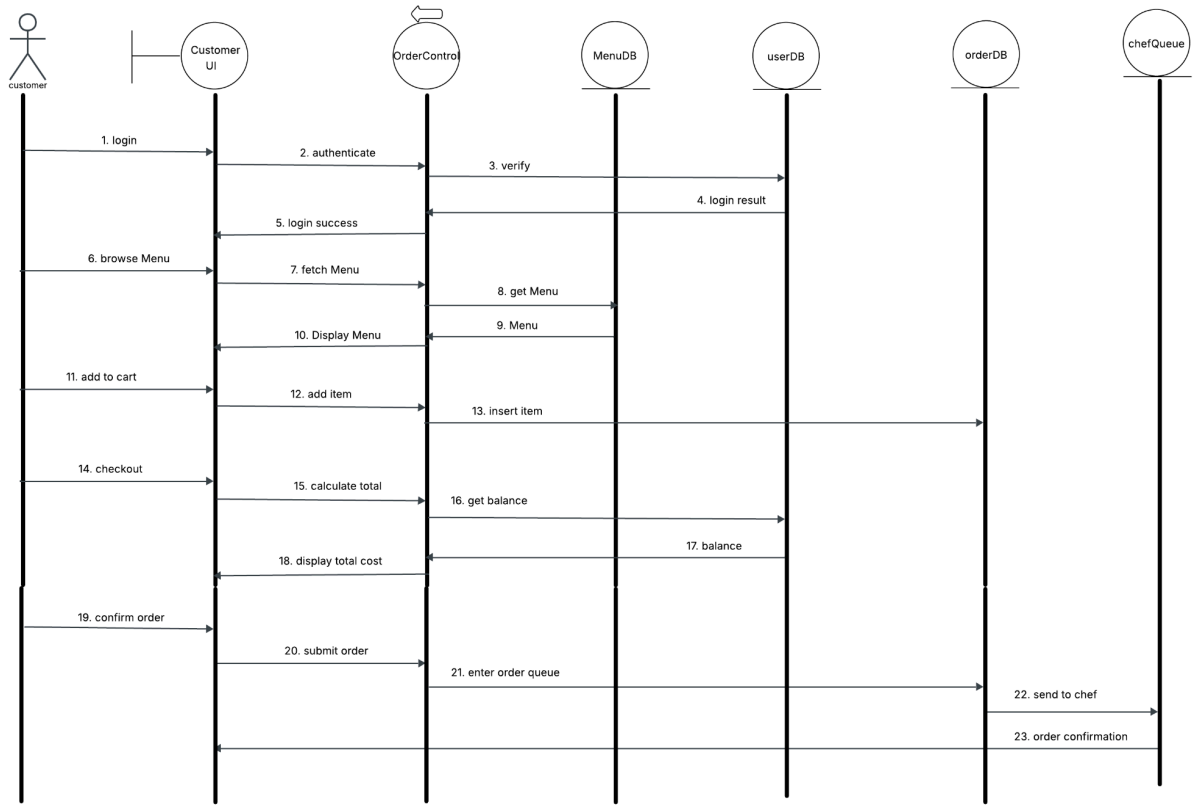
- **A1: Registration Attempt:**
    1. The visitor chooses to register for a Customer account.
    2. The system redirects to the registration form.
- **A2: AI Unavailable:**
    1. If LLM or KB service is offline, the chat box displays a message: “AI assistance temporarily unavailable.”

Postconditions:

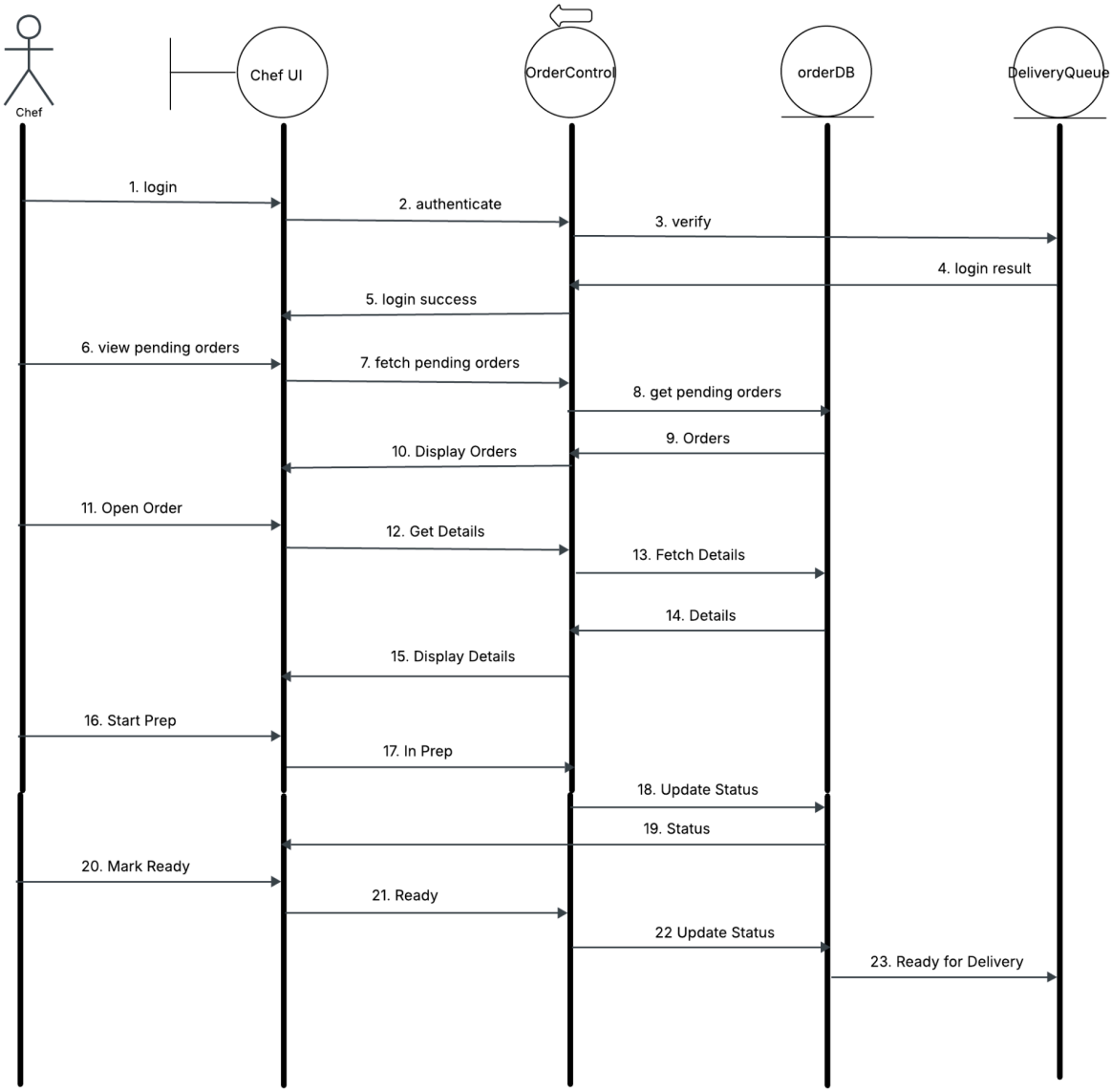
- Visitor’s actions (chat interactions and ratings) are logged.
- Optionally, visitors may proceed to registration or leave the system.

2.2 Sequence Class Diagrams

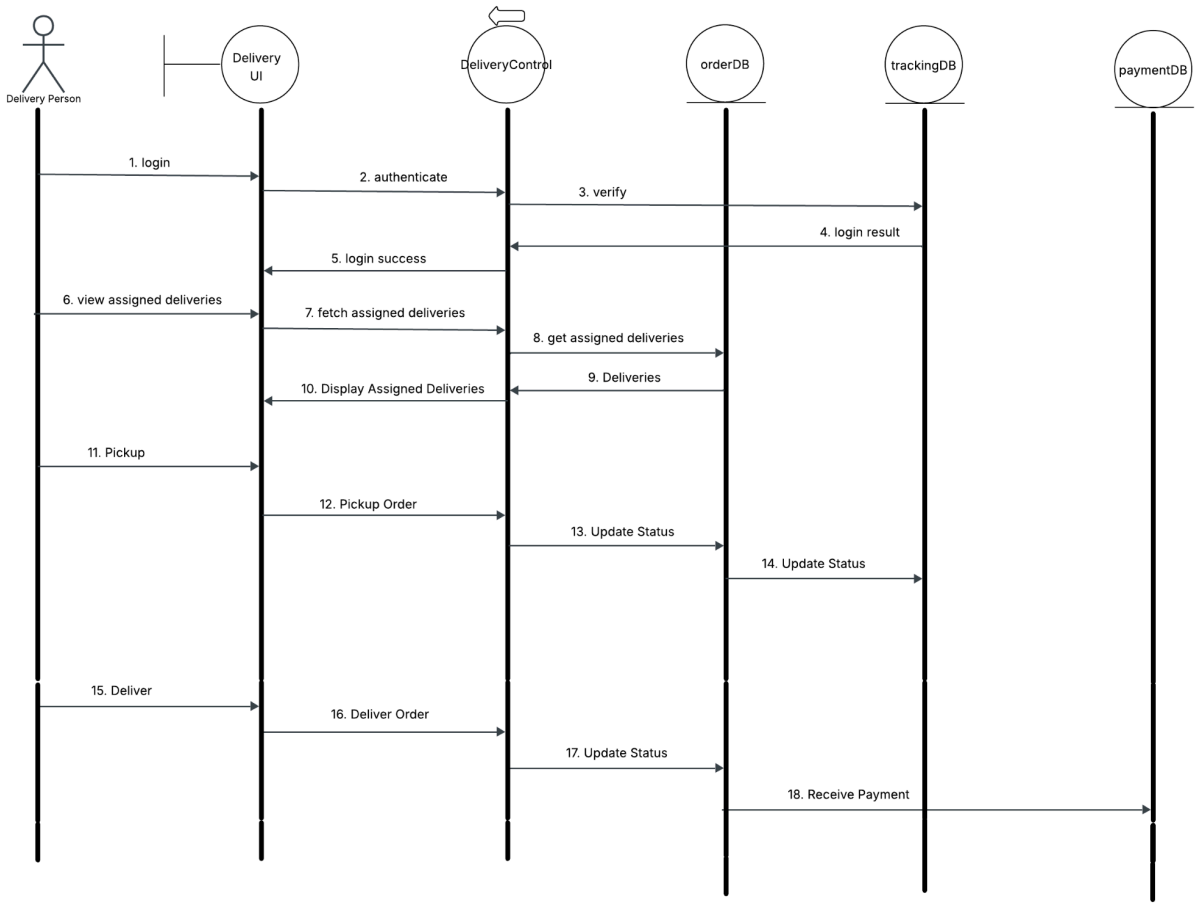
2.2.1 Order Food



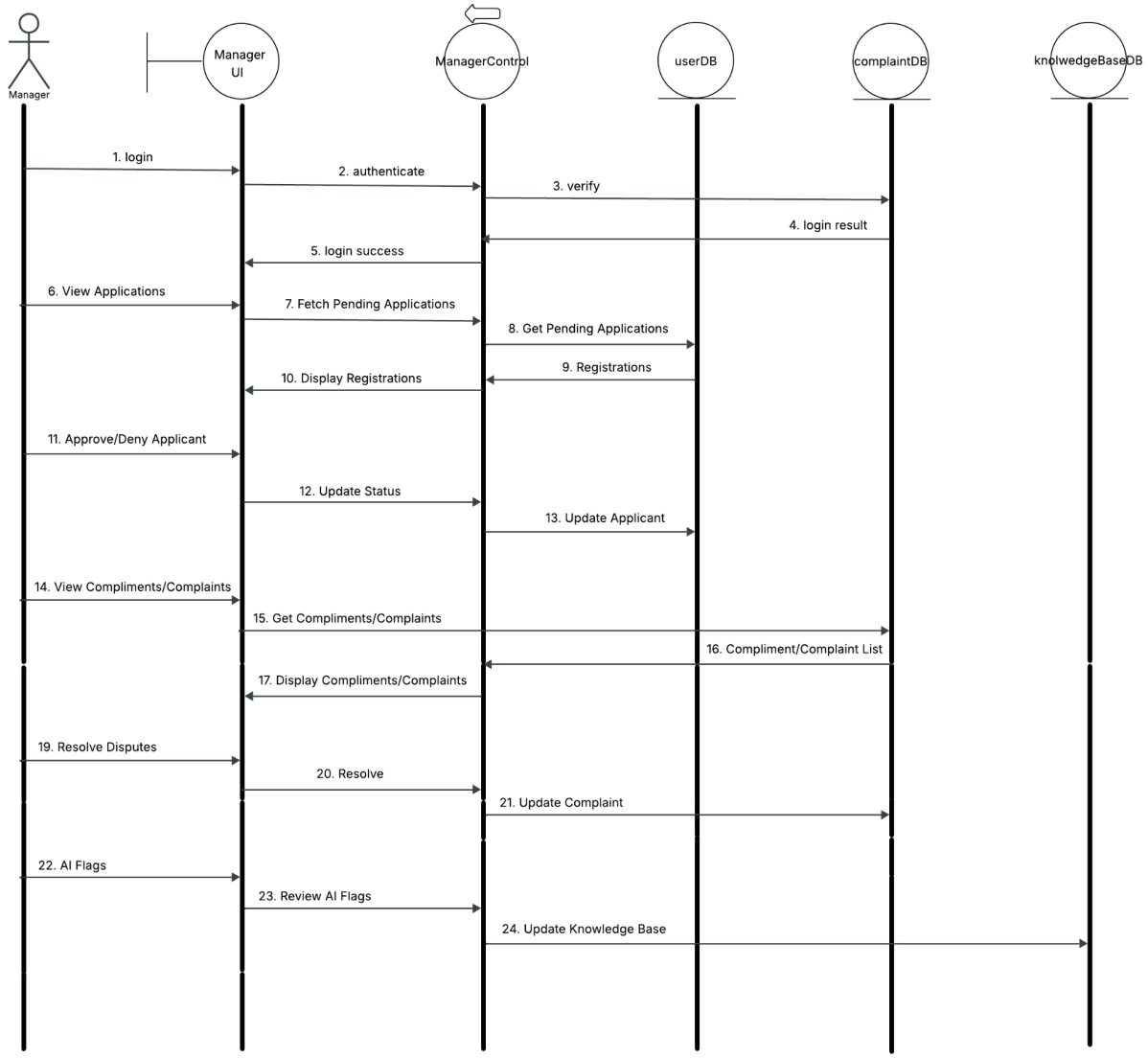
2.2.2 Cook Food



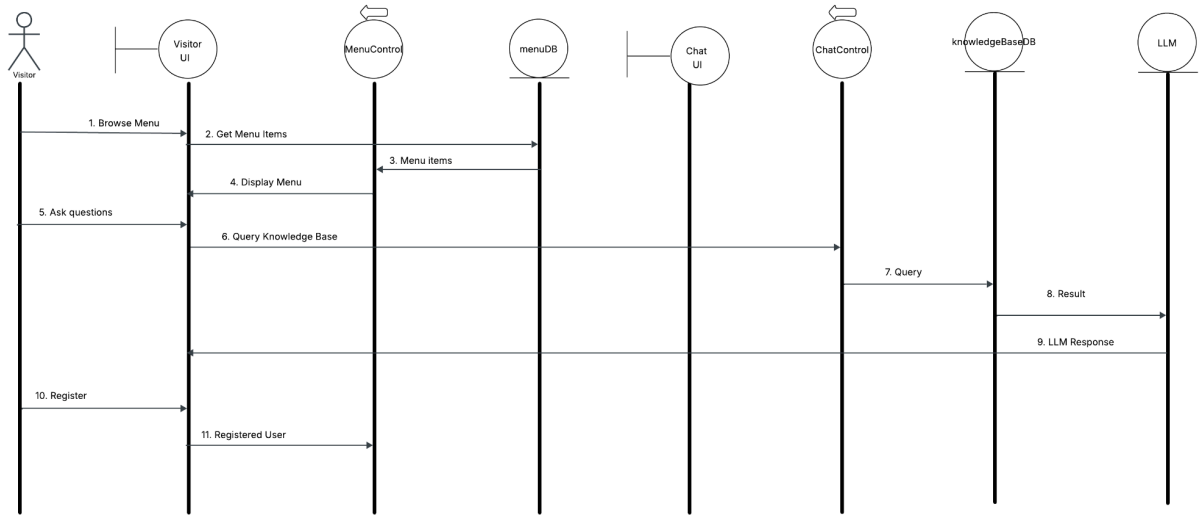
2.2.3 Deliver the Food



2.2.4 Manage System and Personnel

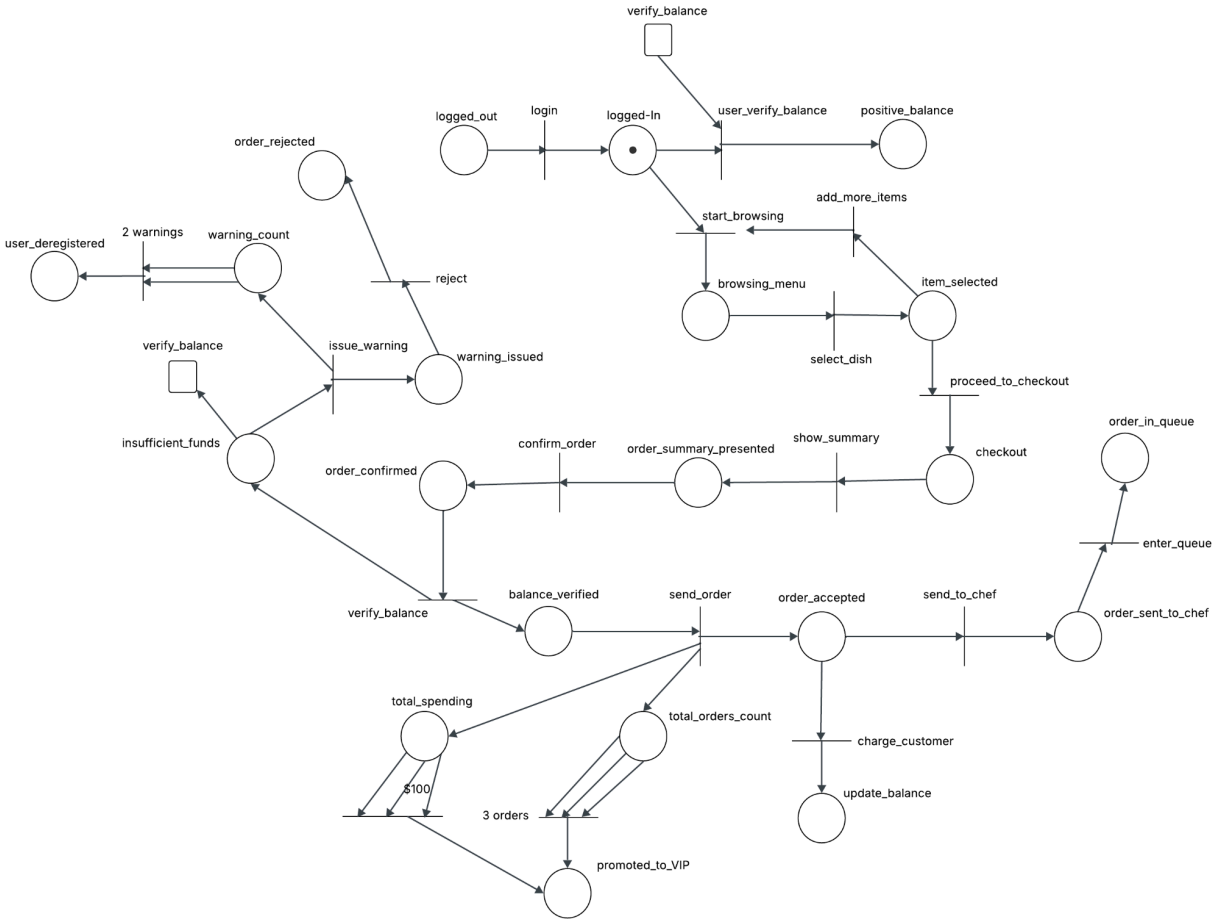


2.2.5 View Menu

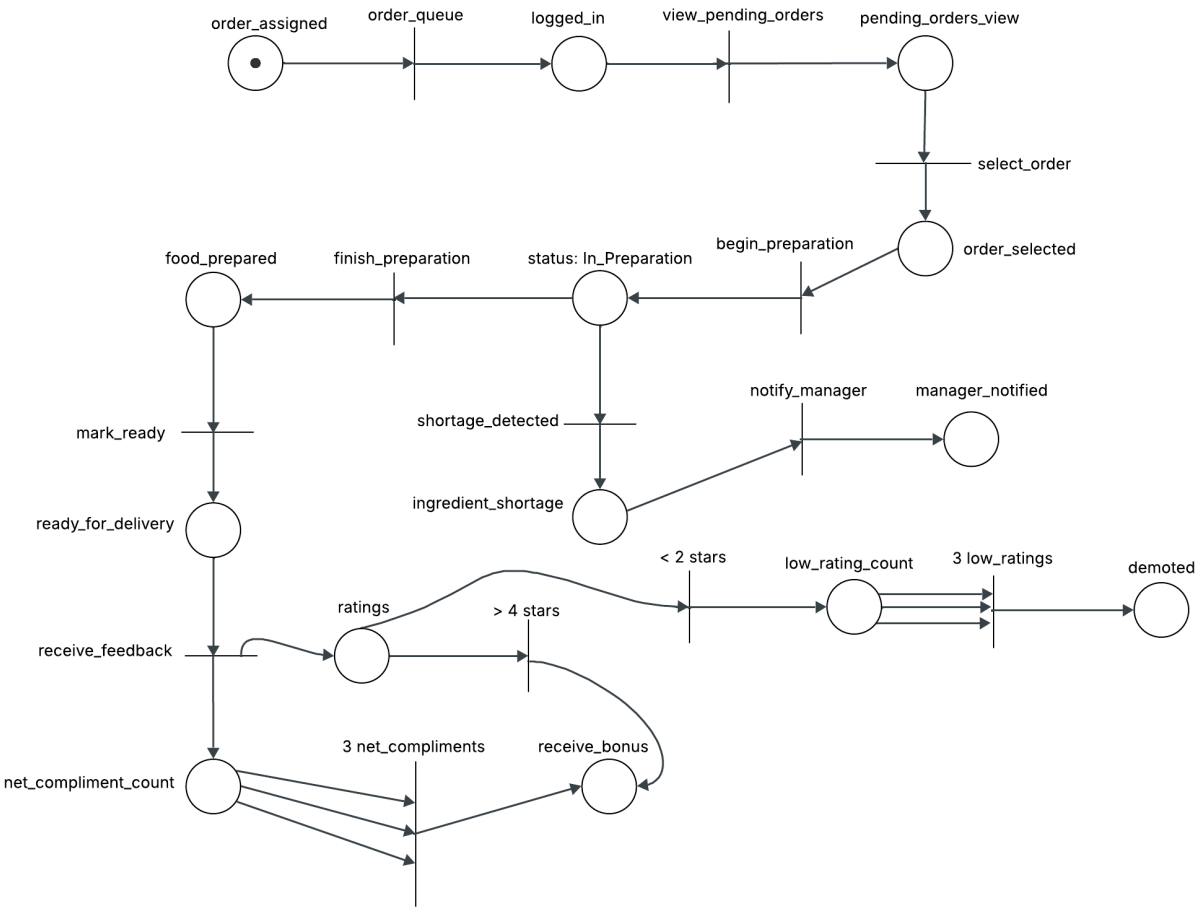


2.3 Petri-net Diagrams

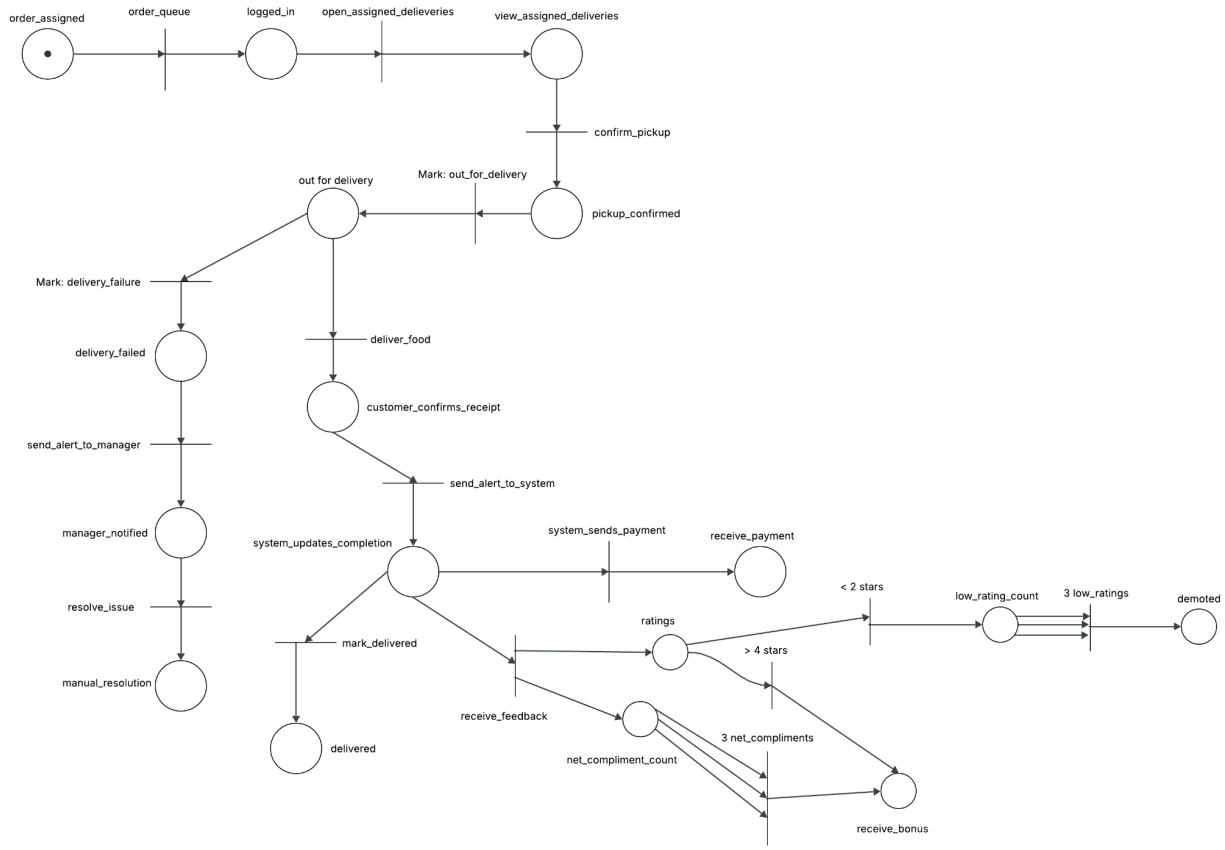
2.3.1 Order Food



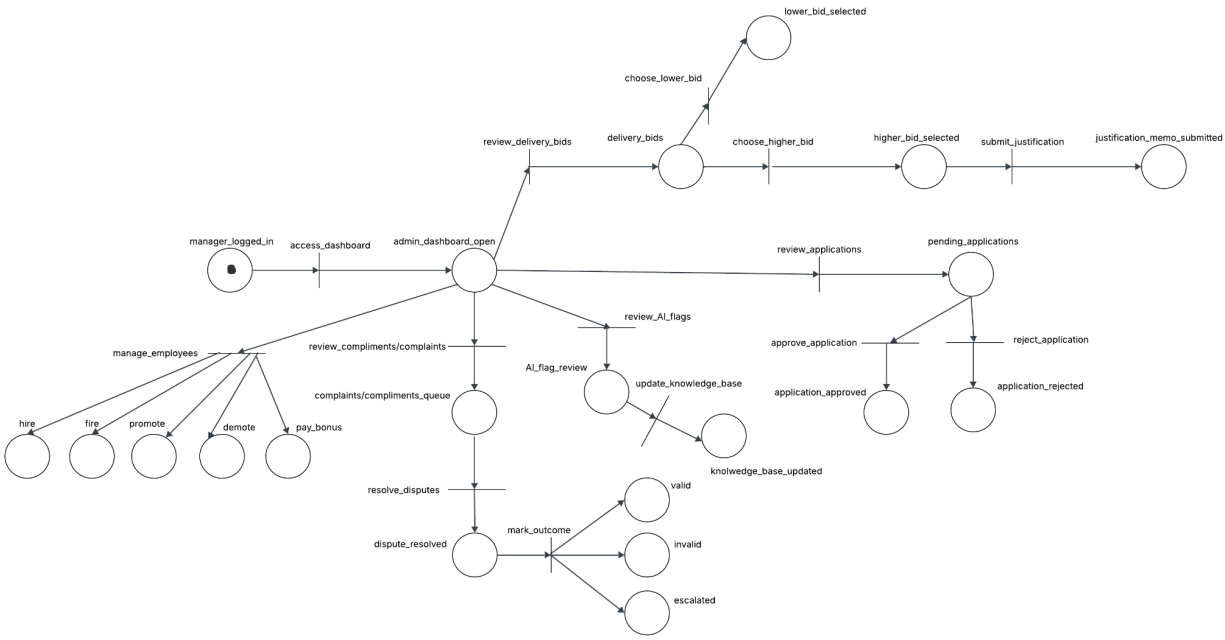
2.3.2 Cook Food



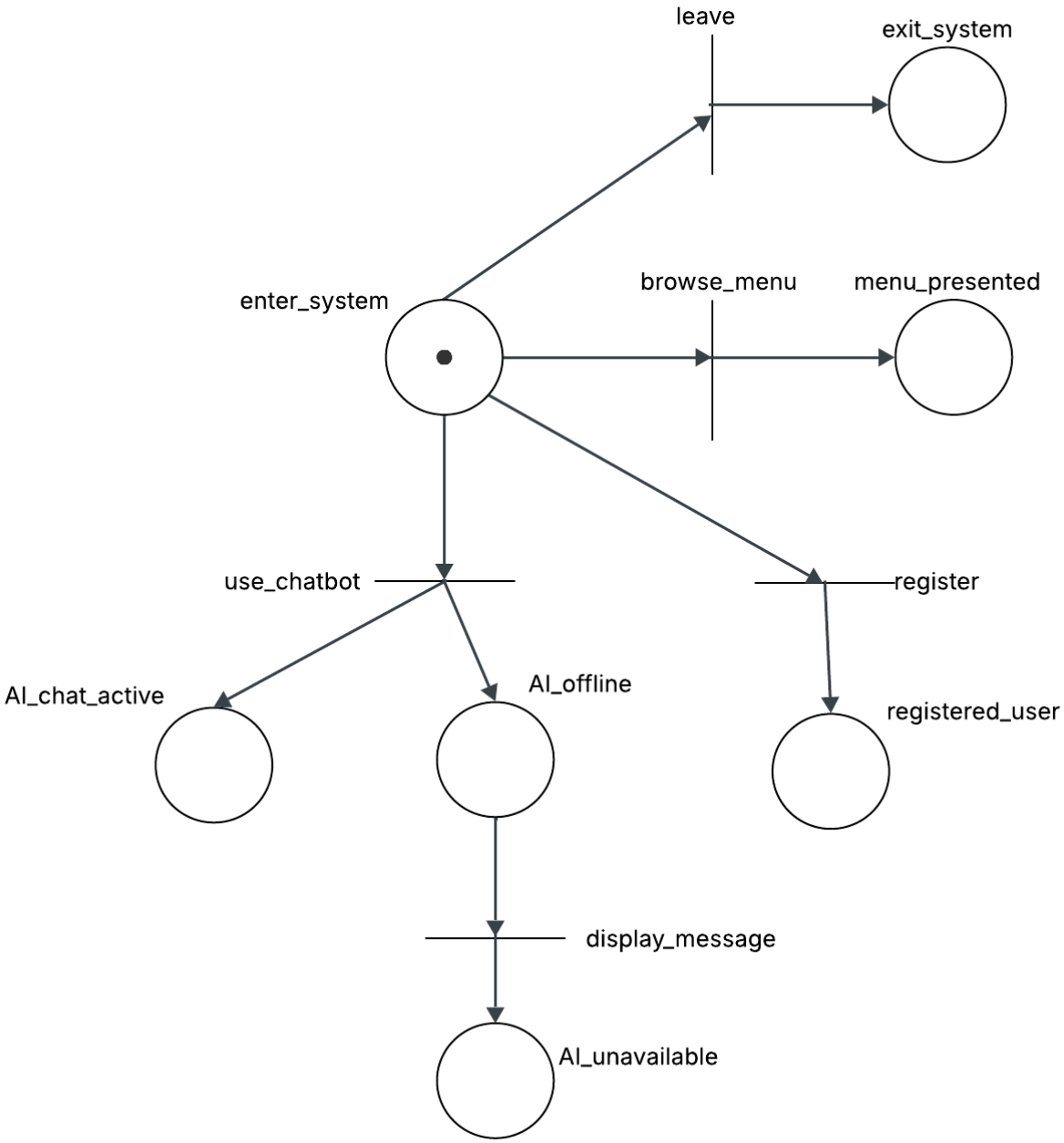
2.3.3 Deliver the Food



2.3.4 Manage System and Personnel

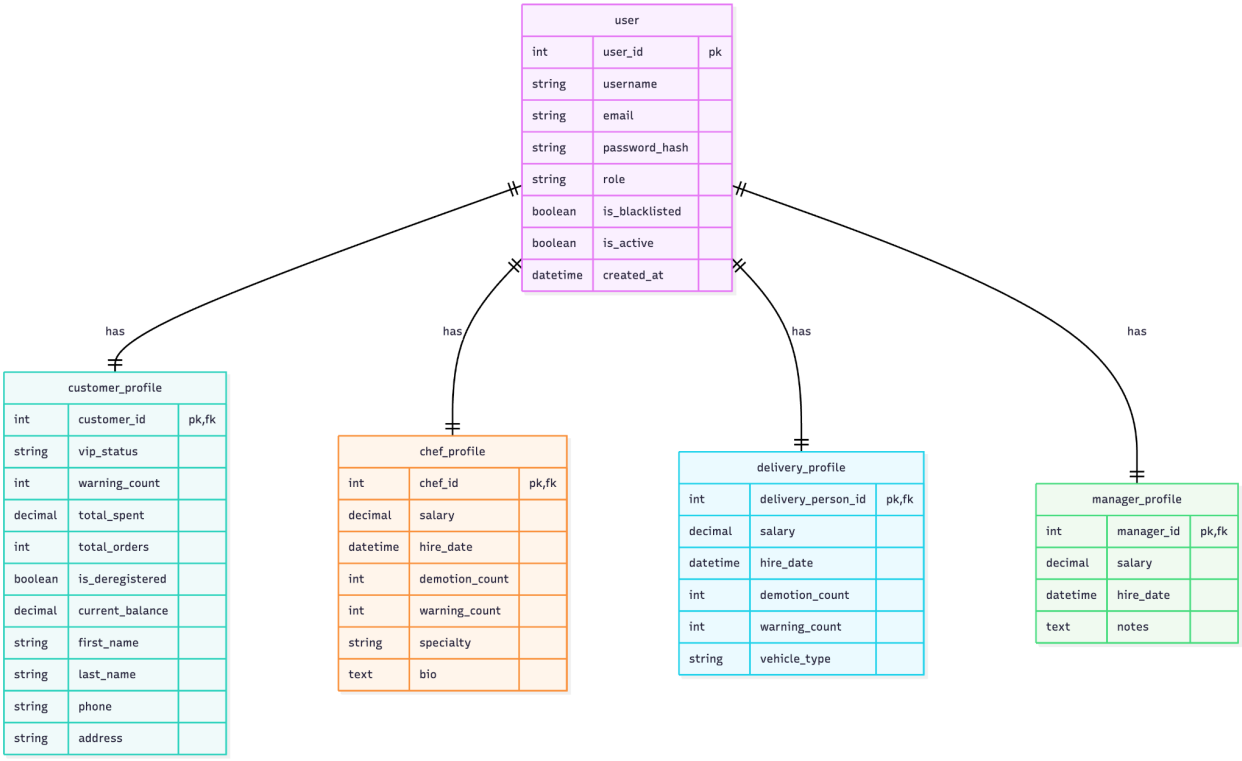


2.3.5 View Menu

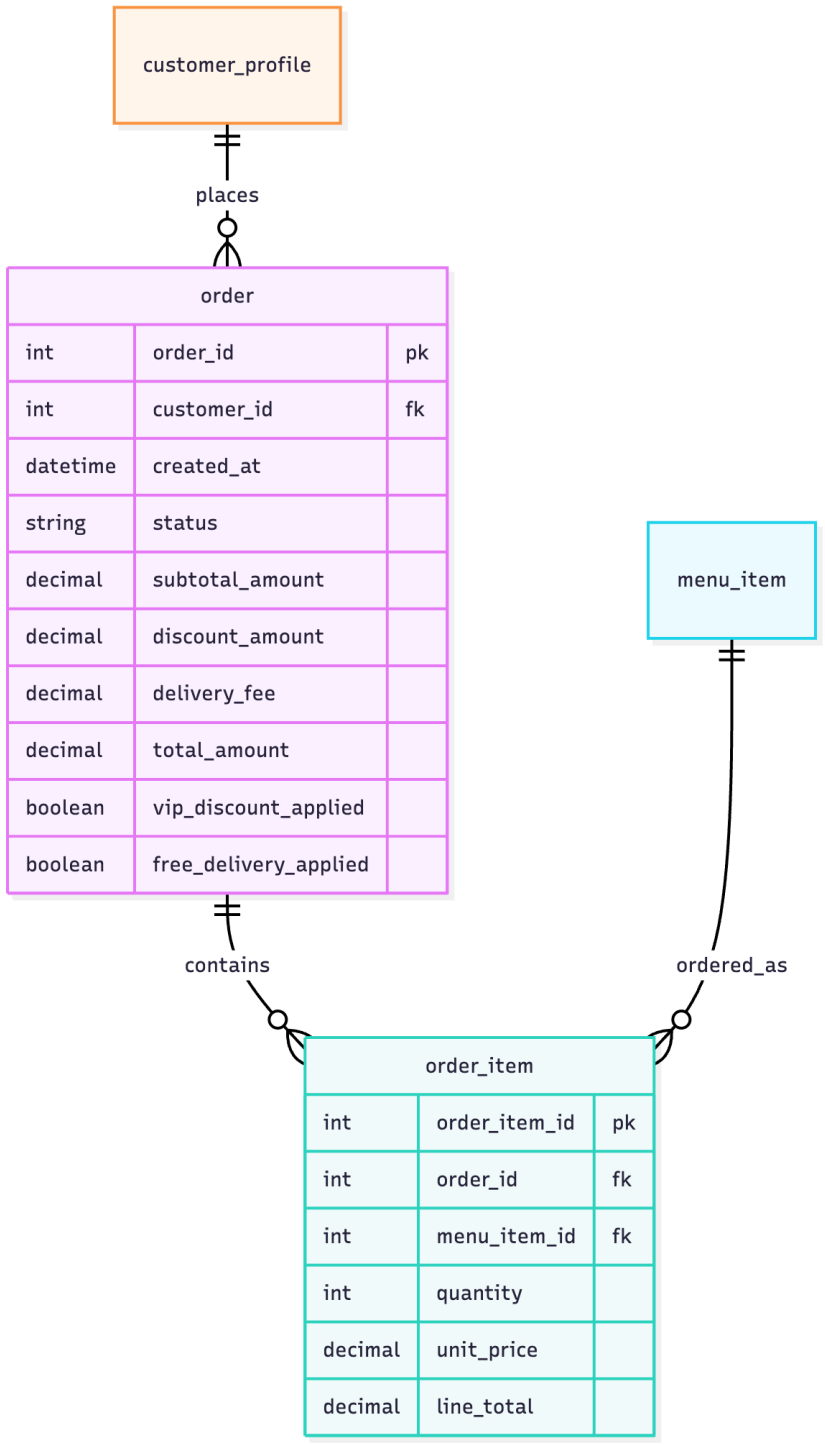


3. E-R Diagrams

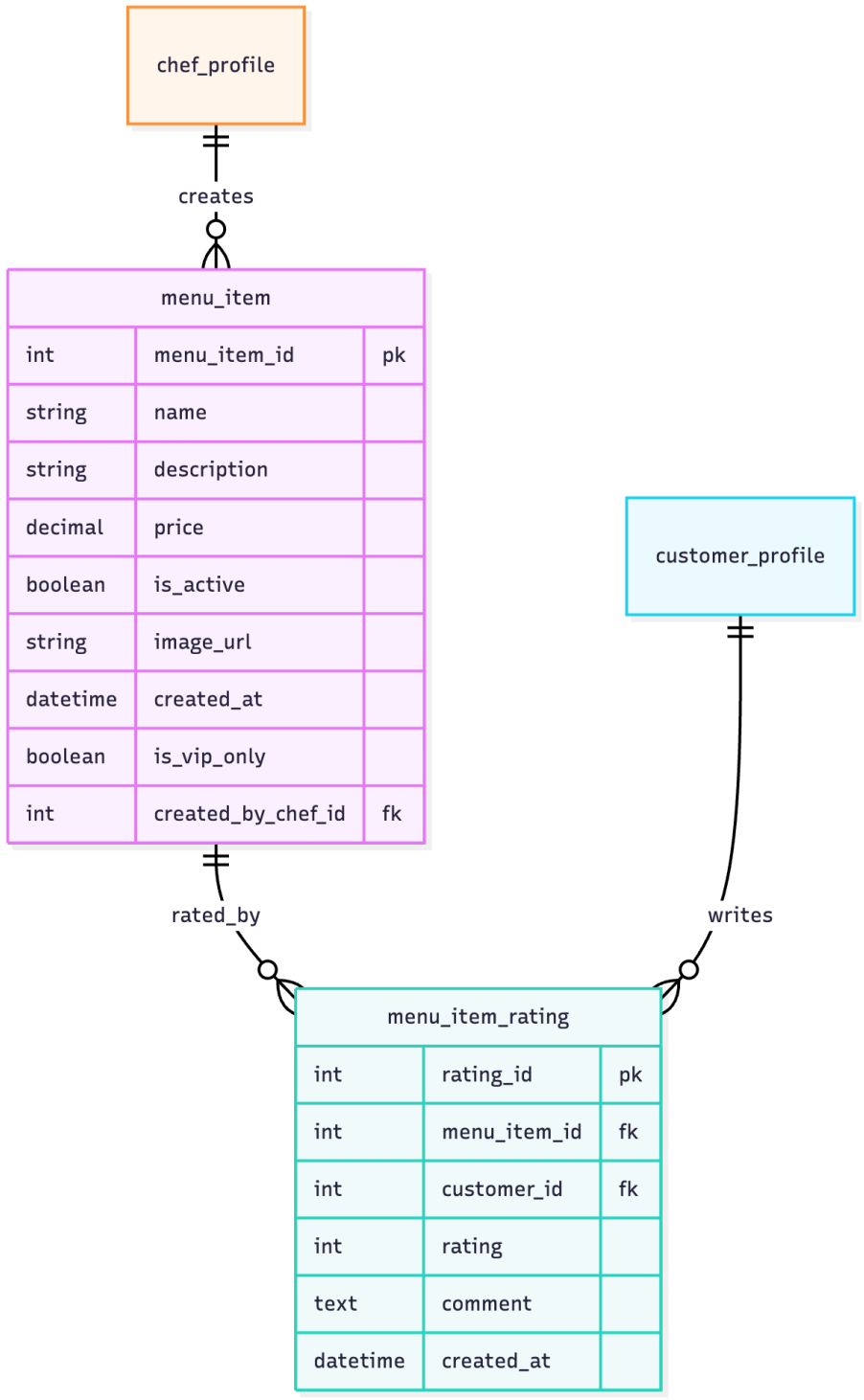
3.1 User - Roles



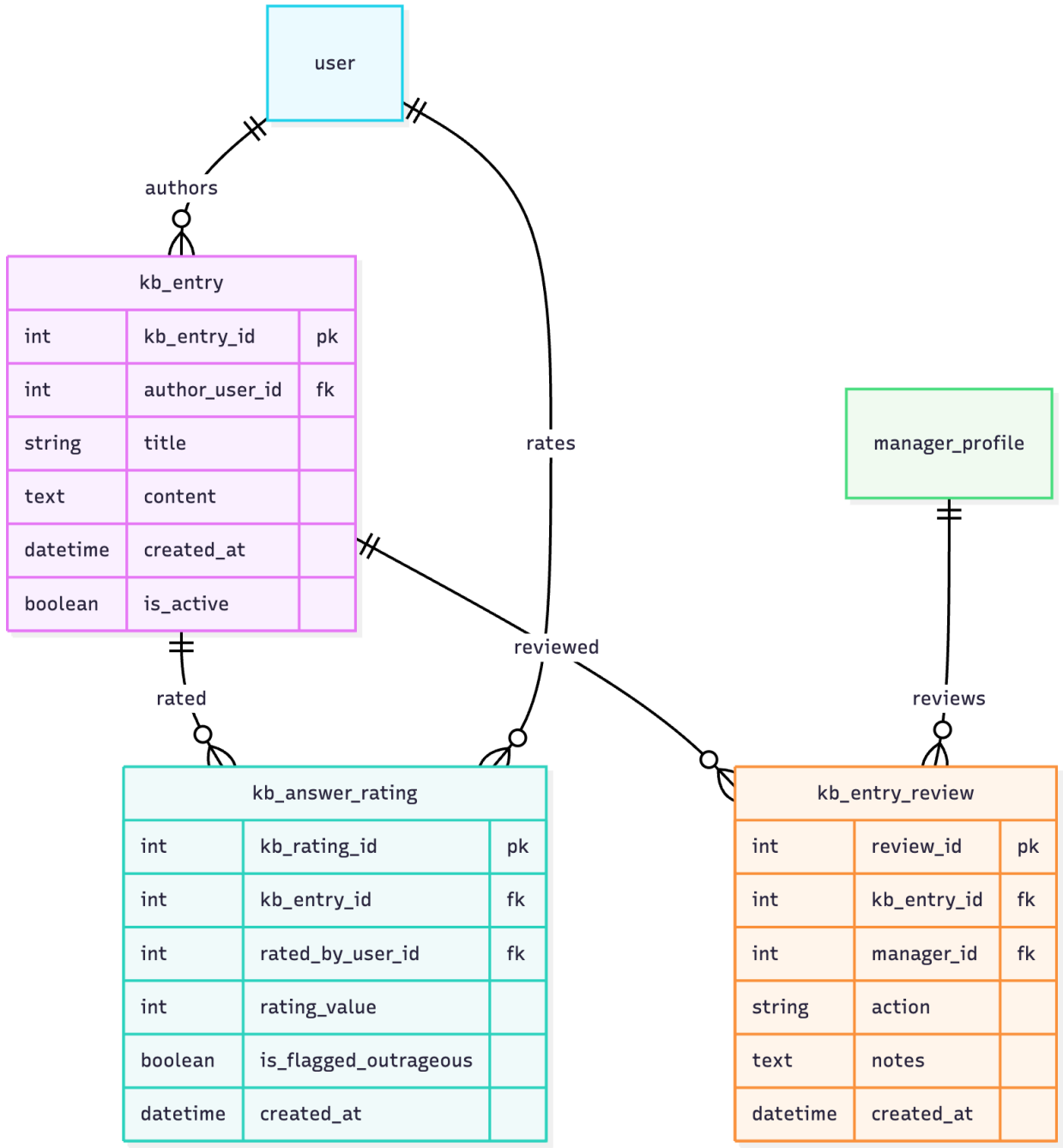
3.2 Orders


customer_profile
places
order
int order_id pk
int customer_id fk
datetime created_at
string status
decimal subtotal_amount
decimal discount_amount
decimal delivery_fee
decimal total_amount
boolean vip_discount_applied
boolean free_delivery_applied
menu_item
contains
ordered_as
order_item
int order_item_id pk
int order_id fk
int menu_item_id fk
int quantity
decimal unit_price
decimal line_total

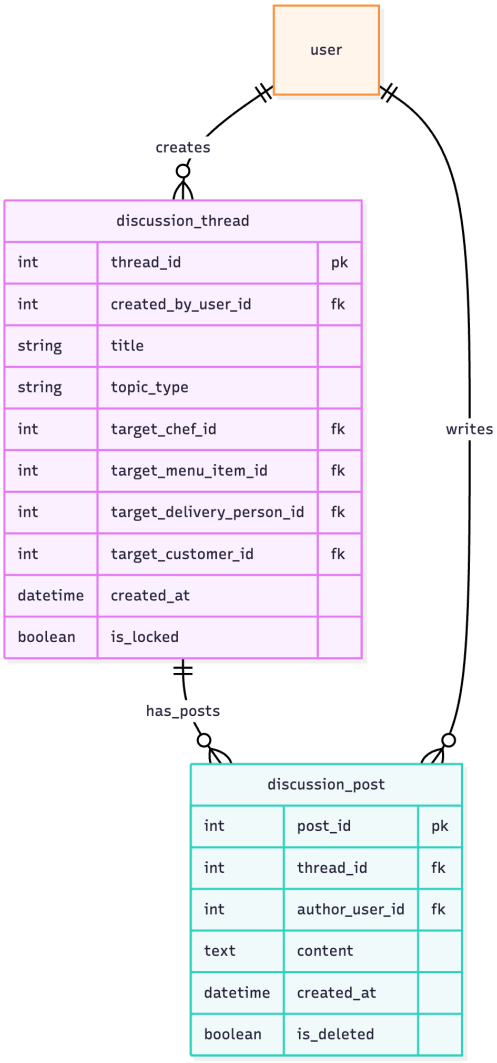
3.3 Menu


chef_profile
creates
menu_item
int menu_item_id pk
string name
string description
decimal price
boolean is_active
string image_url
datetime created_at
boolean is_vip_only
int created_by_chef_id fk
customer_profile
rated_by
writes
menu_item_rating
int rating_id pk
int menu_item_id fk
int customer_id fk
int rating
text comment
datetime created_at

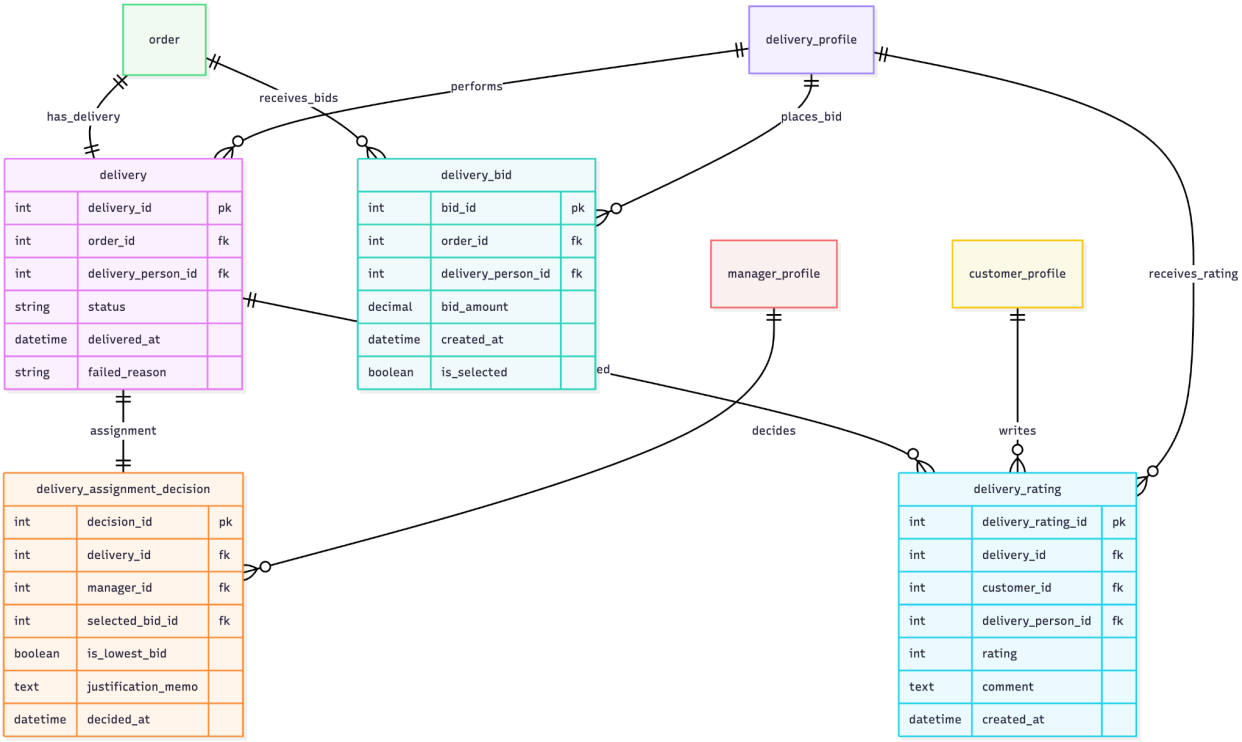
3.4 Knowledge Base



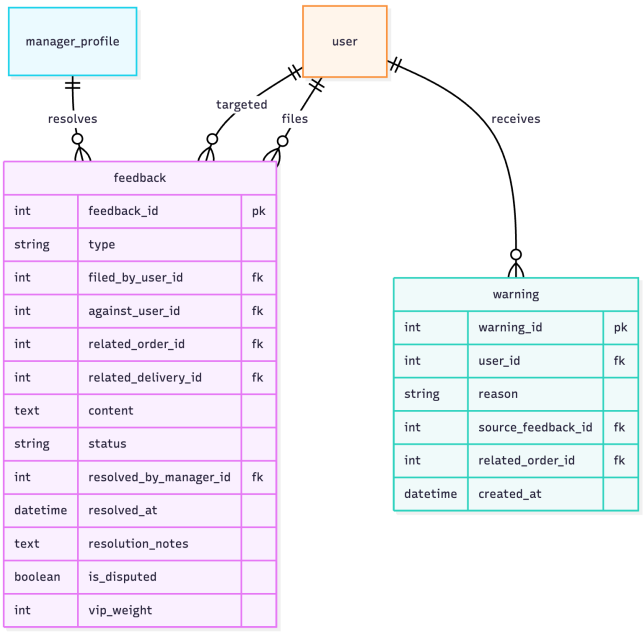
3.5 Forum


user
creates
writes
discussion_thread
int thread_id pk
int created_by_user_id fk
string title
string topic_type
int target_chef_id fk
int target_menu_item_id fk
int target_delivery_person_id fk
int target_customer_id fk
datetime created_at
boolean is_locked
has_posts
discussion_post
int post_id pk
int thread_id fk
int author_user_id fk
text content
datetime created_at
boolean is_deleted

3.6 Delivery



3.7 Complaints



4. Detailed Design of Methods

```
METHOD registerCustomer(formData)
   // Input: formData (name, email, password, address, phone)
   // Output: customerId OR errorCode
   VALIDATE required fields
   IF email EXISTS IN UserTable THEN RETURN EMAIL_ALREADY_EXISTS
   HASH password
   CREATE Customer(name, email, hashedPassword, address, phone,
               role="Customer", balance=0, totalSpent=0,
               orderCount=0, status="PendingApproval",
               warningCount=0, isVIP=false)
   SAVE Customer
   RETURN Customer.id
END


METHOD loginUser(email, password)
   // Input: email, password
   // Output: sessionToken OR errorCode
   user = FIND User BY email
   IF user IS NULL THEN RETURN INVALID_CREDENTIALS
   IF user.isBlacklisted THEN RETURN BLACKLISTED
   IF HASH(password) != user.hashedPassword THEN RETURN INVALID_CREDENTIALS
   sessionToken = GENERATE_SECURE_TOKEN()
   STORE sessionToken WITH userId, expiration
   RETURN sessionToken
END


METHOD processCustomerRegistration(managerId, userId, decision, reason)
   // Input: managerId, userId, decision(APPROVE/REJECT), reason
   // Output: boolean
   VERIFY managerId.role = "Manager"
   user = FIND User BY userId
   IF user IS NULL OR user.status != "PendingApproval" THEN RETURN false
   IF decision = APPROVE THEN
      user.status = "Active"
   ELSE
      user.status = "Rejected"
      user.rejectionReason = reason
   ENDIF
   SAVE user
```

```
    LOG decision
    RETURN true
END


METHOD updateVIPStatus(customerId)
    // Input: customerId
    // Output: boolean (isNowVIP)
    customer = FIND Customer BY customerId
    IF customer IS NULL THEN RETURN false
    IF customer.warningCount > 0 THEN RETURN false
    IF customer.totalSpent > 100 OR customer.orderCount >= 3 THEN
        customer.isVIP = true
        SAVE customer
        RETURN true
    ENDIF
    RETURN false
END


METHOD blacklistUser(managerId, userId)
    // Input: managerId, userId
    // Output: boolean
    VERIFY managerId.role = "Manager"
    user = FIND User BY userId
    IF user IS NULL THEN RETURN false
    user.isBlacklisted = true
    user.status = "Blacklisted"
    SAVE user
    ADD user.email TO BlacklistTable
    RETURN true
END


METHOD getMenuForVisitor()
    // Input: none
    // Output: listOfDishes
    popular = QUERY dishes ORDER BY orderCount DESC LIMIT N
    topRated = QUERY dishes ORDER BY averageRating DESC LIMIT M
    combined = MERGE_AND_DEDUP(popular, topRated)
    RETURN combined
END
```

```
METHOD getMenuForCustomer(customerId)
   // Input: customerId
   // Output: {mostOrdered, highestRated, popular}
   orders = QUERY past orders BY customerId
   mostOrdered = TOP_DISHES_FROM(orders)
   highestRated = TOP_RATED_BY_CUSTOMER(customerId)
   popular = QUERY dishes ORDER BY globalPopularity DESC LIMIT K
   RETURN {mostOrdered, highestRated, popular}
END


METHOD createOrUpdateDish(chefId, dishData)
   // Input: chefId, dishData
   // Output: dishId
   VERIFY chefId.role = "Chef"
   IF dishData.id EXISTS THEN
      dish = FIND Dish BY dishData.id
      UPDATE dish FROM dishData
   ELSE
      dish = NEW Dish(dishData)
      dish.orderCount = 0
      dish.averageRating = 0
   ENDIF
   SAVE dish
   RETURN dish.id
END


METHOD depositMoney(customerId, amount)
   // Input: customerId, amount
   // Output: newBalance OR errorCode
   IF amount <= 0 THEN RETURN INVALID_AMOUNT
   customer = FIND Customer BY customerId
   IF customer IS NULL OR customer.status != "Active" THEN RETURN
INVALID_CUSTOMER
   customer.balance += amount
   SAVE customer
   LOG transaction("DEPOSIT", amount)
   RETURN customer.balance
END
```

```
METHOD createOrder(customerId, cartItems)
   // Input: customerId, cartItems[]
   // Output: orderId OR errorCode
   IF cartItems IS EMPTY THEN RETURN EMPTY_CART
   customer = FIND Customer BY customerId
   IF customer IS NULL OR customer.status != "Active" THEN RETURN
INVALID_CUSTOMER
   order = NEW Order(customerId)
   totalPrice = 0
   FOR EACH item IN cartItems DO
      dish = FIND Dish BY item.dishId
      IF dish IS NULL OR NOT dish.isAvailable THEN RETURN DISH_NOT_AVAILABLE
      linePrice = dish.price * item.quantity
      ADD LINE(order, dish.id, item.quantity, dish.price)
      totalPrice += linePrice
   END FOR
   discount = CALCULATE_VIP_DISCOUNT(customer, totalPrice)
   order.originalPrice = totalPrice
   order.discountApplied = discount
   order.totalPrice = totalPrice - discount
   order.status = "PendingPayment"
   SAVE order
   RETURN order.id
END


METHOD CALCULATE_VIP_DISCOUNT(customer, totalPrice)
   // Input: customer, totalPrice
   // Output: discountAmount
   IF customer.isVIP THEN RETURN totalPrice * 0.05
   RETURN 0
END


METHOD confirmOrder(orderId)
   // Input: orderId
   // Output: true OR errorCode
   order = FIND Order BY orderId
   IF order IS NULL OR order.status != "PendingPayment" THEN RETURN INVALID_ORDER
   customer = FIND Customer BY order.customerId
```

```
    IF customer.balance < order.totalPrice THEN
        APPLY_WARNING(customer.id, "Insufficient funds")
        order.status = "Rejected_Insufficient_Funds"
        SAVE order
        RETURN INSUFFICIENT_FUNDS
    ENDIF
    customer.balance -= order.totalPrice
    customer.totalSpent += order.totalPrice
    customer.orderCount += 1
    SAVE customer
    updateVIPStatus(customer.id)
    order.status = "Paid"
    SAVE order
    enqueueOrderForKitchen(order.id)
    startDeliveryBidding(order.id)
    RETURN true
END


METHOD enqueueOrderForKitchen(orderId)
    // Input: orderId
    // Output: boolean
    order = FIND Order BY orderId
    IF order IS NULL THEN RETURN false
    order.status = "Queued_For_Preparation"
    SAVE order
    ADD orderId TO KitchenQueue
    NOTIFY Chefs
    RETURN true
END


METHOD getPendingOrdersForChef(chefId)
    // Input: chefId
    // Output: listOfOrders
    VERIFY chefId.role = "Chef"
    pending = QUERY Orders WHERE status IN ("Queued_For_Preparation", "In_Preparation")
    RETURN pending
END


METHOD updateOrderPreparationStatus(chefId, orderId, newStatus)
```

```
    // Input: chefId, orderId, newStatus
    // Output: boolean
    VERIFY chefId.role = "Chef"
    order = FIND Order BY orderId
    IF order IS NULL THEN RETURN false
    IF newStatus NOT IN ("In_Preparation","Ready_For_Delivery","On_Hold") THEN RETURN
false
    order.status = newStatus
    SAVE order
    IF newStatus = "Ready_For_Delivery" THEN
        NOTIFY Manager AND DeliveryPeople
    ENDIF
    RETURN true
END


METHOD reportIngredientShortage(chefId, orderId, note)
    // Input: chefId, orderId, note
    // Output: boolean
    VERIFY chefId.role = "Chef"
    order = FIND Order BY orderId
    IF order IS NULL THEN RETURN false
    order.status = "On_Hold"
    order.shortageNote = note
    SAVE order
    NOTIFY Manager
    RETURN true
END


METHOD createDeliveryBid(deliveryPersonId, orderId, bidAmount)
    // Input: deliveryPersonId, orderId, bidAmount
    // Output: bidId OR errorCode
    VERIFY deliveryPersonId.role = "DeliveryPerson"
    order = FIND Order BY orderId
    IF order IS NULL OR order.status != "Ready_For_Delivery" THEN RETURN
INVALID_ORDER
    IF bidAmount <= 0 THEN RETURN INVALID_BID
    bid = NEW DeliveryBid(orderId, deliveryPersonId, bidAmount, "Pending")
    SAVE bid
    RETURN bid.id
END
```

```
METHOD assignDelivery(managerId, orderId, chosenBidId, justificationText)
  // Input: managerId, orderId, chosenBidId, justificationText
  // Output: deliveryId OR errorCode
  VERIFY managerId.role = "Manager"
  order = FIND Order BY orderId
  IF order IS NULL OR order.status != "Ready_For_Delivery" THEN RETURN
INVALID_ORDER
  bids = QUERY DeliveryBids WHERE orderId = orderId AND status = "Pending"
  chosen = FIND IN bids WHERE id = chosenBidId
  IF chosen IS NULL THEN RETURN INVALID_BID
  lowest = MIN_BY_AMOUNT(bids)
  IF chosen.id != lowest.id THEN
    IF justificationText IS EMPTY THEN RETURN JUSTIFICATION_REQUIRED
    LOG justification(managerId, orderId, chosenBidId, justificationText)
  ENDIF
  delivery = NEW Delivery(orderId, chosen.deliveryPersonId, chosen.amount, "Assigned")
  SAVE delivery
  order.status = "Awaiting_Pickup"
  SAVE order
  FOR EACH bid IN bids DO
    bid.status = (bid.id = chosenBidId ? "Accepted" : "Rejected")
    SAVE bid
  END FOR
  NOTIFY chosen.deliveryPersonId
  RETURN delivery.id
END


METHOD updateDeliveryStatus(deliveryPersonId, deliveryId, newStatus, note)
  // Input: deliveryPersonId, deliveryId, newStatus, note
  // Output: boolean
  VERIFY deliveryPersonId.role = "DeliveryPerson"
  delivery = FIND Delivery BY deliveryId
  IF delivery IS NULL OR delivery.deliveryPersonId != deliveryPersonId THEN RETURN false
  delivery.status = newStatus
  delivery.note = note
  SAVE delivery
  order = FIND Order BY delivery.orderId
  IF newStatus = "Out_For_Delivery" THEN
    order.status = "Out_For_Delivery"
```

```
    ELSE IF newStatus = "Delivered" THEN
        order.status = "Completed"
        RELEASE_PAYMENT_TO_DELIVERY_PERSON(deliveryPersonId, delivery.bidAmount)
        PROMPT_FEEDBACK(order.customerId, deliveryPersonId, order.id)
    ELSE IF newStatus = "Delivery_Failed" THEN
        order.status = "Delivery_Failed"
        NOTIFY Manager
    ENDIF
    SAVE order
    RETURN true
END


METHOD fileComplaintOrCompliment(fromUserId, toEntityId, entityType, isComplaint,
message)
    // Input: fromUserId, toEntityId, entityType, isComplaint, message
    // Output: recordId
    fromUser = FIND User BY fromUserId
    weight = 1
    IF fromUser.role = "Customer" AND fromUser.isVIP THEN weight = 2
    record = NEW ReputationRecord(fromUserId, toEntityId, entityType,
                                  isComplaint, message, weight,
                                  status="PendingReview")
    SAVE record
    NOTIFY Manager
    RETURN record.id
END


METHOD resolveComplaint(managerId, complaintId, outcome)
    // Input: managerId, complaintId, outcome
    // Output: boolean
    VERIFY managerId.role = "Manager"
    record = FIND ReputationRecord BY complaintId
    IF record IS NULL OR record.isComplaint = false THEN RETURN false
    record.status = outcome
    SAVE record
    IF outcome = "VALID" THEN
        APPLY_COMPLAINT_EFFECT(record.toEntityId, record.entityType, record.weight)
    ELSE IF outcome = "INVALID" THEN
        APPLY_WARNING(record.fromUserId, "Complaint without merit")
    ENDIF
```

```
    RETURN true
END


METHOD APPLY_WARNING(userId, reason)
    // Input: userId, reason
    // Output: none
    user = FIND User BY userId
    IF user IS NULL THEN RETURN
    user.warningCount += 1
    LOG warning(reason)
    IF user.role = "Customer" THEN
        IF user.isVIP THEN
            IF user.warningCount >= 2 THEN
                user.isVIP = false
                user.warningCount = 0
            ENDIF
        ELSE
            IF user.warningCount >= 3 THEN
                user.status = "Deregistered"
                blacklistUserForWarnings(user.id)
            ENDIF
        ENDIF
    ENDIF
    SAVE user
END


METHOD APPLY_COMPLAINT_EFFECT(employeeId, entityType, weight)
    // Input: employeeId, entityType, weight
    // Output: none
    employee = FIND User BY employeeId
    IF employee IS NULL THEN RETURN
    employee.netComplaints += weight
    SAVE employee
    IF entityType IN ("Chef","DeliveryPerson") AND employee.netComplaints >= 3 THEN
        IF employee.demotionsCount = 0 THEN
            employee.role = "Demoted_" + entityType
            employee.demotionsCount += 1
        ELSE IF employee.demotionsCount = 1 THEN
            employee.status = "Terminated"
        ENDIF
```

```
        SAVE employee
    ENDIF
END


METHOD handleChatQuery(userId, queryText)
    // Input: userId (nullable), queryText
    // Output: (answerText, source, answerId)
    kbAnswer = queryKnowledgeBase(queryText)
    IF kbAnswer != NULL THEN
        answerText = kbAnswer.text
        source = "KB"
    ELSE
        llm = queryLLM(queryText)
        answerText = llm.text
        source = "LLM"
    ENDIF
    record = NEW ChatAnswer(userId, queryText, answerText, source,
                            rating=null, flagged=false)
    SAVE record
    RETURN (answerText, source, record.id)
END


METHOD queryKnowledgeBase(queryText)
    // Input: queryText
    // Output: kbAnswerOrNull
    matches = SEARCH KBArticles BY queryText
    IF matches EMPTY THEN RETURN NULL
    best = HIGHEST_SIMILARITY(matches)
    RETURN {text: best.answerText, articleId: best.id}
END


METHOD queryLLM(queryText)
    // Input: queryText
    // Output: llmAnswer
    resp = CALL_EXTERNAL_LLM_API({prompt: queryText})
    RETURN {text: resp.text}
END
```

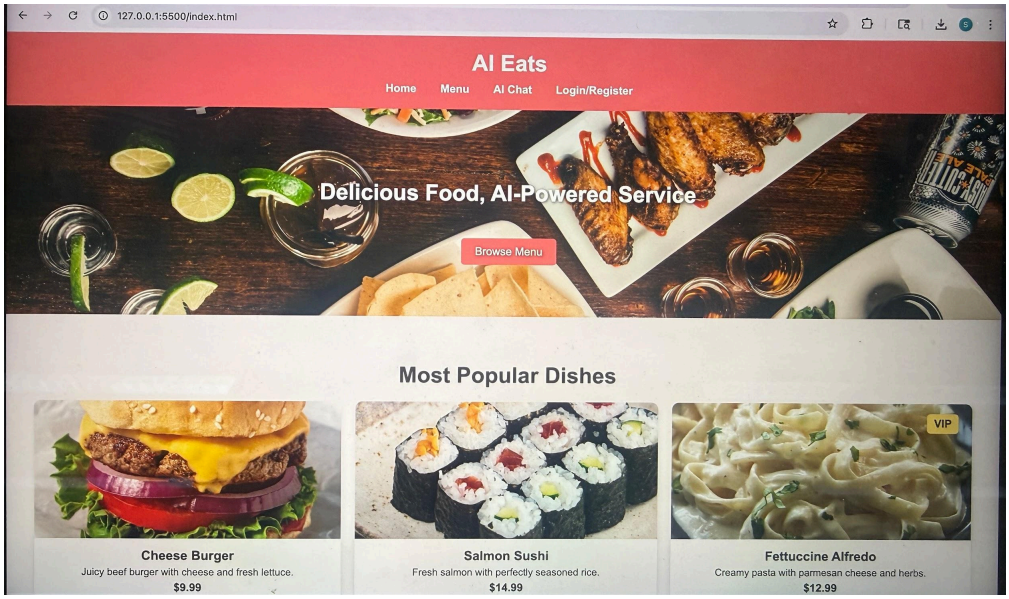
```
METHOD rateAnswer(userId, answerId, rating)
   // Input: userId, answerId, rating(0–5)
   // Output: boolean
   answer = FIND ChatAnswer BY answerId
   IF answer IS NULL THEN RETURN false
   answer.rating = rating
   SAVE answer
   IF rating = 0 THEN
      answer.flagged = true
      SAVE answer
      NOTIFY Manager
   ENDIF
   RETURN true
END


METHOD startVoiceRecording(clientState)
   // Input: clientState
   // Output: updatedClientState
   clientState.mode = "Listening"
   INITIATE_AUDIO_CAPTURE()
   RETURN clientState
END


METHOD stopAndSendRecording(audioFile, userId)
   // Input: audioFile, userId
   // Output: responseAudioFile OR errorCode
   FINALIZE_AUDIO_CAPTURE(audioFile)
   resp = HTTP_POST("/api/voice-query", {audioFile, userId})
   IF resp.status != 200 THEN RETURN VOICE_PROCESSING_ERROR
   RETURN resp.audioFile
END


METHOD processVoiceRequest(audioFile, userId)
   // Input: audioFile, userId
   // Output: responseAudioFile
   transcript = callElevenLabsSTT(audioFile)
   (answerText, source, answerId) = handleChatQuery(userId, transcript)
   responseAudio = callElevenLabsTTS(answerText)
   RETURN responseAudio
```

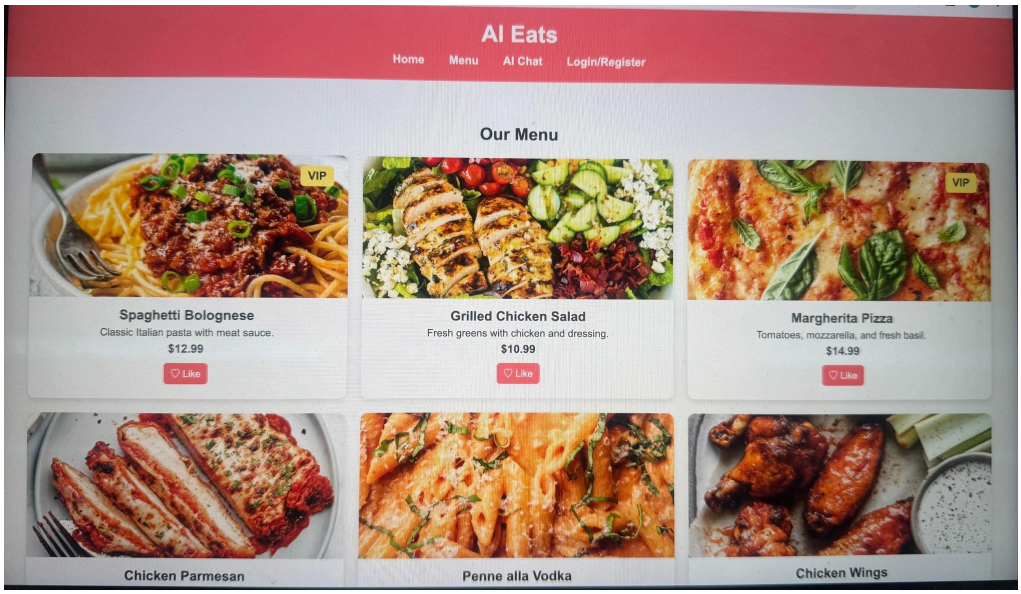
```
END


METHOD callElevenLabsSTT(audioFile)
   // Input: audioFile
   // Output: transcriptText
   resp = CALL_ELEVENLABS_STT_API({audio: audioFile})
   RETURN resp.transcript
END


METHOD callElevenLabsTTS(text)
   // Input: text
   // Output: audioFile
   resp = CALL_ELEVENLABS_TTS_API({text, voiceId: DEFAULT_VOICE})
   RETURN resp.audioFile
END


METHOD getAdminDashboardData(managerId)
   // Input: managerId
   // Output: dashboardData
   VERIFY managerId.role = "Manager"
   pending = QUERY Users WHERE status = "PendingApproval"
   openComplaints = QUERY ReputationRecords WHERE status="PendingReview" AND
isComplaint=true
   flagged = QUERY ChatAnswers WHERE flagged=true
   unresolvedCount = COUNT openComplaints
   RETURN {
      pendingRegistrations: pending,
      openComplaints: openComplaints,
      flaggedAIAnswers: flagged,
      unresolvedComplaintsCount: unresolvedCount
   }
END


METHOD handleSystemAlertForComplaints(managerId)
   // Input: managerId
   // Output: none
   VERIFY managerId.role = "Manager"
```

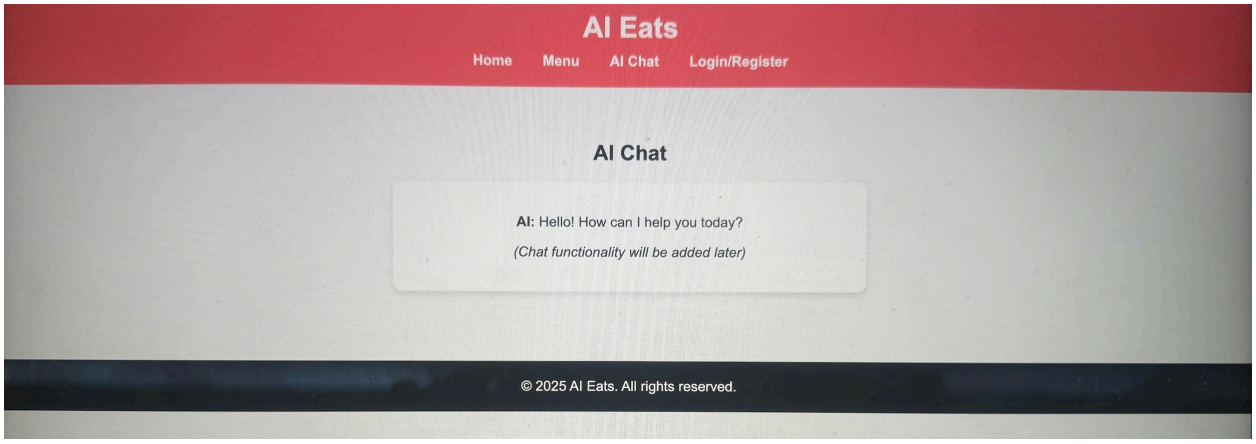
```
    unresolved = QUERY ReputationRecords WHERE status="PendingReview" AND
isComplaint=true
    IF COUNT(unresolved) > THRESHOLD THEN
        SEND_PRIORITY_ALERT(managerId, "Multiple unresolved complaints")
    ENDIF
END
```

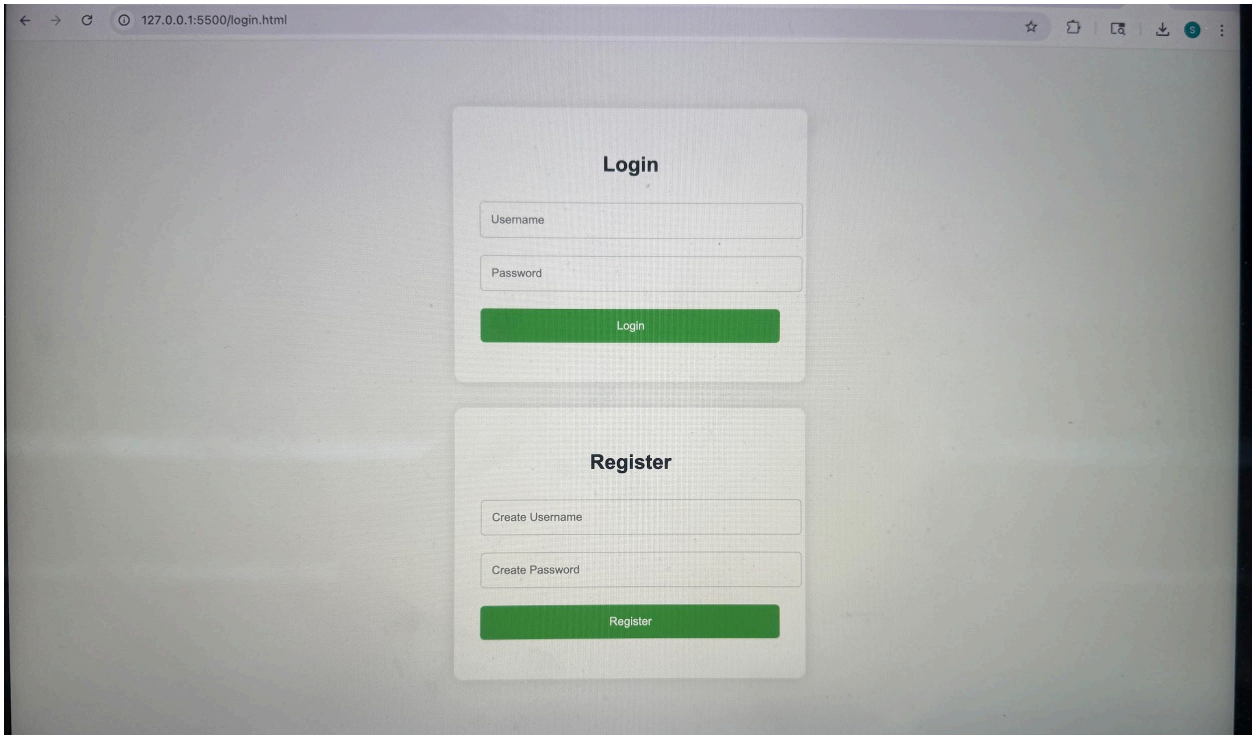
5. System Screens

5.1 Major GUI Screens



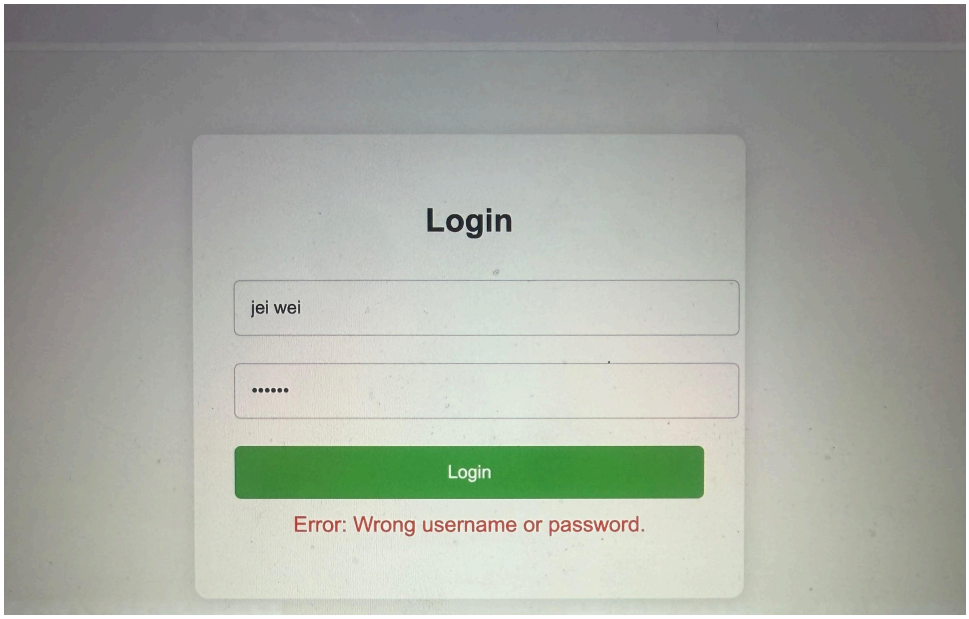


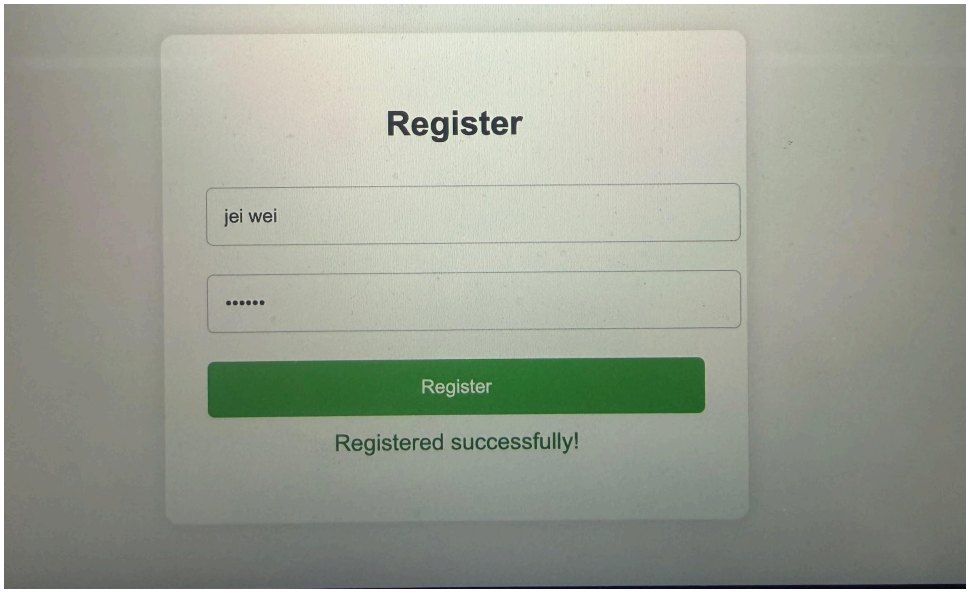


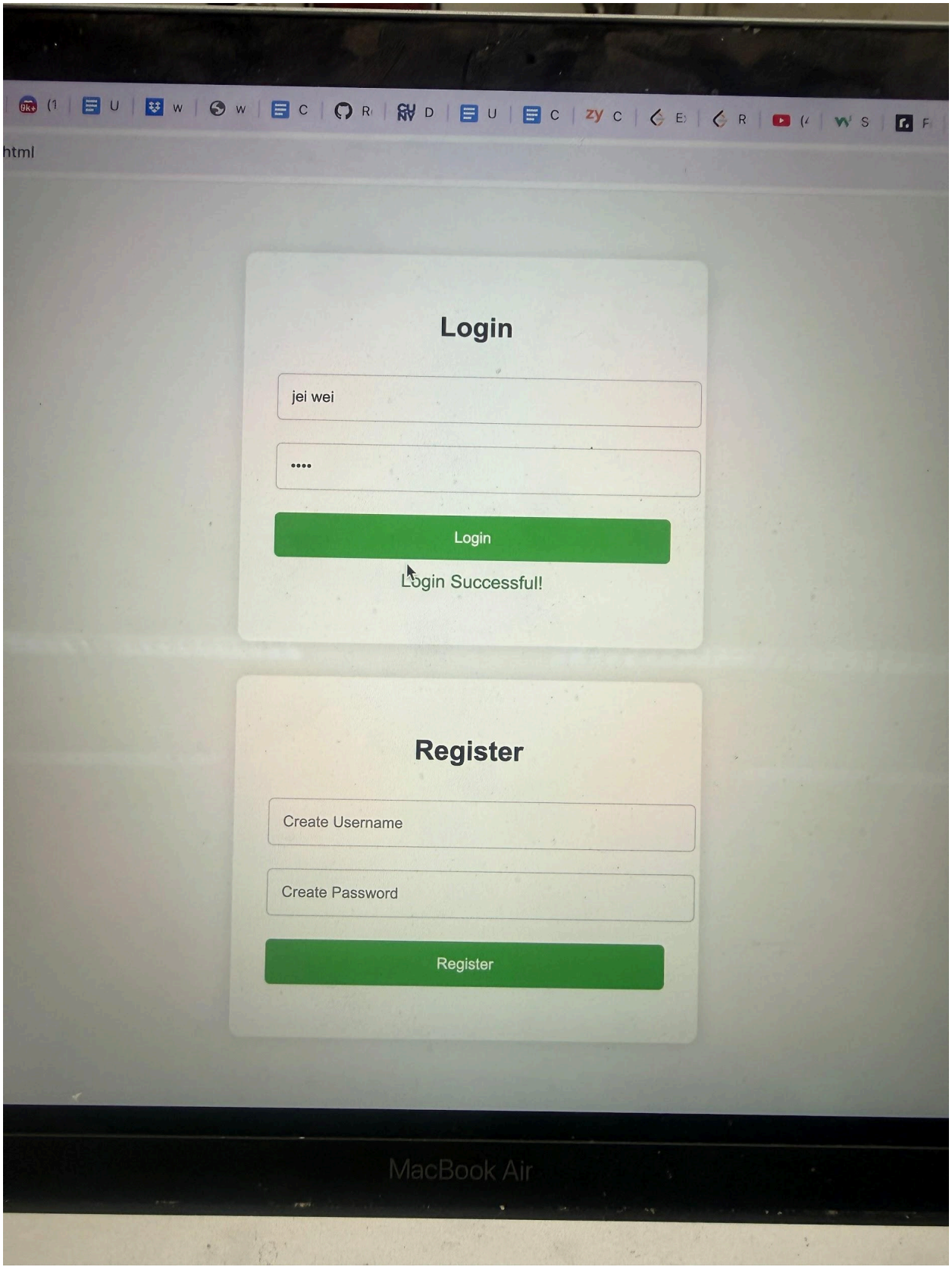


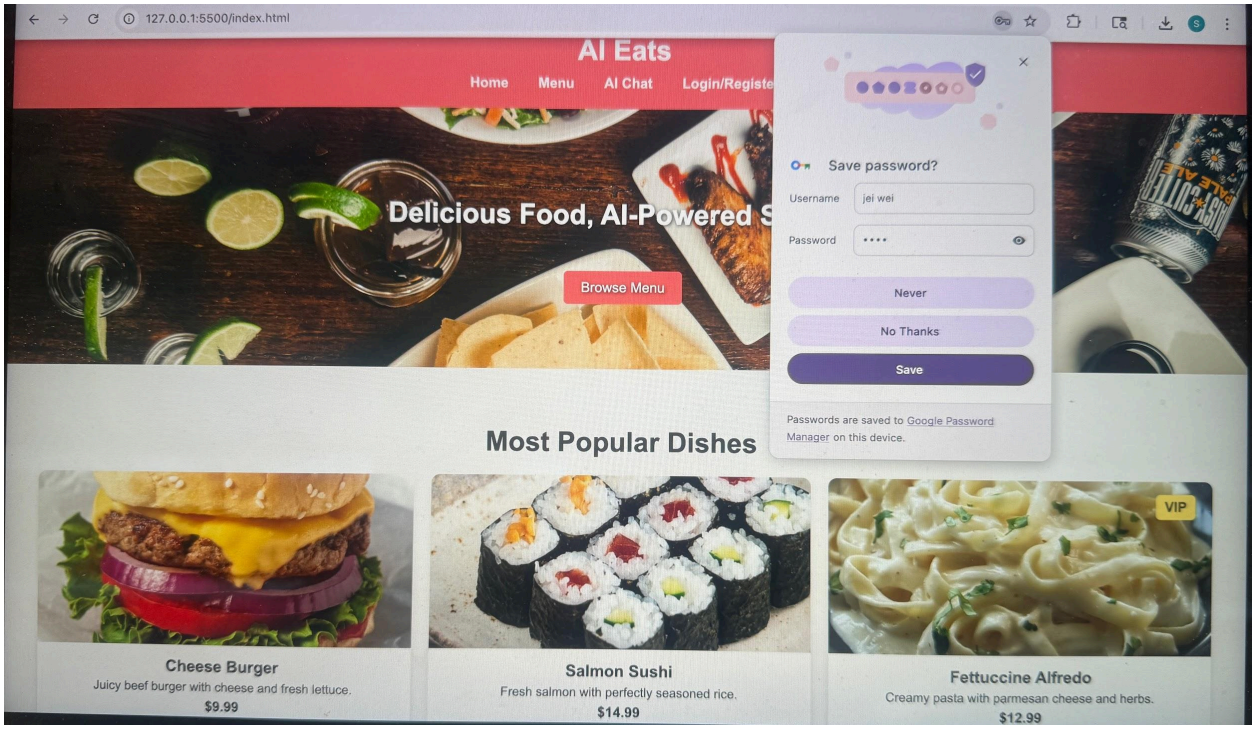
5.2 Sample Prototype of Login/Register Functionality

1) If you aren't a registered user and try to log in, it will return a message saying "Error: Wrong username or password."
2) If you want to register, you can put in a username and password and register.
3) Then when you try to log in, it won't return an error. It will instead return a message saying "Login successful!"
4) After you log in, it will automatically bring you to the home page.


Login
jei wei
••••••
Login
Error: Wrong username or password.


Register
jei wei
••••••
Register
Registered successfully!


html
Login
jei wei
••••
Login
Login Successful!
Register
Create Username
Create Password
Register
MacBook Air



6. Teamwork Memos

Meeting Date: Nov. 11

The group met to outline the design report requirements and clarify the overall direction of the AI Eats system. Everyone reviewed the project workflow and agreed that the priority should be finalizing the use cases before moving on to diagrams. Mohmed began drafting the Order, Cook, and Delivery workflows, while Kritan started outlining the ER diagram structure. Sajid and Amit began sketching out the main method logic for customer registration, ordering, and the major GUI screens and some of the main functionalities for the system. The team agreed to meet weekly and to keep all diagrams and drafts in the shared repository.

Meeting Date: Nov. 13

A work division was finalized. Mohmed continued refining the use cases, making sure alternate flows were fully documented. Kritan took full responsibility for the ER diagrams, ensuring the user roles and delivery interactions were clearly separated. Sajid and Amit focused on detailed pseudocode for VIP updates, order confirmation, and delivery bidding. The team also discussed Petri-nets and agreed they would be created after all use case steps were verified. Everyone

confirmed that once the system flows were stable, each member would later produce sequence diagrams for their assigned sections.

Meeting Date: Nov. 15

The team reviewed progress and began integrating UI considerations into the design. Amit and aid drafted prototype layouts for the login/register and menu screens. Mohmed updated the View Menu use case to reflect the personalized customer experience. Kritan refined the Cook and Delivery diagrams after identifying inconsistencies with status transitions. Several minor issues in the workflow logic were discovered, which the team discussed and resolved together. Everyone continued pushing updates to the shared repo as they worked.

Meeting Date: Nov. 17

With the report deadline approaching, the group reviewed all remaining requirements and ensured each section was on track. Most diagrams were completed, and the method descriptions were nearly finalized. The team discussed timing concerns, the need to finalize Petri-nets, and the importance of maintaining consistent terminology across all diagrams and sections. Members planned to finish any remaining components individually and begin assembling the full report. A final proofreading pass and diagram formatting review were scheduled for later in the week.

7. Git Repository

https://github.com/kritanbaniya/Resturant322