CurryMuncha — Phase II: Design Report

Version: 1.0
Team: Syed Ferdows, John Ortega, Aaliyan Qureshi, Minhazur Rahman, Shaeem Rockcliffe
Date: 2025-11-13

---

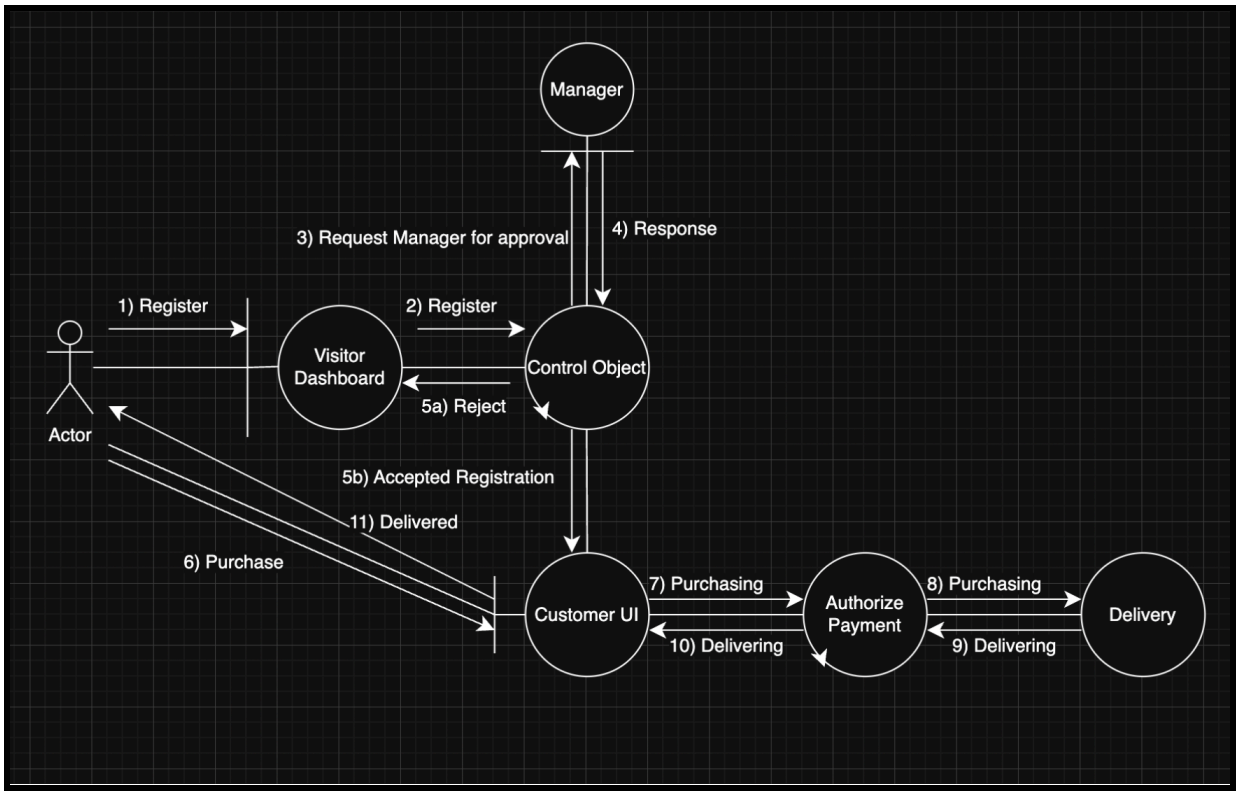
1. Introduction — System overview & collaboration-class diagram

Goal: Phase II The design emphasizes modularity, clear data models, role-based access, and AI integration points.

High-level architecture (textual):

- **Frontend (React):** UI components, pages, client-side validation, cart management, and chatbot UI.
- **Backend (Express + Node.js):** REST API for CRUD operations, order processing, authentication, payment gateway handler, and a dedicated AI microservice that queries the LLM and local knowledge base.
- **Database (Firestore):** Users, Restaurants, MenuItems, Orders, Deliveries, Reviews, ChatLogs, Inventory.
- **Third-party services:** Payment gateway (Stripe-like), Maps API for routing, Ollama/HF LLM for recommendations, Firebase Auth (optional).

Collaboration (class) diagram (text description + Mermaid class diagram)



---

2. Use cases (each: normal + exceptional scenarios)

We list every use case from Phase I and provide both normal and exceptional flows.

2.1 Browsing Menu

Actors: Visitor, Customer, VIP Customer

Normal scenario:

1. Actor opens Menu page.

2. Frontend requests **`/api/menu`** (optionally with filters).
3. Backend returns list of categories and menu items with images, price, availability.
4. UI displays results; user filters/sorts; selects item to view details.

Exceptional scenarios:

- Backend returns partial data (some images missing) → show placeholders and a retry button.
- Network error/timeouts → show cached data (if present) and an "offline read-only" banner.
- Item marked unavailable after page load → on add-to-cart, backend validates availability and shows message.

2.2 Placing Order

Actors: Customer, VIP Customer

Normal scenario:

1. User adds items to cart.
2. User proceeds to checkout, enters/chooses address and payment method.
3. Client calls **`/api/orders/checkout`** → server validates items/inventory, computes total, applies discounts, and creates **`Order`** with status **`PLACED`.**
4. Payment gateway is called; on success, system queues order for kitchen and assigns a delivery (or marks for pickup).
5. UI shows order confirmation and tracking link.

Exceptional scenarios:

- Payment fails → order is not created or is created as **`PENDING_PAYMENT`**. Notify user and provide retry.
- Inventory shortage detected during checkout → remove or flag unavailable items, show adjusted total, ask user to reconfirm.
- Duplicate submission (user clicks twice) → idempotency token on checkout to prevent double orders.

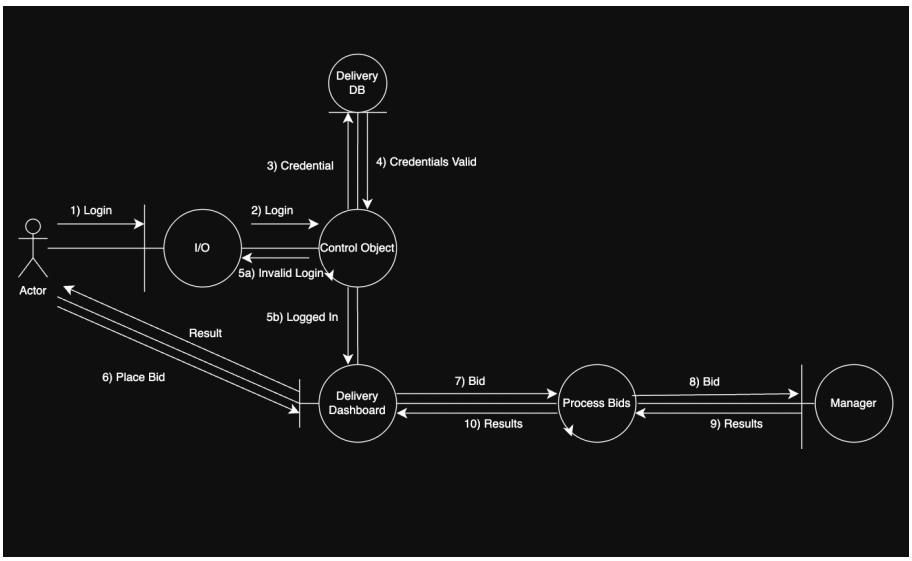
2.3 Delivering Order

Actors: Delivery Dude

Normal scenario:

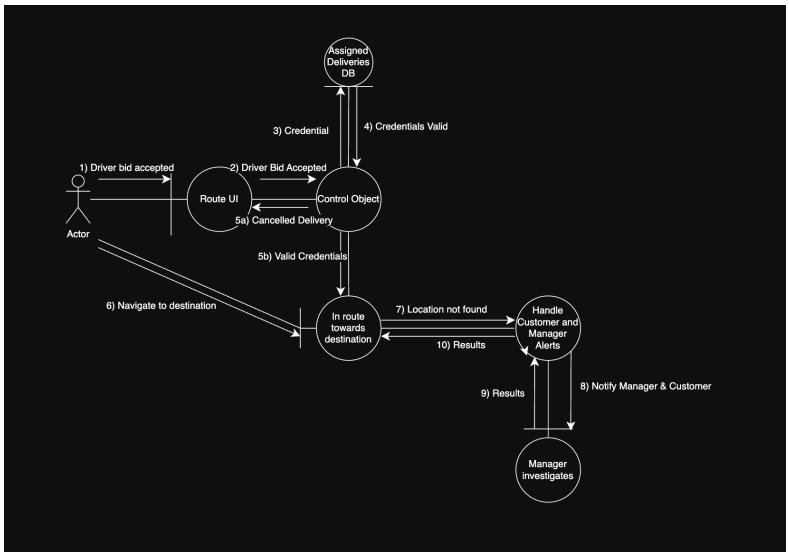
1. Delivery logs in, fetches assigned orders via **`/api/delivery/assignments`.**

2. Delivery places bid to have a chance at being selected to deliver meals
3. System records actual delivery time and updates order to the user.



Exceptional scenarios:

- Delivery cannot locate address → delivery marks **`FAILED_ATTEMPT`**; system notifies customer and manager.

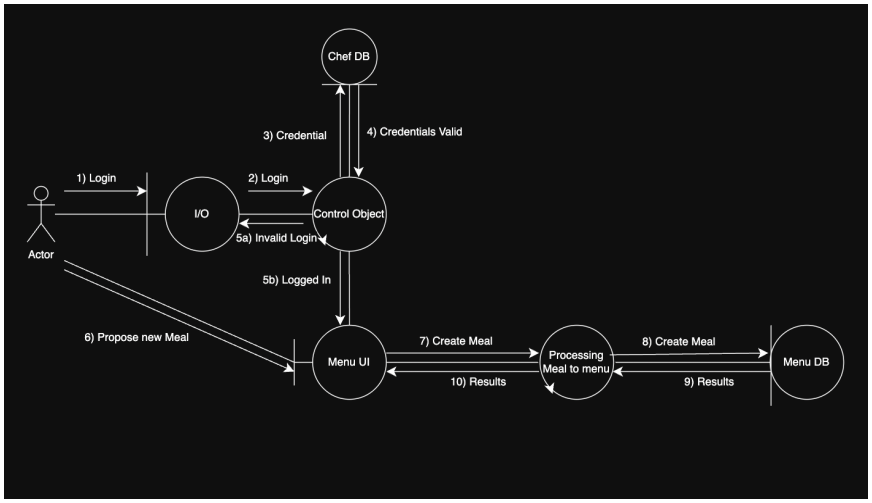


2.4 Change Menu

Actors: Chef (authorized)

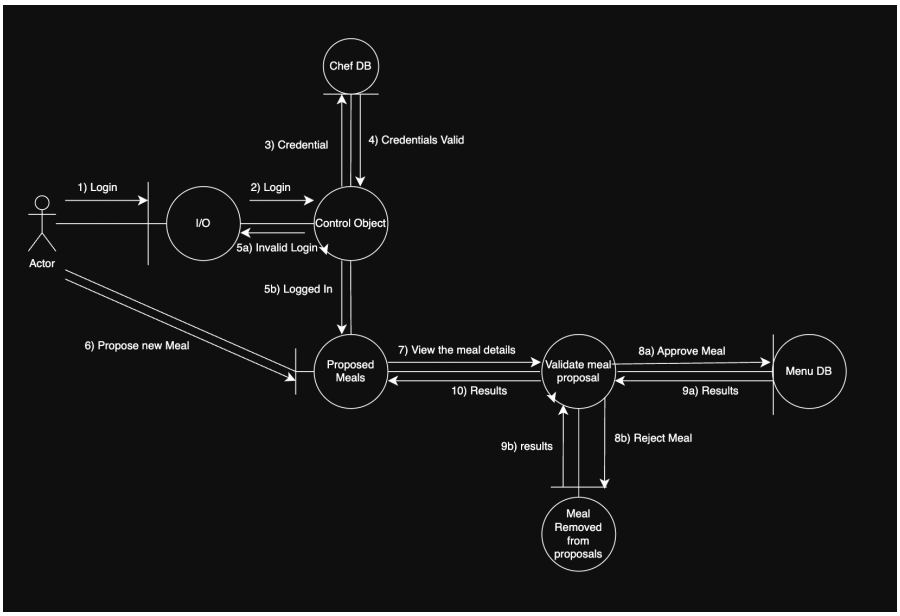
Normal scenario:

1. Chef logs in to admin UI, navigates to menu management.
2. Chef creates a new meal, adds/edits a dish with images, price, category, and inventory counts.
3. Backend validates and updates **`MenuItem`** and **`Inventory in Menu Database`**.



Exceptional scenarios:

- Invalid image upload → reject and show accepted file types and size.
- The manager does not approve of the meal. The manager will see the meal in their manager UI, and reject the meal proposal, meals must be approved by the manager.

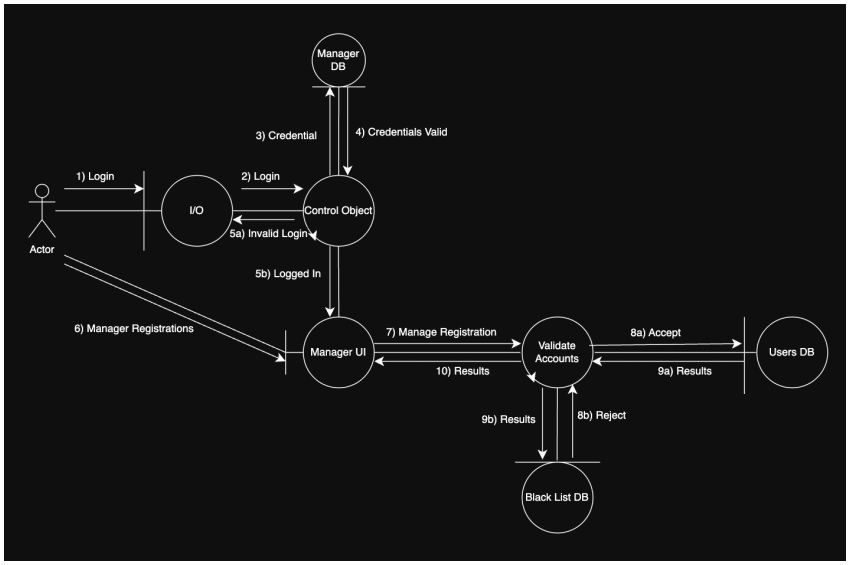


2.5 Manager accepting user registrations

Actors: Manager, Customers

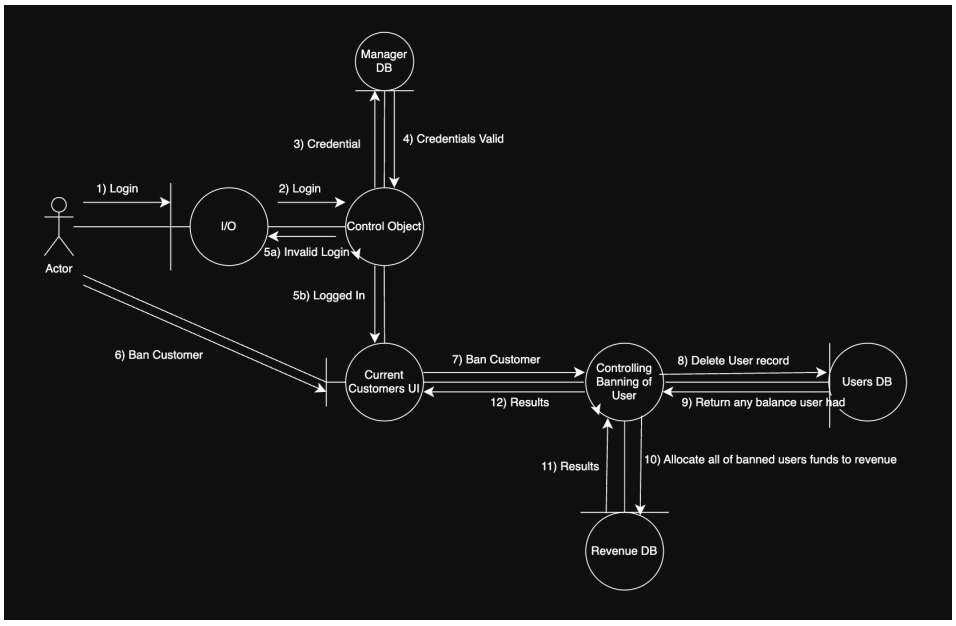
Normal scenario:

1. Manager logs in to the manager UI
2. Manager is able to validate accounts and decide whether or not to accept or reject them



Exceptional scenarios:

- The manager has to ban an existing customer



2.6 AI Chatbot Interactions

Actors: Visitor, Customer, VIP Customer

Normal scenario:

1. User enters a query in chat UI.
2. Client sends query to **`/api/chat`**.
3. Backend checks local KB for a direct answer; if insufficient, forwards to **`RecommendationEngine`** (LLM microservice) with contextual prompt + relevant KB snippets.
4. LLM returns natural response and optionally recommended menu items; chat and recommendation are logged.

Exceptional scenarios:

- LLM unavailable → fallback to local KB answers or canned responses.
- LLM returns unsafe or nonsensical answer → apply post-filtering (safety checks) and regenerate or display safe fallback.
- Long-running LLM response → stream partial responses; show typing indicator.

2.7 Reviews & Complaints (Manager workflows)

Actors: Manager, Chef, Delivery Dude

Normal scenario:

- Customers submit reviews and complaints via UI; data is routed to manager dashboard; manager may issue warnings or adjust user status.

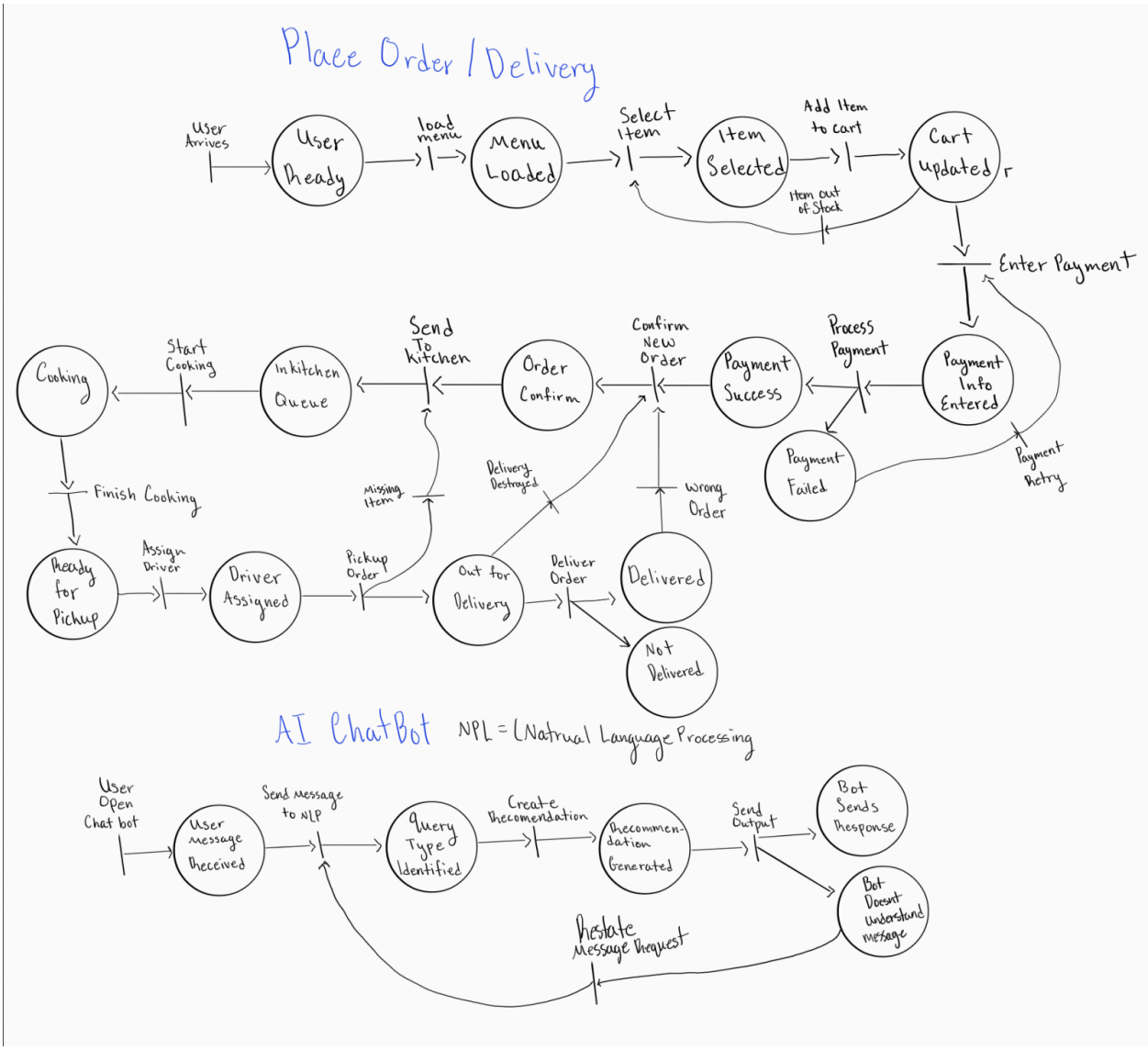
Exceptional scenarios:

- Spam or abusive content → auto-detect using content filters and hide pending review; flag for human review.
- Fraudulent ratings → detect using heuristics and temporarily suspend suspicious accounts.

---

3. Choose 3+ Use Cases — Petri-nets (instead of sequence/class diagrams)

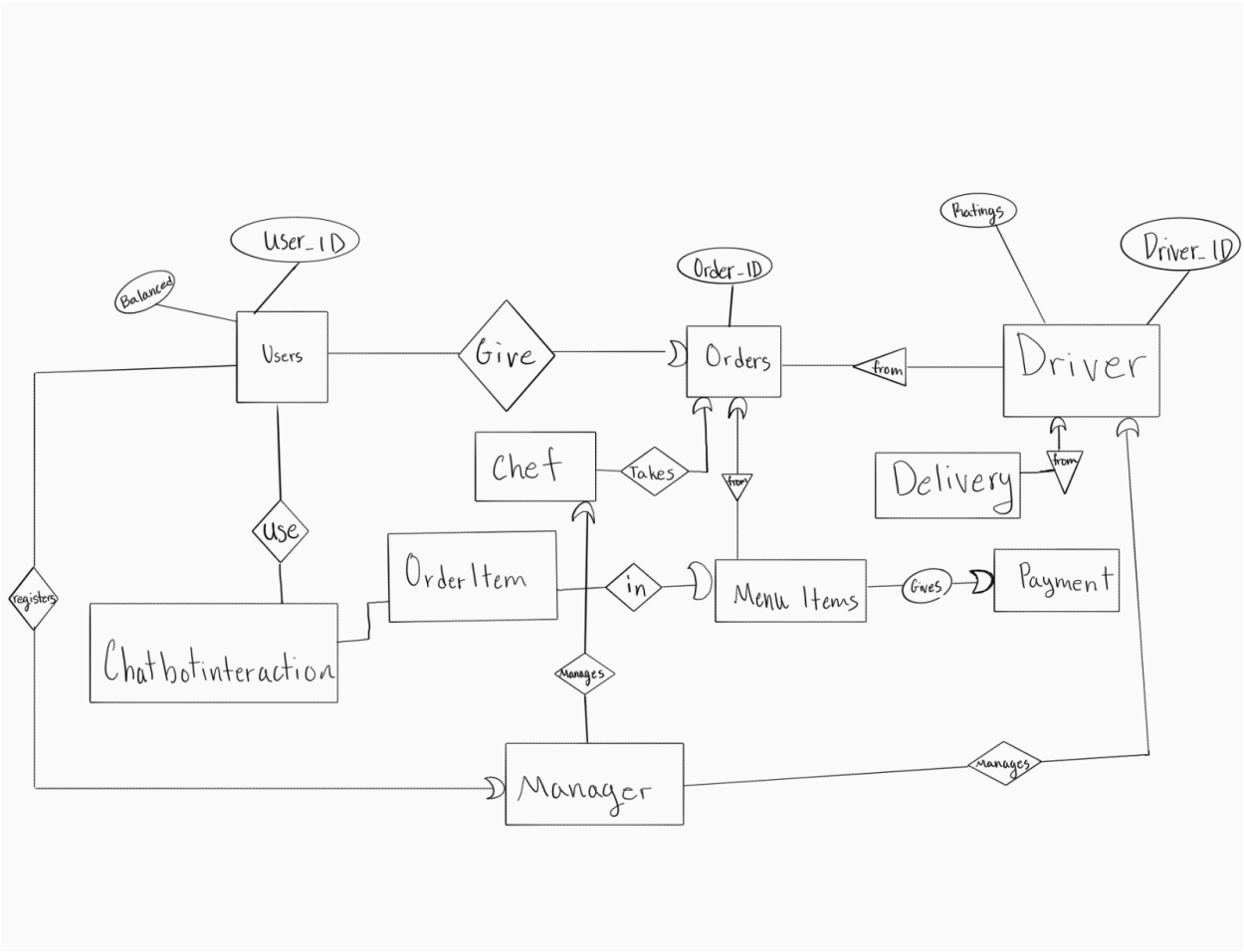
We provide Petri-net descriptions for these three use cases: *Placing Order*, *Delivering Order*, *AI Chatbot Interaction*.



---

4. E-R Diagram for the entire system

Key entities, attributes, and keys (primary keys underlined):



---

5. Detailed design — classes, methods, and pseudocode

Below we list modules and for every public method we provide pseudocode specifying inputs, outputs, and main steps.

5.1 `AuthService`

- **`register(userData)`**
  - Input: **`{name, email, password, phone}`**
  - Output: **`{success, userId, message}`**

Pseudocode:
validate email & password
if emailExists(email) return {success:false, message: 'Email taken'}
hashed = hashPassword(password)

userId = db.create('User', {name,email,hashed,role:'CUSTOMER'})
sendVerificationEmail(email)
return {success:true, userId}

    -

- **`login(email, password)`**
    - Input: email, password
    - Output: JWT token or error

Pseudocode:
user = db.findUserByEmail(email)
if !user return error
if !verify(password, user.hashed) return error
token = jwt.sign({userId: user.id, role: user.role})
return {token}

    -

5.2 `MenuService`

- **`getMenu(filters)`**
    - Input: category, search, sort
    - Output: list of MenuItem objects

Pseudocode:
query = buildQuery(filters)
items = db.query('MenuItem', query)
for item in items: item.rating = computeRating(item.id)
return items

    -

- **`updateMenuItem(itemId, payload)`**
    - Validate chef/manager role
    - Apply optimistic locking (version check)

5.3 `OrderService`

- **`addItemToCart(userId, itemId, qty)`**
    - Input: userId, itemId, qty
    - Output: cart snapshot
    - Pseudocode: validate availability, create-or-update `Cart` record, return cart.
- **`checkout(userId, paymentDetails, idempotencyKey)`**
    - Input: userId, paymentDetails, idempotencyKey
    - Output: orderId or error

Pseudocode:
```
if previouslyProcessed(idempotencyKey) return previousResult
cart = db.getCart(userId)
validate inventory for each item
total = computeTotal(cart)
orderId = db.create('Order', {...status:'PENDING'})
paymentResult = PaymentGateway.charge(paymentDetails, total)
if paymentResult.success:
  db.update(orderId, status:'PLACED', paymentId:paymentResult.id)
  queueKitchen(orderId)
  assignDelivery(orderId)
  return {orderId}
else:
  db.update(orderId, status:'PAYMENT_FAILED')
  return error
```

-

5.4 `RecommendationEngine` (AI microservice)

- `getRecommendations(userId, context)`
  - Input: userId, recentOrders, preferences, localKB
  - Output: list of recommended MenuItem IDs with explanation text

Pseudocode:
```
kb_snippets = localKB.fetchRelevant(context)
prompt = buildPrompt(userPrefs, recentOrders, kb_snippets)
llmResp = llm.call(prompt, settings={maxTokens:400})
filtered = postFilter(llmResp)
parse recommendations -> itemIds + explanation
return {items, explanation}
```

-

Post-filtering must remove hallucinated items (check ids exist in DB) and sanitize content.

5.5 `DeliveryService`

- **`assignDelivery(orderId)`**
  - Search for available delivery users within range and assign.
  - Pseudocode: find nearest **`Delivery`** with status **`AVAILABLE`**; create **`DeliveryAssignment`** and notify.
- **`updateDeliveryStatus(assignmentId, status, location)`** — updates ETA and status transitions.

5.6 `ChatService`

- **`handleMessage(sessionId, message)`**
    - If rule-based KB match -> return immediate answer
    - Else forward to AI microservice and stream response back to client

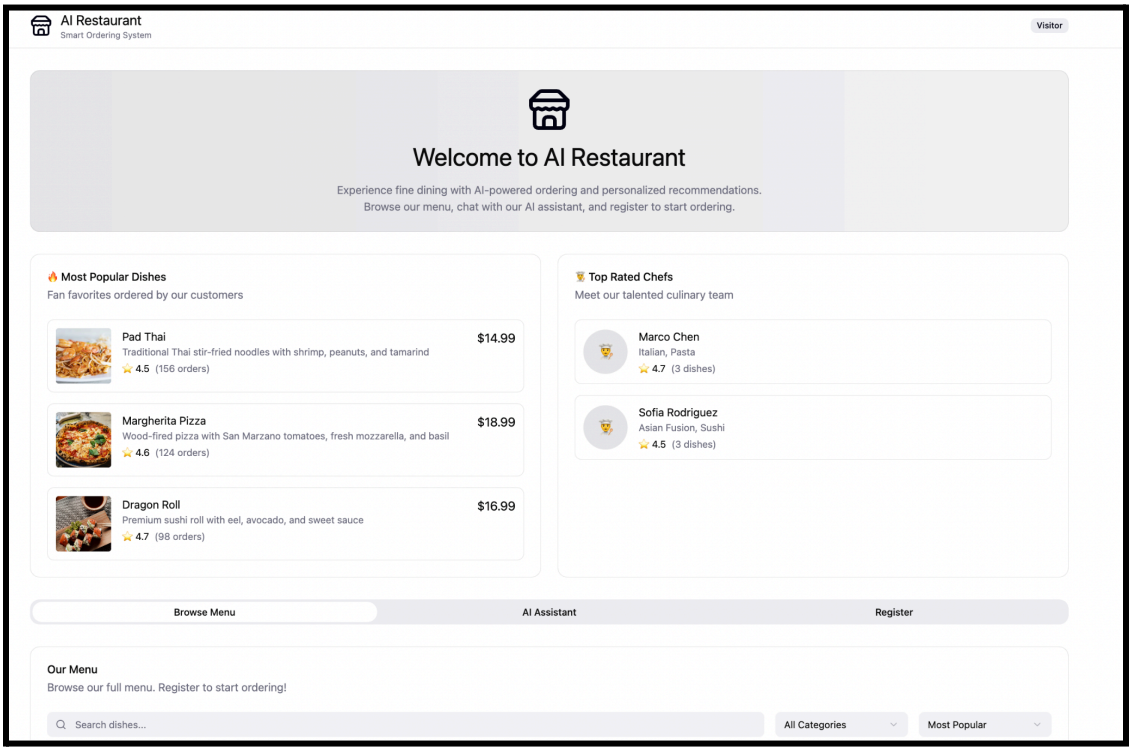
---

6. System screens (GUI) & prototype flow

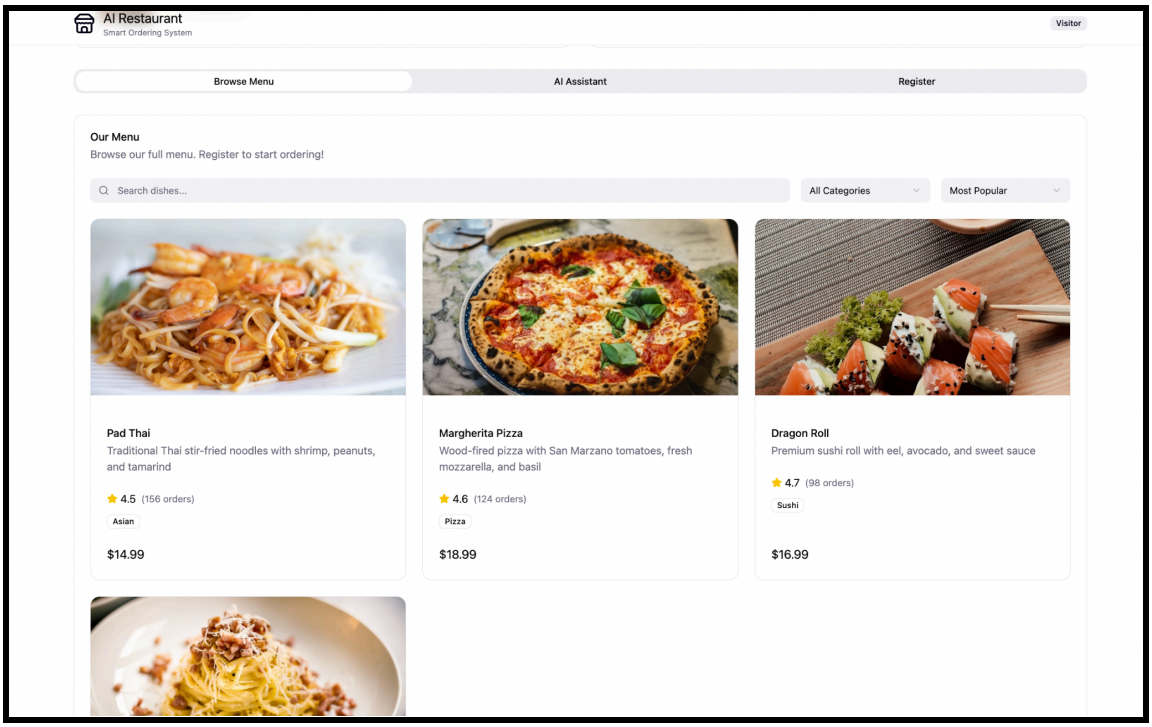
We include mockups (textual) for the major screens and a sample prototype flow for **Checkout + AI-suggested upsell**.

Major pages:

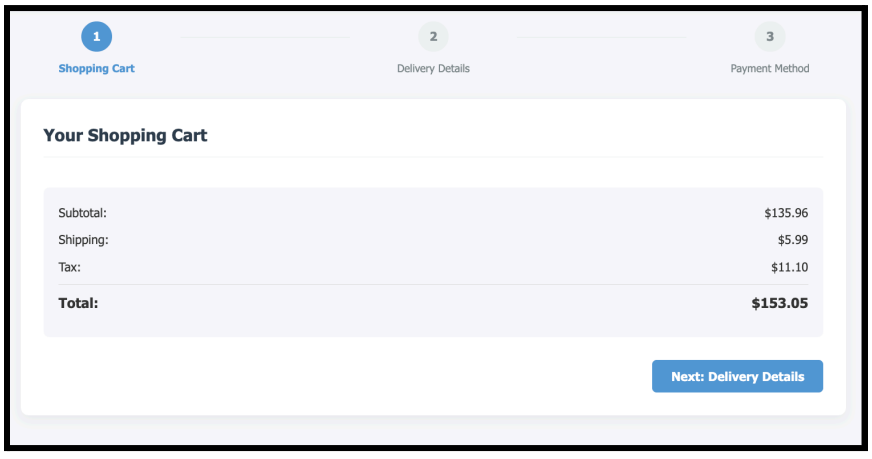
- Home / Landing (featured dishes, search)

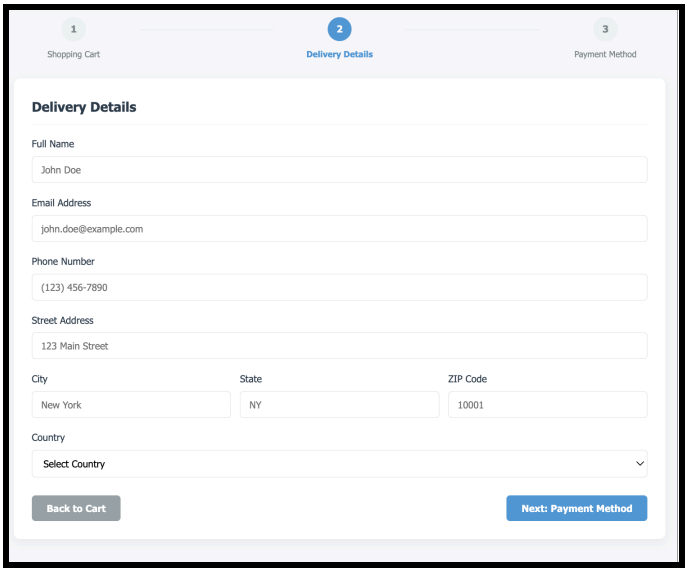


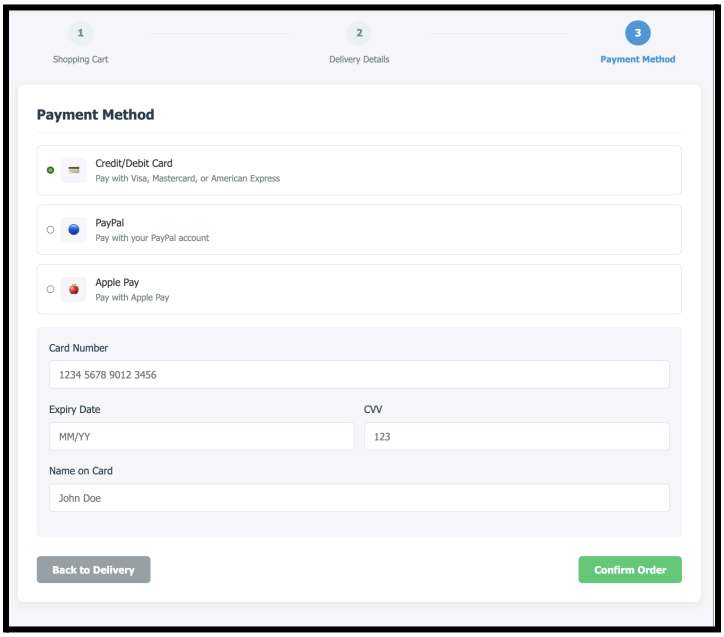
- Menu Listing (filter by category)



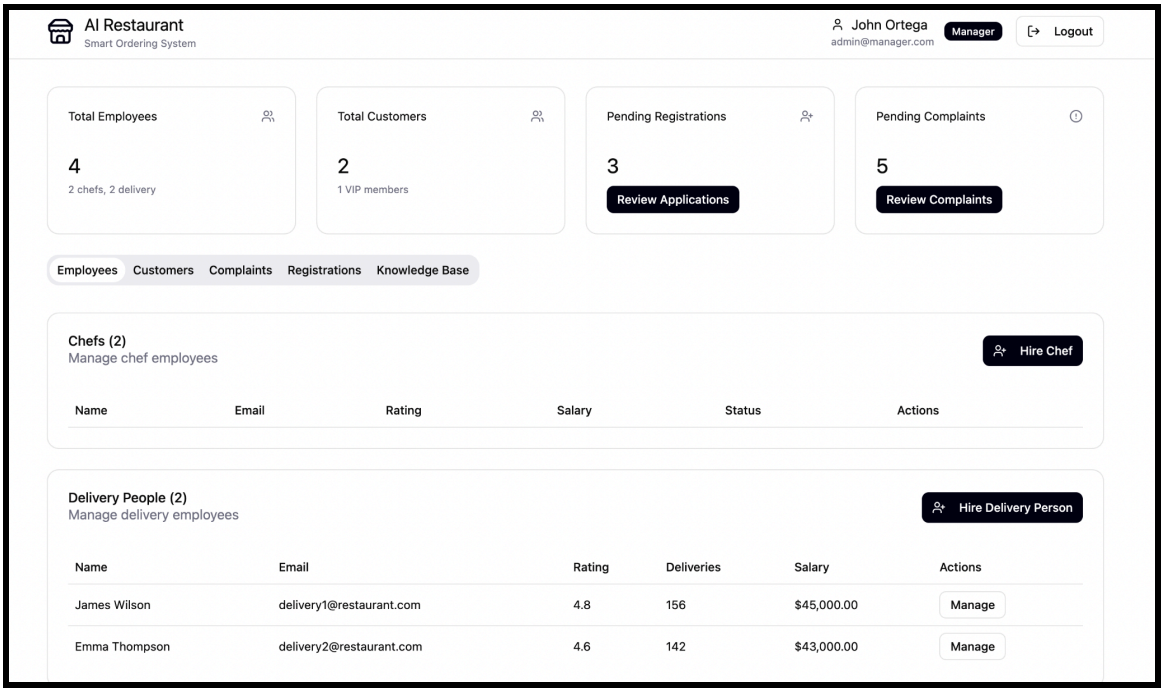
- Cart / Checkout (address, delivery/pickup, payment)



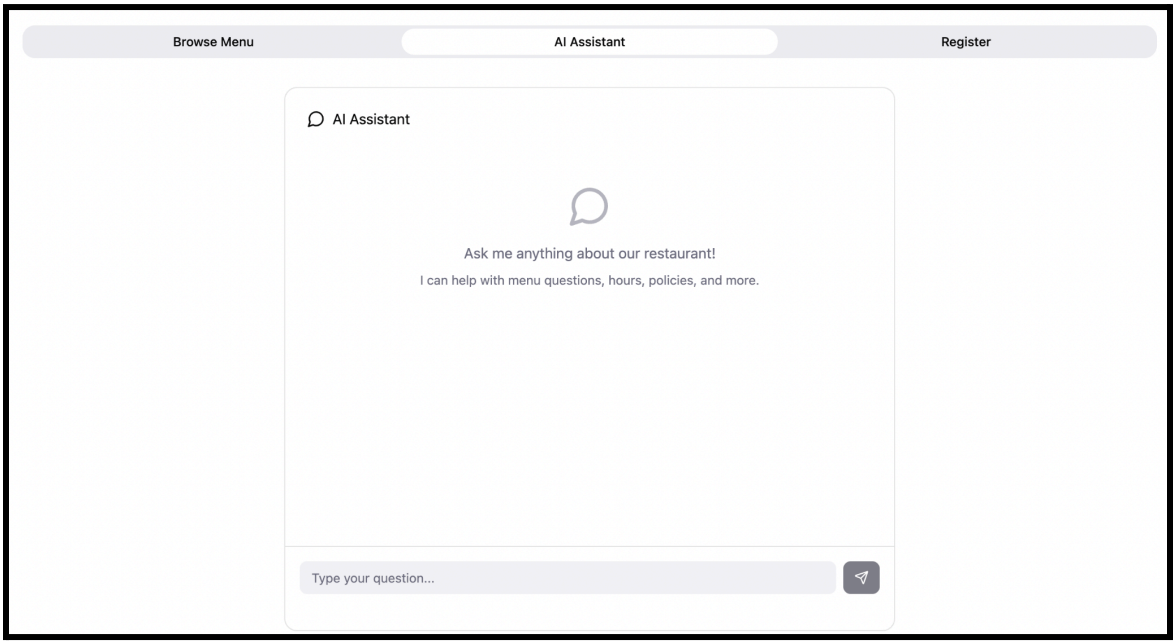




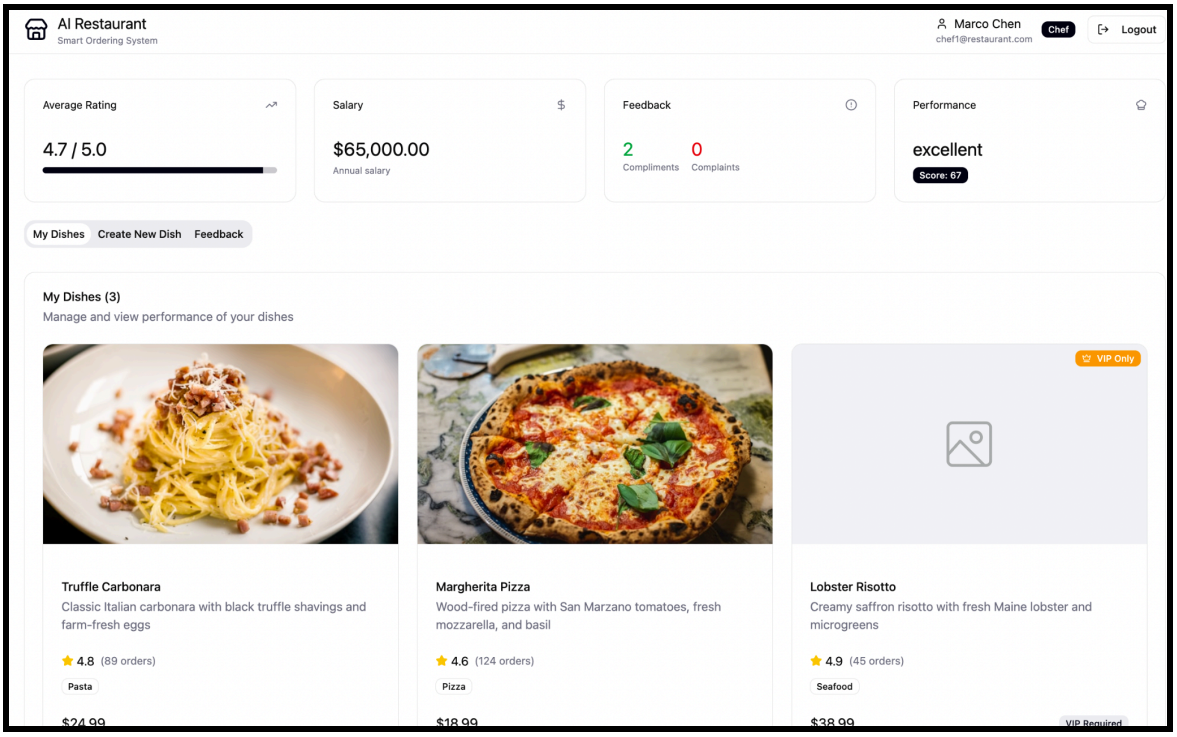
- Admin Dashboard (menu management, orders queue, reviews)



- AI Assistant modal/chat (suggestions & FAQs)



- Chef Dashboard



Prototype flow — Checkout + AI Upsell:

1. User opens cart, clicks checkout.
2. Before finalizing, client calls `RecommendationEngine.getRecommendations(userId, cartContext)`.
3. Engine returns a side-panel suggestion: "Add Garlic Naan? Customers who bought your entree also bought..." with two-click add-to-cart.
4. User accepts suggestion, proceeds to payment, and completes order.

---

7. Memos of group meetings & team-work concerns

- Meeting 2025-10-23: Drafted SRS (Phase I). Action: complete Phase II by Nov 18.
- Meeting 2025-11-08: Decided on Firestore for quick prototyping and an AI microservice using Ollama. Action: prototype recommendation API.
- Concerns: LLM rate limits, handling offline devices, concurrent menu edits, secure storage of payment keys.
- Mitigations: caching, optimistic locking, rate-limited API calls to LLM, role-based access control, PCI-compliant payment gateway.

---

8. Git repository & deliverables

Repository address (placeholder): https://github.com/jtega149/CSc322_Project

---

Appendix: Additional implementation notes

- **Idempotency:** use idempotency keys on checkout API to avoid duplicate orders.
- **Observability:** emit events for **`ORDER_PLACED`, `PAYMENT_SUCCESS`,** `DELIVERY_UPDATE` to a message bus for analytics.
- **Security:** password hashing, JWT expiration, CSRF protections for web sessions, input sanitization for chat inputs.
- **Testing:** unit tests for services, integration tests for checkout flow, end-to-end tests for order lifecycle.

Next steps (team actions)

1. Confirm choice of DB (Firestore vs Cloud SQL).
2. Assign owners for the AI microservice and payment integration.
3. Produce visual exports (PNG) for Collaboration class-diagram and ER diagram if required.
4. Implement a minimal prototype: Menu listing + Checkout + Basic AI suggestion.

End of Phase II: Design Report (draft).