System Architecture & Requirements

Category	Engineering
Created by	João Francisco Labriola
Created time	@January 13, 2025 11:43 AM
Last updated by	João Francisco Labriola
Status	Draft

Executive Summary

▼ 1.1 Project Overview

Apollo Guide is a digital platform designed to transform the dining experience for individuals with dietary restrictions, preferences and allergies, while providing restaurants with an efficient, modern solution for menu transparency. Apollo Guide will connect diners to a web-based platform where they can seamlessly filter menu items based on allergens, diets and ingredients, ensuring a safer and more confident dining experience.

The system is built around the TRAP branding principles:

- **Trust**: Establishing reliability through expert-driven allergen information and partnerships with establishments adhering to strict health and safety standards.
- **Reliability**: Employing robust screening and monitoring processes to ensure accurate and up-to-date menu data.
- **Access**: Empowering customers with dietary needs by providing easy access to trustworthy, dining options.

- **Practicality**: Simplifying the dining process by removing friction and making it effortless for people to identify dishes in seconds.

Apollo Guide's core functionality includes a scanning device available at partner establishments. Diners are directed to an intuitive, visually appealing web interface where they can filter menu items based on selected filters. This process enhances both customer satisfaction and operational efficiency for restaurants by reducing the time staff spend addressing dietary inquiries.

Initially launched as a web-based application, Apollo Guide is designed with scalability in mind. Future iterations will integrate additional features, such as a guide with a large range of restaurants, cross-platform accessibility, and expanded dietary filtering options, to cater to evolving user needs and industry standards. By aligning technology with inclusivity and practicality, Apollo Guide aims to revolutionize dining accessibility for both customers and restaurants.

▼ 1.2 Business Objectives

The primary business objectives of Apollo Guide are centered around creating value for both diners and restaurant partners:

- For Diners: Enhance dining confidence and safety by providing accurate, real-time dietary information and seamless menu filtering capabilities.
- For Restaurants: Increase operational efficiency, reduce liability risks, and improve customer satisfaction through streamlined dietary information management.
- Market Position: Establish Apollo Guide as the leading platform for dietary-conscious dining, capturing a significant market share in the restaurant technology sector.

▼ 1.3 Stakeholders

Key stakeholders in the Apollo Guide ecosystem include:

- Diners with Special Dietary Needs: People who seek safe and enjoyable dining experiences while managing allergies, restrictions, or personal food preferences.
- Restaurant Partners: Establishments that implement Apollo Guide's scanning system and maintain up-to-date menu information.
- Medical Advisor (Co-founder): Expert providing medical oversight and ensuring the platform's alignment with health and safety standards for various dietary conditions.
- Creative Director (Co-founder): Leader responsible for brand vision and marketing strategy development.
- Development Team: Technical professionals responsible for platform development, maintenance, and updates.
- Product Management Team: Professionals overseeing product roadmap, feature prioritization, and user feedback implementation.
- Customer Success Team: Staff dedicated to onboarding and supporting both restaurant partners and end users.

System Overview

▼ 2.1 System Description

Frontend (React App)

- **Framework:** React (Vite or Next.js)
- **Hosting:** AWS Amplify *(or S3 + CloudFront)*
- **CI/CD:** GitHub → Amplify auto-deploy on push
- **Routing:** `apollo-guide.com/restaurantname` (React Router)
- **State:** Initial data fetched once and stored in context/localStorage

---

Backend (Serverless API)

- **Runtime:** AWS Lambda (Node.js)
- **API Gateway:** Handles `/api/restaurant/:name` routes
- **CI/CD:** GitHub → GitHub Actions → Serverless Framework deploys Lambda/API
- **Security:** IAM roles, HTTPS (TLS 1.3), optional Lambda authorizers

---

Database

- **Engine:** Aurora Serverless v2 (PostgreSQL)
- **Data Shape:** One query per restaurant returns all data for the React app
- **Scaling:** Min 0.5 ACUs, configured for aggressive scale-down
- **Connection:** Lambda connects via IAM authentication or secret manager

---

Caching Layer

- **Primary:** AWS CloudFront
  - Caches API responses per `/restaurant/:name`
  - Serves React app globally
- **Optional:** ElastiCache Redis
  - Used for ultra-fast DB response caching (if needed)

Custom Domain

- **Domain:** `apollo-guide.com`
- **Managed via:** Route 53
- **SSL:** Auto-managed via ACM (Amazon Certificate Manager)

Monitoring & Observability

- **CloudWatch Dashboards:**
  - Lambda: invocations, duration, errors
  - API Gateway: 4xx/5xx rates
  - Aurora: ACU usage, connections
- **CloudWatch Alarms:** Notify on error spikes, DB limits, slow responses

---

Request Flow

1. **User opens** `apollo-guide.com/restaurant123`
2. **CloudFront** serves React app from global edge
3. App hits `/api/restaurant/restaurant123`
4. **API Gateway** routes to Lambda
5. Lambda checks CloudFront/Redis cache
   - If cache miss → queries Aurora
   - Result is cached and returned
6. React renders data → user interacts without more DB calls

CI/CD Summary

Layer	Tool	Trigger
Frontend	AWS Amplify	GitHub push
Backend	GitHub Actions + Serverless Framework	GitHub push

Summary

This stack gives you:

- **Fast global access** via CloudFront
- **Serverless auto-scaling** at every layer
- **Low-cost operation** during idle periods
- **Secure, SQL-based DB** with robust JSON handling
- **Zero infrastructure maintenance**

This system architecture ensures a streamlined, reliable, and secure experience for both diners and restaurant operators, meeting the scalability and performance demands of a growing global user base.

▼ 2.2 System Context

Apollo Guide operates within an ecosystem focused on providing accurate menu information to diners through a web-based platform. The system's primary interface is through QR scanning devices placed at participating restaurants, which direct people to a web interface displaying that establishment's menu with comprehensive filtering capabilities.

The system interacts with the following external entities:

- Restaurant Partners: Manual menu and ingredient information updates through the Apollo Guide management interface.
- Cloud Infrastructure: AWS services for hosting, storage, and content delivery.
- Analytics Tools: Integration with tracking and reporting systems to monitor platform usage and performance.

▼ 2.3 Assumptions and Dependencies

Key assumptions and dependencies that underpin the Apollo Guide system include:

Technical Assumptions

- Restaurant Participation: The system assumes active participation from restaurants in maintaining accurate, up-to-date menu and ingredient information.
- Internet Connectivity: The platform depends on reliable internet access at restaurant locations for QR devices scanning and real-time menu access.
- Technical Infrastructure: The system relies on consistent AWS service availability and performance for core functionality.
- Mobile Device Compatibility: people are assumed to have smartphones capable of scanning QR codes and accessing web applications.
- Browser Support: The web application assumes compatibility with modern web browsers.

Business Dependencies

- Restaurant Data Accuracy: Success depends on restaurants providing and maintaining accurate ingredient and allergen information.
- Staff Training: Effective implementation relies on restaurant staff being trained to support and promote the system.
- User Adoption: System success depends on people’s willingness to adopt QR code scanning for menu access.

External Dependencies

- Third-party Services: Reliance on AWS and other cloud service providers for infrastructure and hosting.
- Regulatory Compliance: Dependency on maintaining compliance with food safety and allergen labeling regulations. **
- Industry Standards: Alignment with evolving dietary and allergen classification standards.

Functional Requirements

▼ 3.1 User Requirements

Apollo Guide must meet the needs of two primary user groups: diners and restaurant partners. The functional requirements focus on ensuring a seamless and secure user experience for both groups.

3.1.1 Diners

1. **Menu Access via QR Scanning Devcice**
    - Diners must be able to scan a QR device at participating restaurants to access the platform's web interface.
2. **Dietary Filtering**
    - Diners must have the ability to filter menu items based on allergens, dietary preferences, and specific ingredients.
    - Filtering results must update dynamically based on selected preferences.
3. **Menu Details**
    - The system must display detailed information about each menu item, including name, description, ingredients, allergens, and other relevant dietary labels.
4. **Up-to-date Data**
    - Diners must view accurate and up-to-date menu data that reflects any changes made by the restaurant.
5. **Mobile Compatibility**
    - The web interface must be fully responsive and optimized for mobile devices to ensure accessibility for all diners.
6. **Secure Connection**
    - The platform must ensure secure browsing via HTTPS encryption to protect user data during interaction.

3.1.2 Restaurant Partners

1. **QR Device Integration**
    - Restaurants must be able to generate and deploy unique QR codes for diners to access their specific menus.
2. **User Support**
    - Restaurants could have access to technical support for onboarding, troubleshooting, and system updates.

3.1.3 Administrative Users

1. **System Management**
    - Administrative users must have access to oversee platform operations, including adding or removing restaurants, monitoring system health, and managing user access levels.
2. **Compliance Monitoring**
    - The platform could enable administrative users to verify that restaurants are adhering to health and safety standards in maintaining accurate data.

▼ 3.2 System Features

Apollo Guide’s system features focus on providing a robust, scalable, and user-friendly experience for diners and restaurant partners while ensuring the platform's reliability and security.

3.2.1 QR Device Scanning

1. The system must generate unique QR codes for each participating restaurant to direct people to the corresponding menu page.
2. The QR code must support both static URLs (restaurant-specific pages) and dynamic query parameters for detailed filtering.

3.2.2 Menu Display and Filtering

1. The platform must dynamically display restaurant-specific menus, including item names, descriptions, and details such as ingredients, allergens, and dietary labels.
2. people must be able to apply filters for:
    - Allergens (e.g., gluten, peanuts, shellfish).
    - Dietary preferences (e.g., vegan, halal, keto).
    - Specific ingredient (e.g., exclude garlic or onions).
3. The filtering system must update menu results in real time based on person inputs.

3.2.3 Data Management

1. Administrators must have the ability to add, update, and delete menu items and ingredients through a secure management interface.
2. Changes made by restaurants must reflect on the platform after screening and approval of Apollo Guide and Establishment.

3.2.4 User Authentication and Access Control

1. The system must support different user roles:
    - **Diners**: No login required; direct access to menu filtering.
    - **Administrative Users**: Full access to oversee platform operations and compliance.

3.2.5 Analytics and Reporting

1. Apollo Guide must track analytics for internal use:
    - Menu item popularity.
    - Commonly applied filters and preferences.
    - Interaction metrics (e.g., QR code scans, time spent on the menu page).
2. The platform could allow administrative users to generate compliance reports for restaurant menu data.

3.2.6 Performance and Scalability

1. The platform must handle concurrent traffic from diners accessing menus across multiple restaurants simultaneously.

2. The system must cache frequently accessed data (e.g., popular menus) to minimize server load and improve response times.

3.2.7 Security

1. All communication between people and the platform must be encrypted using HTTPS with SSL certificates.
2. Sensitive data (e.g., menu updates, person credentials) must be securely stored and transmitted.
3. The platform must implement rate limiting and DDoS protection to mitigate abuse.

3.2.8 Cross-Device Compatibility

1. The platform must be optimized for mobile devices, tablets, and desktops (desktop support but in the structure of mobile device) to ensure accessibility for all people.
2. The QR device scanning functionality must work seamlessly with built-in smartphone cameras and third-party scanning apps.

3.2.9 System Monitoring

1. The platform could integrate with monitoring tools (e.g., AWS CloudWatch) to track system health and performance.
2. Alerts must notify administrators of issues such as high latency, downtime, or database errors.

▼ 3.3 Use Cases

The use cases describe the primary interactions between the people and the Apollo Guide system, ensuring that functional requirements align with real-world scenarios.

3.3.1 Use Case: Accessing a Restaurant Menu

- **Actors**: Diner, System
- **Preconditions**:
  - The diner has a smartphone with a QR scanner or camera.
  - The restaurant has set up a unique QR code.
- **Basic Flow**:
  1. The diner scans the QR code at a participating restaurant.
  2. The system retrieves the corresponding menu data based on the QR code.
  3. The platform displays the restaurant’s home page on the diner’s device.
  4. The diner selects view menu.
  5. The platform displays the restaurant’s menus on the diner’s device.
- **Postconditions**:
  - The diner views the menu and optionally applies filters.
- **Exceptions**:
  - QR code fails to load the menu due to connectivity issues or an invalid QR code.

3.3.2 Use Case: Filtering Menu Items

- **Actors**: Diner, System
- **Preconditions**:
  - The diner has accessed the menu via the QR code.
- **Basic Flow**:
  1. The diner selects filters (e.g., allergens, dietary preferences).
  2. The system processes the selected filters.
  3. The menu dynamically updates to show filtered results.
- **Postconditions**:
  - The diner sees menu items that match their preferences.
- **Exceptions**:
  - No menu items match the applied filters.

3.3.5 Use Case: Monitoring System Health

- **Actors**: Administrative User, System
- **Preconditions**:
  - The administrative user has authenticated their account.
- **Basic Flow**:
  1. The administrative user accesses the monitoring dashboard.
  2. They view real-time metrics, such as server performance and error rates.
  3. The system sends alerts for any identified issues.
- **Postconditions**:
  - The administrative user identifies and resolves system performance issues.
- **Exceptions**:
  - Monitoring tools fail to load due to connectivity problems.

```
graph TD
    A((Diner)) --> |Scan QR Code| B[QR Device]
    B --> |Restaurant Route| C[System]
    C --> D[Open Web Page]
    C --> E[Display Menu]
    C --> F[Update Filtered Menu]
    F --> E
    A --> |Select View Menu| C
```

```
A --> |Apply Filters| C

%% Styling
style A fill:#dfe,stroke:#333,stroke-width:0px
style B fill:#dfd,stroke:#333,stroke-width:0px
style C fill:#dff,stroke:#333,stroke-width:0px
style D fill:#dfd,stroke:#333,stroke-width:0px
style E fill:#dfd,stroke:#333,stroke-width:0px
style F fill:#dfd,stroke:#333,stroke-width:0px
```

Non-Functional Requirements

▼ 4.1. Performance Requirements

The Apollo Guide platform must meet the following performance requirements to ensure a smooth and reliable user experience:

4.1.1 Response Time

- Page load times must not exceed 2 seconds under normal operating conditions.
- Menu filtering operations must complete within 500 milliseconds.

4.1.2 Availability

- The system must maintain 99.9% uptime during peak restaurant operating hours.
- Planned maintenance windows should be scheduled during off-peak hours (2 AM - 5 AM local time).

4.1.3 Scalability

- The platform must support up to 10,000 concurrent users without performance degradation.
- Database queries must maintain response times under high load conditions.

▼ 4.2. Security Requirements

Security is a critical aspect of the Apollo Guide platform. The following requirements must be implemented to protect user data and system integrity:

4.2.1 Data Protection

- All sensitive data must be encrypted at rest using industry-standard encryption (AES-256).
- Regular security audits must be conducted to identify and address vulnerabilities.

4.2.2 Access Control

- Role-based access control (RBAC) must be implemented for all administrative functions.
- Authentication must use secure protocols and enforce strong password policies.

4.2.3 Data Transport Security

- All data transmission between server/database and client must use TLS 1.3 encryption.
- Implementation of secure WebSocket connections for real-time data updates.
- API endpoints must enforce HTTPS-only communication.
- Regular monitoring of SSL/TLS certificate validity and automatic renewal.

▼ 4.3. Scalability Requirements

The Apollo Guide platform must be designed with robust scalability features to accommodate growth and varying loads:

4.3.1 Horizontal Scaling

- The system architecture must support automatic scaling of server instances based on demand.
- Load balancing mechanisms must efficiently distribute traffic across available resources.

4.3.2 Database Scaling

- Database architecture must support sharding for distributed data management.
- Implementation of read replicas to optimize query performance under heavy loads.

▼ 4.4. Usability Requirements

The Apollo Guide platform must prioritize user experience through intuitive design and accessibility features:

4.4.1 Interface Design

- The menu interface must be intuitive and require no training or instructions for first-time users.
- System must support zooming for type 1 menus
- Visual hierarchy and typography must ensure optimal readability across all device sizes.

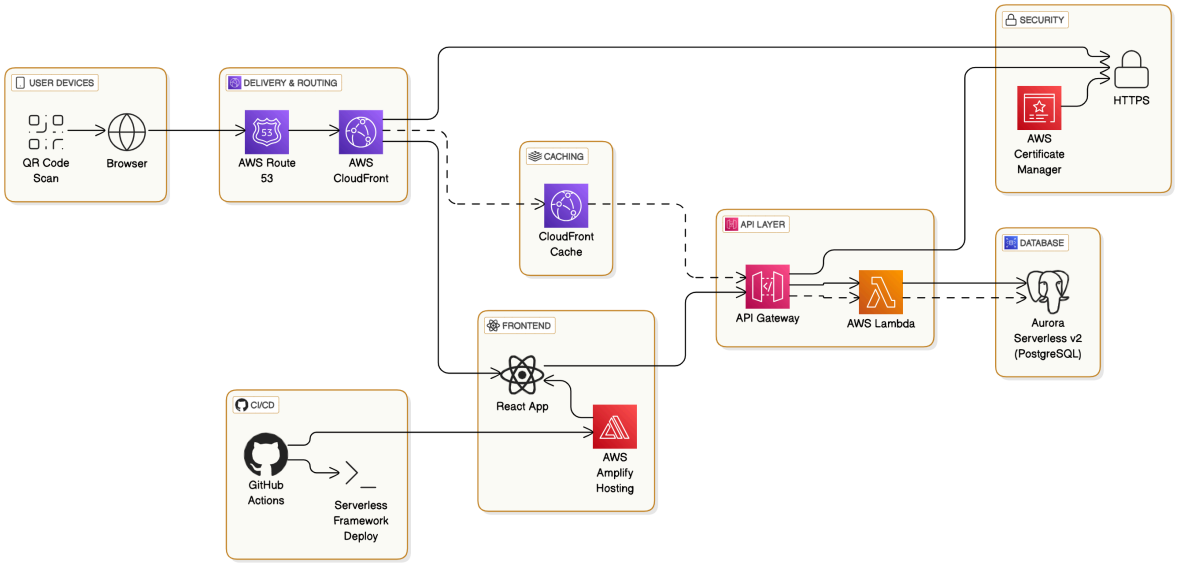
4.4.2 Accessibility

- The platform must comply with WCAG 2.1 Level AA standards.
- Support for screen readers and keyboard navigation must be implemented.

System Architecture

▼ 5.1. High-Level Architecture

The Apollo Guide platform follows a modern, cloud-native architecture designed for scalability, reliability, and maintainability. The system is built using a microservices architecture pattern, with distinct services handling specific business domains such as menu management, user authentication, and analytics. This architecture enables independent scaling of components and ensures system resilience through service isolation.



Main concerns:

Caching - multiple repeated requests will be done by users (one request as app opens and multiple users will be the same as app will be the same for all users coming from one restaurant route)

Routing - accessing restaurant data by the route on the url apollo-guide.com/resutarantname

Security - private access and transport of db data

Edge - for access from multiple regions worldwide

Backend integration to db, with integration to react frontend skeleton populated by db fetched data

Automated CI/CD backend and frontend code from github

Custom domain connection

SQL based db

▼ 5.2. Component Design (Draft)

> **Core Components**

Restaurant Components

- `RestaurantProvider`
  - Manages restaurant data fetching and state
  - Provides context for child components
  - Props: restaurantId
- `RestaurantHeader`
  - Restaurant name, logo, hours
  - Navigation menu
  - Responsive design

    - Props: restaurantData
- `RestaurantInfo`
    - Restaurant details panel
    - Address, contact info
    - Allergen practices/disclaimers
    - Props: restaurantData

Layout Components

- `AppShell`
    - Main wrapper component
    - Handles responsive container width
    - Contains header, main content area
    - Props: children
- `NavigationTabs`
    - Horizontal scrolling tab navigation
    - Active state indication
    - Props: menuSections, activeSection

Menu Components

- `MenuContainer`
    - Manages menu state and filtering
    - Contains MenuView and FilterDrawer
    - Props: restaurantId, initialMenuType
- `MenuView`
    - Displays filtered menu sections and items
    - Handles menu layout
    - Props: menuData, activeFilters
- `MenuSection`
    - Section header with title
    - Grid/list of menu items
    - Props: sectionTitle, items
- `MenuItem`
    - Item name, description, price
    - Allergen indicators

    - Props: name, description, price, allergens, dietary
- `MenuItemDetail`
    - Modal with detailed item view
    - Full ingredient list and allergens
    - Props: itemDetails

Filter Components

- `FilterDrawer`
    - Slide-up/slide-down filter panel
    - Contains filter groups
    - Props: filterOptions, onFilterChange
- `AllergenFilter`
    - Icon-based allergen selection
    - Toggle functionality
    - Props: allergens, selectedAllergens
- `SearchBar`
    - Text input for menu search
    - Clear functionality
    - Props: onSearch, placeholder

UI Elements

- `IconButton`
    - Consistent icon button styling
    - Props: icon, onClick, active
- `AllergenIcon`
    - Standard allergen iconography
    - Props: allergenType, size
- `Badge`
    - Small indicator for dietary preferences
    - Props: type, label

Data Management

Context Providers

- `RestaurantContext`

    - Restaurant information state
    - Hours and location
    - Allergen practices
    - Props: restaurantId
- `MenuContext`
    - Active menu selection
    - Filter states
    - Search query
    - Props: initialMenuType

> **State Types**

```
interface Restaurant {
  id: string;
  name: string;
  logo: string;
  hours: OperatingHours[];
  location: Location;
  allergenPractices: AllergenPractices;
}

interface MenuItem {
  id: string;
  name: string;
  description: string;
  price: number;
  allergens: string[];
  dietary: string[];
  ingredients: string[];
}

interface MenuSection {
  id: string;
  title: string;
  items: MenuItem[];
}
```

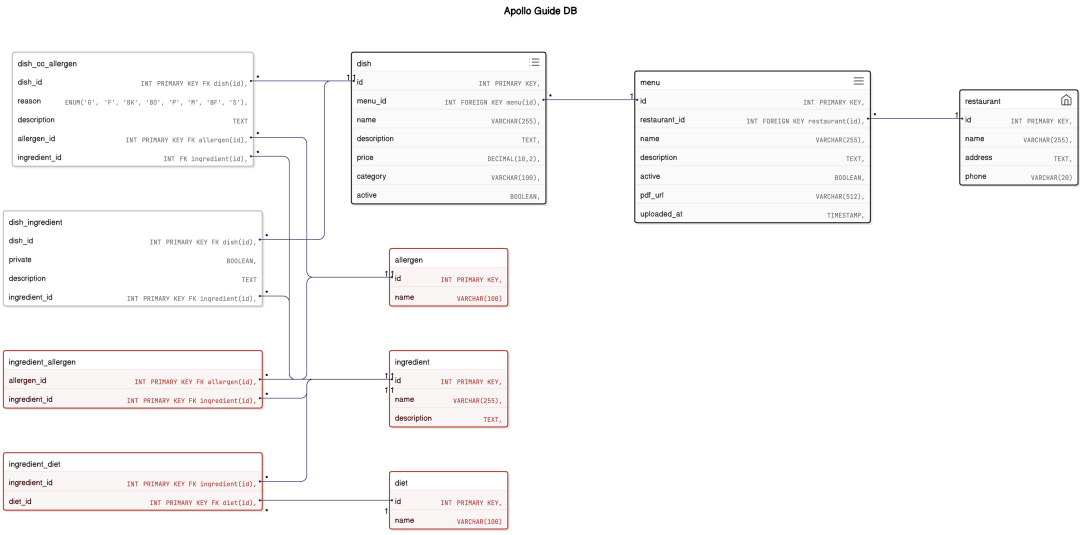
> Component Hierarchy

```
AppShell
├── RestaurantProvider
│   ├── RestaurantHeader
│   ├── RestaurantInfo
│   └── MenuContainer
│       ├── NavigationTabs
│       ├── FilterDrawer
│       │   ├── AllergenFilter
│       │   └── SearchBar
│       └── MenuView
│           ├── MenuSection[]
│           │   └── MenuItem[]
│           └── MenuItemDetail
```

> Implementation Notes

1. Data Flow:
   - Restaurant data flows down from RestaurantProvider
   - Menu filtering/search handled at MenuContainer level
   - Item details managed by MenuView
2. State Management:
   - Restaurant data in RestaurantContext
   - Menu state in MenuContext
   - UI state handled locally
3. User Interactions:
   - Filter changes trigger MenuView updates
   - Item selection opens MenuItemDetail
   - Menu navigation updates active section
4. Styling:
   - Use Tailwind utility classes
   - shadcn/ui components for common elements
   - Responsive design at all levels

▼ 5.3. Data Architecture




```
CREATE DATABASE IF NOT EXISTS apollo_guide_db;
USE apollo_guide_db;

CREATE TABLE restaurant (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    address TEXT NOT NULL,
    phone VARCHAR(20) NOT NULL,
    logo_url VARCHAR(512),
    website_url VARCHAR(512)
) AUTO_INCREMENT = 100000;

-- Menus Table
CREATE TABLE menu (
    id INT AUTO_INCREMENT PRIMARY KEY,
    restaurant_id INT NOT NULL,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    active BOOLEAN NOT NULL,
    menu_overlay JSON,
    uploaded_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (restaurant_id) REFERENCES restaurant(id) ON DELETE CASCADE
) AUTO_INCREMENT = 200000;

-- Dishes Table
CREATE TABLE dish (
    id INT AUTO_INCREMENT PRIMARY KEY,
    menu_id INT NOT NULL,
    name VARCHAR(255) NOT NULL,
```

```
    description TEXT,
    price DECIMAL(10,2) NOT NULL,
    category VARCHAR(100),
    active BOOLEAN NOT NULL,
    FOREIGN KEY (menu_id) REFERENCES menu(id) ON DELETE CASCADE
) AUTO_INCREMENT = 300000;

-- Ingredients Table
CREATE TABLE ingredient (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL COLLATE utf8mb4_bin UNIQUE,
    description TEXT
) AUTO_INCREMENT = 400000;

-- Allergens Table
CREATE TABLE allergen (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL COLLATE utf8mb4_bin UNIQUE
) AUTO_INCREMENT = 500000;

-- Diets Table
CREATE TABLE diet (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL COLLATE utf8mb4_bin UNIQUE
) AUTO_INCREMENT = 600000;

-- Ingredient-Allergen Relationship
CREATE TABLE ingredient_allergen (
    ingredient_id INT NOT NULL,
    allergen_id INT NOT NULL,
    PRIMARY KEY (ingredient_id, allergen_id),
    FOREIGN KEY (ingredient_id) REFERENCES ingredient(id) ON DELETE CASCADE,
    FOREIGN KEY (allergen_id) REFERENCES allergen(id) ON DELETE CASCADE
);

-- Ingredient-Diet Relationship
CREATE TABLE ingredient_diet (
    ingredient_id INT NOT NULL,
    diet_id INT NOT NULL,
    PRIMARY KEY (ingredient_id, diet_id),
    FOREIGN KEY (ingredient_id) REFERENCES ingredient(id) ON DELETE CASCADE,
    FOREIGN KEY (diet_id) REFERENCES diet(id) ON DELETE CASCADE
);

-- Dish-Ingredient Relationship
```

```
CREATE TABLE dish_ingredient (
    dish_id INT NOT NULL,
    ingredient_id INT NOT NULL,
    private BOOLEAN NOT NULL,
    description TEXT,
    PRIMARY KEY (dish_id, ingredient_id),
    FOREIGN KEY (dish_id) REFERENCES dish(id) ON DELETE CASCADE,
    FOREIGN KEY (ingredient_id) REFERENCES ingredient(id) ON DELETE CASCADE
);

-- Dish-Allergen-Cross-Contamination Relationship
CREATE TABLE dish_cc_allergen (
    dish_id INT NOT NULL,
    allergen_id INT NOT NULL,
    reason ENUM('G', 'F', 'BK', 'BO', 'P', 'M', 'BF', 'S') NOT NULL,
    ingredient_id INT NULL,
    description TEXT NULL,
    PRIMARY KEY (dish_id, allergen_id, reason, ingredient_id),
    FOREIGN KEY (dish_id) REFERENCES dish(id) ON DELETE CASCADE,
    FOREIGN KEY (allergen_id) REFERENCES allergen(id) ON DELETE CASCADE,
    FOREIGN KEY (ingredient_id) REFERENCES ingredient(id) ON DELETE SET NULL
);

-- Audit Logging Table
CREATE TABLE audit_log (
    id INT AUTO_INCREMENT PRIMARY KEY,
    table_name VARCHAR(255) NOT NULL,
    row_id INT NOT NULL,
    action_type ENUM('INSERT', 'UPDATE', 'DELETE') NOT NULL,
    old_value JSON NULL,
    new_value JSON NULL,
    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    changed_by VARCHAR(255) NULL
) AUTO_INCREMENT = 700000;

-- Indexing for Performance
CREATE INDEX idx_menu_id ON dish (menu_id);
CREATE INDEX idx_restaurant_id ON menu (restaurant_id);
CREATE INDEX idx_ingredient_id ON ingredient_allergen (ingredient_id);
CREATE INDEX idx_allergen_id ON ingredient_allergen (allergen_id);
CREATE INDEX idx_ingredient_diet ON ingredient_diet (ingredient_id);
CREATE INDEX idx_diet_id ON ingredient_diet (diet_id);
CREATE INDEX idx_dish_ingredient ON dish_ingredient (dish_id, ingredient_id);
```

The database schema above represents the core data structure for the Apollo Guide platform. This design supports efficient querying of menu items, ingredients, allergens, and dietary information while maintaining data integrity through proper relationships and constraints. The schema is optimized for both read and write operations, with consideration for future scalability needs.

Business Domain Constraints

`restaurant`, `menu`, `dish`, `ingredient`, `allergen` and `diet` tables' `id` field must be unique.

`ingredient`, `allergen` and `diet` table's `name` field must be unique.

Tables `ingredient`, `allergen`, `diet`, `ingredient_alergen`, `ingredient_diet` are unique to apollo guide and not to restaurants. Restaurant data refers to these tables but do not affect them in any way.

▼ 5.4 Query Engineering (Requires Testing)

Retrieve Unique List of Non-Private Ingredients for a Restaurant

This query retrieves a unique list of non-private ingredients used across all dishes at a specified restaurant, ordered alphabetically by ingredient name.

```
SELECT JSON_ARRAYAGG(
    JSON_OBJECT(
        'id', i.id,
        'name', i.name
    )
) AS ingredients
FROM (
    SELECT DISTINCT i.id, i.name
    FROM ingredient i
    JOIN dish_ingredient di ON i.id = di.ingredient_id
    JOIN dish d ON di.dish_id = d.id
    JOIN menu m ON d.menu_id = m.id
    WHERE m.restaurant_id = ?
    AND di.private = FALSE
    ORDER BY i.name
) AS ingredient_list;
```

Retrieve Complete Menu Hierarchy with Ingredients, Allergens, and Diets

Retrieves the complete menu hierarchy for a restaurant, including all dishes with their ingredients, allergens, dietary information, and cross-contamination risks. This query provides a structured breakdown of each menu item, displaying both public and private ingredients, potential allergens derived from ingredients, compatible dietary preferences, and any allergens present due to cross-contamination. Results are organized by menu and dish, making it suitable for menu management systems and real-time dietary filtering.

```
WITH dish_allergens AS (
  SELECT
    d.id AS dish_id,
    JSON_ARRAYAGG(
      DISTINCT JSON_OBJECT(
        'id', a.id,
        'name', a.name
      )
    ) AS allergens
  FROM dish d
  JOIN dish_ingredient di ON d.id = di.dish_id
  JOIN ingredient_allergen ia ON di.ingredient_id = ia.ingredient_id
  JOIN allergen a ON ia.allergen_id = a.id
  GROUP BY d.id
),
dish_diets AS (
  SELECT
    d.id AS dish_id,
    JSON_ARRAYAGG(
      JSON_OBJECT(
        'id', diet.id,
        'name', diet.name
      )
    ) AS diets
  FROM dish d
  JOIN dish_ingredient di ON d.id = di.dish_id
  JOIN ingredient_diet id ON di.ingredient_id = id.ingredient_id
  JOIN diet diet ON id.diet_id = diet.id
  GROUP BY d.id
  HAVING COUNT(DISTINCT di.ingredient_id) = (
    SELECT COUNT(DISTINCT ingredient_id)
    FROM dish_ingredient
    WHERE dish_id = d.id
  )
),
dish_ingredients AS (
  SELECT
    d.id AS dish_id,
    JSON_ARRAYAGG(
      JSON_OBJECT(
        'id', i.id,
        'name', i.name,
        'private', di.private,
```

```
            'description', i.description
        )
    ) AS ingredients
  FROM dish d
  JOIN dish_ingredient di ON d.id = di.dish_id
  JOIN ingredient i ON di.ingredient_id = i.id
  GROUP BY d.id
),
dish_cross_contamination AS (
  SELECT
    d.id AS dish_id,
    JSON_ARRAYAGG(
      JSON_OBJECT(
        'allergen_id', a.id,
        'allergen_name', a.name,
        'reason', dca.reason,
        'ingredient_id', dca.ingredient_id,
        'description', dca.description
      )
    ) AS cross_contamination
  FROM dish d
  JOIN dish_cc_allergen dca ON d.id = dca.dish_id
  JOIN allergen a ON dca.allergen_id = a.id
  GROUP BY d.id
)
SELECT JSON_OBJECT(
  'menus', JSON_ARRAYAGG(
    JSON_OBJECT(
      'id', m.id,
      'name', m.name,
      'dishes', (
        SELECT JSON_ARRAYAGG(
          JSON_OBJECT(
            'id', d.id,
            'name', d.name,
            'description', d.description,
            'price', d.price,
            'category', d.category,
            'ingredients', COALESCE(di.ingredients, JSON_ARRAY()),
            'allergens', COALESCE(da.allergens, JSON_ARRAY()),
            'diets', COALESCE(dd.diets, JSON_ARRAY()),
            'cross_contamination', COALESCE(dc.cross_contamination, JSON_ARRAY())
          )
        )
        FROM dish d
```

```
            LEFT JOIN dish_ingredients di ON d.id = di.dish_id
            LEFT JOIN dish_allergens da ON d.id = da.dish_id
            LEFT JOIN dish_diets dd ON d.id = dd.dish_id
            LEFT JOIN dish_cross_contamination dc ON d.id = dc.dish_id
            WHERE d.menu_id = m.id
            AND d.active = TRUE
          )
        )
    )
) AS menu_data
FROM menu m
WHERE m.restaurant_id = ?
AND m.active = TRUE;
```

Duplicate keys update the value

Insert Ingredient Data | *Native*

```
INSERT INTO ingredient (name, description)
VALUES
    ('Tomato', 'Red Tomato'),
    ('Cherry Tomato', 'Red Cherry Tomato')
ON DUPLICATE KEY UPDATE
    description = VALUES(description);
```

Insert Allergen Data | *Native*

```
-- Original

INSERT INTO allergen (name)
VALUES
    ('Egg'),
    ('Milk'),
    ('Soya'),
    ('Wheat'),      -- Gluten
    ('Tree Nuts'),
    ('Peanuts'),
    ('Nuts'),       -- All Nuts
    ('Shellfish'),
    ('Fish'),
    ('Seafood'),    -- Fish & Shellfish
```

```
    ('Sesame'),
    ('Coeliac')     -- Gluten / Wheat
ON DUPLICATE KEY UPDATE
    name = VALUES(name);
```

Insert Diet Data | *Native*

```
-- Original

INSERT INTO diet (name)
VALUES
    ('Vegetarian'),
    ('Vegan'),
    ('Pescatarian'),
    ('Gluten Free'),
    ('Lactose Free'),
    ('Dairy Free'),
    ('Fructose Free')
ON DUPLICATE KEY UPDATE
    name = VALUES(name);
```

Insert Ingredient-Allergen | *Native - Relationship*

```
INSERT INTO ingredient_allergen (ingredient_id, allergen_id)
VALUES
    (400000, 500007), -- Cheese → Dairy
    (400001, 500000), -- Bread → Gluten
    (400002, 500006), -- Salmon → Fish
    (400003, 500003), -- Peanuts → Peanuts
    (400004, 500004), -- Eggs → Eggs
    (400005, 500002), -- Shrimp → Shellfish
    (400006, 500001), -- Soy Sauce → Soy
    (400007, 500008), -- Cashew → Tree Nuts
    (400008, 500009), -- Sesame Seeds → Sesame
    (400009, 500005)  -- Mustard → Mustard
ON DUPLICATE KEY UPDATE
    ingredient_id = VALUES(ingredient_id),
    allergen_id = VALUES(allergen_id);
```

Insert Ingredient-Diet | *Native - Relationship*

```
INSERT INTO ingredient_diet (ingredient_id, diet_id)
VALUES
   (400000, 600001), -- Chicken is not suitable for Vegetarian
   (400000, 600000), -- Chicken is not suitable for Vegan
   (400004, 600000), -- Cheese is not suitable for Vegan
   (400006, 600003), -- Egg is not suitable for Pescatarian
   (400007, 600000), -- Cream (dairy) is not suitable for Vegan
   (400009, 600006), -- Bacon is not suitable for Halal
   (400009, 600005), -- Bacon is not suitable for Kosher
   (400011, 600000), -- Honey is not suitable for Vegan
   (400012, 600004), -- Bread with gluten is not suitable for Gluten-Free
   (400015, 600000)  -- Butter is not suitable for Vegan
ON DUPLICATE KEY UPDATE
   ingredient_id = VALUES(ingredient_id),
   diet_id = VALUES(diet_id);
```

Insert Restaurant Data (Whole) | *Client*

Insert Restaurant Data (Restaurant) | *Client*

```
INSERT INTO restaurant (name, address, phone)
VALUES
   ('The Gourmet Spot', '123 Fine Dining Street, Paris, France', '+33 1 23 45 67 89'),
   ('Ocean Breeze', '456 Seaside Ave, Miami, USA', '+1 305-555-1234')
ON DUPLICATE KEY UPDATE
   name = VALUES(name),
   address = VALUES(address),
   phone = VALUES(phone);
```

Insert Restaurant Data (Menu) | *Client*

```
INSERT INTO menu (restaurant_id, name, description, active, pdf_url)
VALUES
   (100000, 'Lunch Specials', 'A selection of our best lunch dishes', TRUE, 'https://example.co
m/menu-lunch.pdf'),
   (100000, 'Dinner Menu', 'Our signature dinner offerings', TRUE, 'https://example.com/menu-d
inner.pdf')
ON DUPLICATE KEY UPDATE
   name = VALUES(name),
```

```
    description = VALUES(description),
    active = VALUES(active),
    pdf_url = VALUES(pdf_url);
```

Insert Restaurant Data (Dish) | *Client*

```
INSERT INTO dish (menu_id, name, description, price, category, active)
VALUES
    (200000, 'Grilled Chicken Salad', 'Fresh greens with grilled chicken and house dressing', 12.9
9, 'Salads', TRUE),
    (200000, 'Classic Cheeseburger', 'Beef patty with melted cheese and special sauce', 14.50,
'Burgers', TRUE)
ON DUPLICATE KEY UPDATE
    name = VALUES(name),
    description = VALUES(description),
    price = VALUES(price),
    category = VALUES(category),
    active = VALUES(active);
```

Insert Restaurant Data (Dish-Ingredient) | *Client - Relationship*

```
INSERT INTO dish_ingredient (dish_id, ingredient_id, private, description)
VALUES
    (300000, 400000, FALSE, 'Grilled chicken breast'),
    (300000, 400001, FALSE, 'Mixed greens')
ON DUPLICATE KEY UPDATE
    private = VALUES(private),
    description = VALUES(description);
```

Insert Restaurant Data (Dish-CC) | *Client - Relationship*

```
INSERT INTO dish_cc_allergen (dish_id, allergen_id, reason, ingredient_id, description)
VALUES
    (300000, 500000, 'G', 400000, 'Ingredient grilled with Shellfish'),
    (300000, 500001, 'F', NULL, 'Fish fried with Gluten')
ON DUPLICATE KEY UPDATE
    reason = VALUES(reason),
    ingredient_id = VALUES(ingredient_id),
    description = VALUES(description);
```

Implementation Plan

▼ 6.1. Development Phases

1. Prototype Phase (Weeks 1-2)
    - Design prototyping
        - Create wireframes and mockups
        - User flow mapping
        - Stakeholder feedback integration
    - Interactive prototype
        - Build clickable prototype in Figma
        - Test core user journeys
        - Conduct initial user testing

The 2-week prototype phase ensures alignment with user needs and reduces development risks.

1. Foundation Phase (Weeks 3-5)
    - Database schema implementation and testing
        - Create tables, relationships, and indexes
        - Implement data validation and constraints
        - Performance testing with sample data
    - Core API endpoints development
        - Restaurant and menu CRUD operations
        - Ingredient and allergen management
        - Query optimization and caching strategy

The 3-week foundation phase focuses on core data structure and API development.

1. Frontend Development (Weeks 6-9)
    - Component architecture implementation
        - Setup React component library
        - Implement state management architecture
        - Build reusable UI components
    - Restaurant and menu views
        - Responsive layout development
        - Menu navigation and item display
        - Detail view implementations
    - Filter and search functionality

    - Real-time filtering system
    - Search optimization
    - Filter persistence

The 4-week frontend phase ensures proper component architecture while allowing for UI/UX refinement.

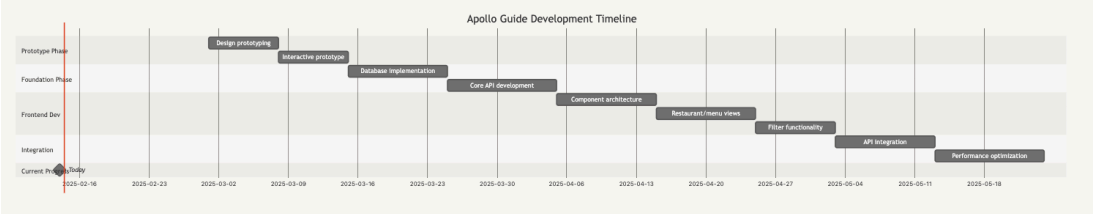
1. Integration Phase (Weeks 10-12)
    - API integration with frontend
        - Implementation of API clients
        - Error handling and recovery
        - Data synchronization
    - Performance optimization
        - Load testing and benchmarking
        - Code splitting and lazy loading
        - Image optimization pipeline

The 3-week integration phase focuses on connecting all components and optimization.

▼ 6.2. Timeline

```
gantt
    title Apollo Guide Development Timeline
    dateFormat  YYYY-MM-DD
    section Prototype Phase
    Design prototyping      :2025-03-01, 1w
    Interactive prototype   :2025-03-08, 1w
    section Foundation Phase
    Database implementation :2025-03-15, 10d
    Core API development    :2025-03-25, 11d
    section Frontend Dev
    Component architecture  :2025-04-05, 10d
    Restaurant/menu views   :2025-04-15, 10d
    Filter functionality    :2025-04-25, 8d
    section Integration
    API integration         :2025-05-03, 10d
    Performance optimization:2025-05-13, 11d

    section Current Progress
    Today :milestone, 2025-02-14, 0d
```



The timeline above shows the project phases starting from March 2025:

- Prototype Phase: March 1 - March 14 (2 weeks)
- Foundation Phase: March 15 - April 4 (3 weeks)
- Frontend Development: April 5 - May 2 (4 weeks)
- Integration Phase: May 3 - May 24 (3 weeks)

Total project duration: 12 weeks

```
flowchart TD
    subgraph Prototype["Prototype Phase (2 weeks)"]
        A["Project Start"] --> B1["Design Prototyping"]
        B1 --> B2["🖱️ Interactive Prototype"]
    end

    subgraph Foundation["Foundation Phase (3 weeks)"]
        C1["Database Implementation"]
        C2["Core API Development"]
    end

    subgraph Frontend["Frontend Development (4 weeks)"]
        D1["Component Architecture"]
        D2["Restaurant/Menu Views"]
        D3["Filter Functionality"]
    end

    subgraph Integration["Integration Phase (3 weeks)"]
        E1["API Integration"]
        E2["Performance Optimization"]
    end

    B2 --> Foundation
    C1 --> C2
    C2 --> Frontend
    D1 --> D2
    D2 --> D3
    D3 --> Integration
    E1 --> E2
```

```
    E2 --> F["Project Completion"]

    style Prototype fill:#bbf,stroke:#333,stroke-width:1px
    style Foundation fill:#bbf,stroke:#333,stroke-width:1px
    style Frontend fill:#bbf,stroke:#333,stroke-width:1px
    style Integration fill:#bbf,stroke:#333,stroke-width:1px
```

▼ 6.3. Resource Requirements

The following resources will be required for successful project implementation:

- Development Team
  - J ;)
- Infrastructure
  - Cloud hosting environment (AWS)
  - CI/CD pipeline
  - Development and staging environments
- Tools and Software
  - Version control (Git)
  - Project management software
  - Design tools (Figma)