SAHYADRI
COLLEGE OF ENGINEERING & MANAGEMENT
An Autonomous Institution
MANGALURU

Mini Project

Database Management System Lab – CS5227L5C

"Platora-Online Food Delivery System"

Name and USN:

1.	**Ajith Goveas**	**4SF23CS014**
2.	**Sahana S Madival**	**4SF23CS181**

Branch/Section: CSE / 5A

Faculty Incharge:

Ms. Chaithra S

Assistant Professor

Department of Computer Science & Engineering

SAHYADRI
COLLEGE OF ENGINEERING & MANAGEMENT
An Autonomous Institution
MANGALURU

Department of Computer Science & Engineering

CERTIFICATE

This is to certify that the mini project work entitled **"Platora-Online Food Delivery System"** has been carried out by **Ajith Goveas (4SF23CS014) and Sahana S Madival (4SF23CS181)** the bonafide students of Sahyadri College of Engineering & Management in partial fulfillment of the requirements for the V semester of Bachelor of Engineering in Computer Science and Engineering of Visvesvaraya Technological University, Belagavi during the year 2025 - 26. It is certified that all suggestions indicated for Internal Assessment have been incorporated in the report deposited in the departmental library. The project report has been approved as it satisfies the academic requirements in respect of project work prescribed for the said degree.

Course Coordinator **Ms. Chaithra S** Assistant Professor	**Dr. Mustafa Basthikodi** Professor & Head of Department

Name and Signature with Date

Examiner 1: _____

Examiner 2: _____

SAHYADRI
COLLEGE OF ENGINEERING & MANAGEMENT
An Autonomous Institution
MANGALURU

Department of Computer Science & Engineering

DECLARATION

We hereby declare that the entire work embodied in this Mini Project Report titled **"Platora-Online Food Delivery System"** has been carried out by us at Sahyadri College of Engineering & Management, Mangaluru under the supervision of **Ms. Chaitra S,** in partial fulfillment of the requirements for the V semester of **Bachelor of Engineering** in **Computer Science and Engineering**. This report has not been submitted to this or any other University for the award of any other degree.

Ajith Goveas (4SF23CS014)

Sahana S Madival(4SF23CS181)

ABSTRACT

Platora is a full-fledged online food delivery system designed to streamline the interaction between customers, restaurants, delivery agents, and administrators. The platform enables customers to browse restaurants, explore menus, place food orders, make secure payments, track order and delivery status in real time, and provide ratings and reviews. Restaurants can efficiently manage their profiles, update menus, monitor incoming orders, and analyse their daily sales and performance. Delivery agents receive automatically assigned delivery tasks, update delivery progress, and view their earnings. Administrators oversee the entire platform with tools for user management, restaurant approval, system monitoring, and analytics.

The core of Platora is supported by a robust PostgreSQL database designed using normalization principles to reduce redundancy and maintain data integrity. The system incorporates advanced DBMS components such as views, stored procedures, triggers, and indexes to automate workflows and improve performance. For example, stored procedures handle order creation, invoice generation, and delivery assignment, while triggers automatically update timestamps and restaurant ratings. Views are used for generating aggregated information like order history, customer spending, restaurant performance, and delivery efficiency.

Built using modern technologies such as Next.js, TypeScript, Node.js, Express, and Tailwind CSS, the system ensures a scalable and user-friendly experience. By combining a well-structured relational database with a dynamic full-stack architecture, Platora demonstrates how DBMS concepts can be applied to real-world applications, offering a reliable, efficient, and scalable solution for online food ordering and delivery management.

Keywords: Online food delivery system, PostgreSQL database, order management, restaurant management, delivery tracking, stored procedures, triggers, views, role-based access control.

TABLE OF CONTENT

CHAPTER 1
INTRODUCTION

1.1 Objectives

1. To design and implement a relational database for managing restaurants, customers, orders, and deliveries.
2. To apply constraints, relationships, and normalization techniques ensuring data integrity.
3. To practice SQL queries, views, triggers, and stored procedures for real-time operations.
4. To integrate a simple front-end with the database for practical usability.
5. To provide role-based access for customers, restaurants, and delivery staff.

1.2 PostgreSQL

PostgreSQL is used as the core database system in the Platora Online Food Delivery Platform, handling all essential data such as users, restaurants, menu items, orders, deliveries, payments, and reviews. It provides a reliable and structured way to store information using primary keys, foreign keys, constraints, and indexes, ensuring high data integrity and accurate relationships between entities. PostgreSQL also supports complex transactional operations, which is crucial for order placement, delivery assignment, and payment processing, allowing all steps to occur safely without data conflicts. Advanced features like stored procedures and triggers automate important tasks such as generating invoices, updating order status, and recalculating restaurant ratings in real time. Additionally, PostgreSQL views are used to produce analytics, including daily sales, customer spending, and delivery performance. These features make the system efficient, scalable, and capable of handling multiple users simultaneously, making PostgreSQL an ideal choice for a real-time food delivery application.

1.3 TypeScript

TypeScript is used throughout the project to make the code more reliable, structured, and easier to maintain. With its strong typing system, TypeScript reduces runtime errors by catching mistakes during development. This is especially important for handling critical operations like adding menu items, creating orders, assigning deliveries, or validating user inputs. TypeScript ensures consistent data structures across backend APIs, frontend components, and database interactions, significantly improving the stability and predictability of the entire system.

1.4 Next.js

Next.js is used as the main frontend framework for Platora, providing fast, scalable, and SEO-friendly web pages. It enables server-side rendering (SSR) and static page generation, which helps the application load restaurant lists, menus, and order details more efficiently. Next.js also handles navigation, routing, and API routes, making it easier to build features such as authentication, order tracking, user dashboards, and restaurant management within a single framework. Its optimized performance and file-based routing system make the UI smooth and responsive, creating a seamless experience for customers, restaurants, and delivery agents**.**

1.5 Node.js

Node.js powers the backend of Platora, acting as the runtime environment that executes the server-side logic. It handles user authentication, restaurant operations, order creation, payment processing, review submissions, and delivery updates. Node.js allows asynchronous and non-blocking operations, making it ideal for real-time features such as order status updates and live delivery tracking. Its performance and scalability make it capable of handling multiple users and requests simultaneously, which is essential for an online food delivery platform.

1.6 Express.js

Express.js is the web application framework used on top of Node.js to build the backend APIs of Platora. It simplifies the creation of endpoints for authentication, menu management, order processing, delivery handling, and admin functionalities. Using Express, the backend organizes routes, controllers, and middleware in a clean and modular structure. Express also handles validation, error responses, JWT-based authentication, and PostgreSQL database communication, making backend development faster, more organized, and maintainable.

1.7 Tailwind CSS

Tailwind CSS is used to design the user interface of Platora with a modern and responsive layout. It provides utility-first styling that allows rapid UI development without writing long CSS files. Tailwind helps in creating clean and visually appealing components such as restaurant cards, menu grids, navigation bars, buttons, forms, and order summaries. It ensures consistent styling across the platform and makes the frontend fully responsive across devices. Tailwind CSS significantly speeds up UI development while maintaining a professional and cohesive design.

CHAPTER 2
SYSTEM ANALYSIS

2.1 Existing System

In many small and local restaurants, the food ordering process is still managed manually. Customers usually place their orders by physically visiting the restaurant or by calling on the phone. While this method may seem simple, it often leads to several operational challenges and inefficiencies, especially when customer volume increases. Without a digital platform, restaurant staff must depend on handwritten notes, verbal communication, or basic registers—all of which are prone to human error.

Drawbacks of the Existing Manual System:

- Absence of a centralized platform: There is no unified system to store customer details, menu information, or order records. This makes it difficult to maintain consistency and retrieve previous order information.
- High chances of miscommunication: Orders communicated verbally over phone calls can be misheard or misunderstood. This may result in incorrect orders, wrong quantities, or incomplete requests.
- Increased possibility of errors: Manual entry increases the likelihood of mistakes such as missing orders, duplicating orders, mixing up customer details, or delivering to the wrong address.
- Lack of reporting and analytics: Restaurants cannot analyse sales performance, best-selling items, peak order times, or customer behaviour. This limits their ability to make informed decisions and improve business operations.

2.2 Proposed System

The proposed Platora Online Food Delivery System is designed to overcome the drawbacks of traditional food ordering methods by introducing a fully digital and automated platform. Instead of relying on phone calls or in-person communication, the system integrates all major components of the food ordering process—customers, restaurants, menus, orders, payments, and deliveries—into one unified application. This creates a smooth and organized workflow that improves accuracy, reduces delays, and enhances customer satisfaction.

Features of the Proposed System:

- Centralized database: The system maintains all information related to customers, restaurants, menus, orders, payments, and delivery agents in a single, well-structured PostgreSQL database. This ensures consistency, easy retrieval, and efficient data management.

- Digital menu browsing and online ordering: Customers can view updated menus, check item details, and place orders directly through the online platform, reducing the need for phone calls or manual note-taking.
- Role-based authentication: Users are assigned specific roles such as customer, restaurant owner, admin, or delivery agent. Each role has defined permissions, ensuring secure access and preventing unauthorized actions.
- User-friendly interface for management: Restaurant owners can easily update menus, mark items as available or unavailable, monitor incoming orders, and manage delivery assignments. Customers can track order history, while administrators can oversee the overall system.
- Secure, scalable, and fast processing: The structured database design ensures quick query execution, and the use of Node.js with PostgreSQL makes the system scalable, secure, and suitable for handling increasing users and orders.
- Automated order history and reporting: Order information is stored automatically, allowing the system to generate sales reports, customer trends, and menu performance analytics in the future.

Overall, the proposed system provides a complete and modern solution for online food ordering, improving communication, reducing manual errors, and offering a more convenient experience for all users involved.

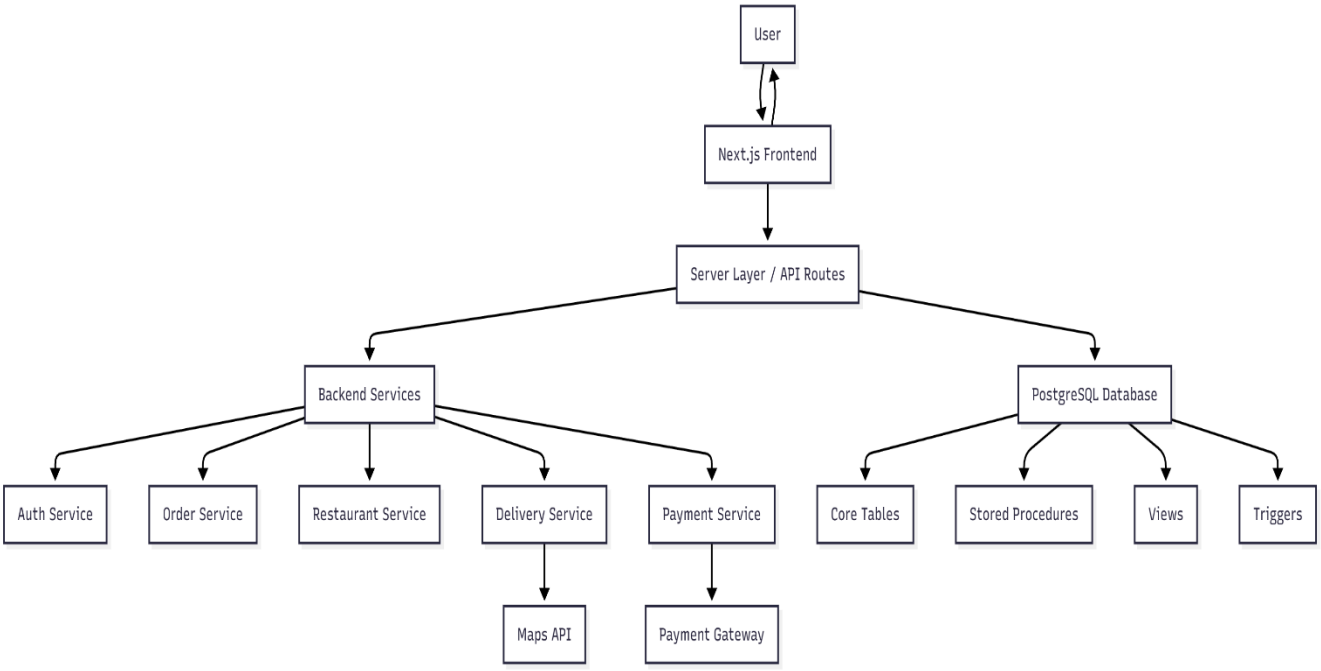

Figure 2.2.1 Flow Diagram

2.3 Feasibility Study

Aspect	Status	Rationale
Technical Feasibility	Highly Feasible	Implemented using standard SQL and PostgreSQL (proven transactional technology).
Operational Feasibility	Highly Feasible	Directly solves concurrency and data freshness problems; integrates with modern web backends.
Economic Feasibility	Feasible	PostgreSQL is open-source, minimizing initial software costs.

CHAPTER 3
SYSTEM DESIGN

3.1 ER Diagram

The ER Diagram serves as the conceptual blueprint, visually modeling all major data entities and the relationships between them. This abstraction ensures a clear, unambiguous representation of the system's data structure before implementation.

- **Users:** Users store customer, restaurant-owner, and delivery-agent details and can place multiple orders.
- Restaurants**:** Restaurants are owned by a user and can receive many orders and contain many menu items.
- Menu Items**:** Menu items belong to a single restaurant and are referenced in multiple order items.
- Orders: Orders are placed by users, belong to restaurants, and connect to order items, invoices, payments, and deliveries.
- Order Items**:** Order items link each order to specific menu items with quantity and price details.
- Invoices**:** Each order generates exactly one invoice containing billing information.
- Payments: Payments record how an order was paid and link directly to the corresponding invoice.
- Deliveries: Deliveries track which delivery agent handles an order
- Reviews: Stores customer ratings and comments for restaurants.
- Delivery_Agents**:** Stores additional profile details for users who work as delivery agents.
- Restaurant_Requests**:** Stores restaurant registration requests submitted by owners for admin approval.

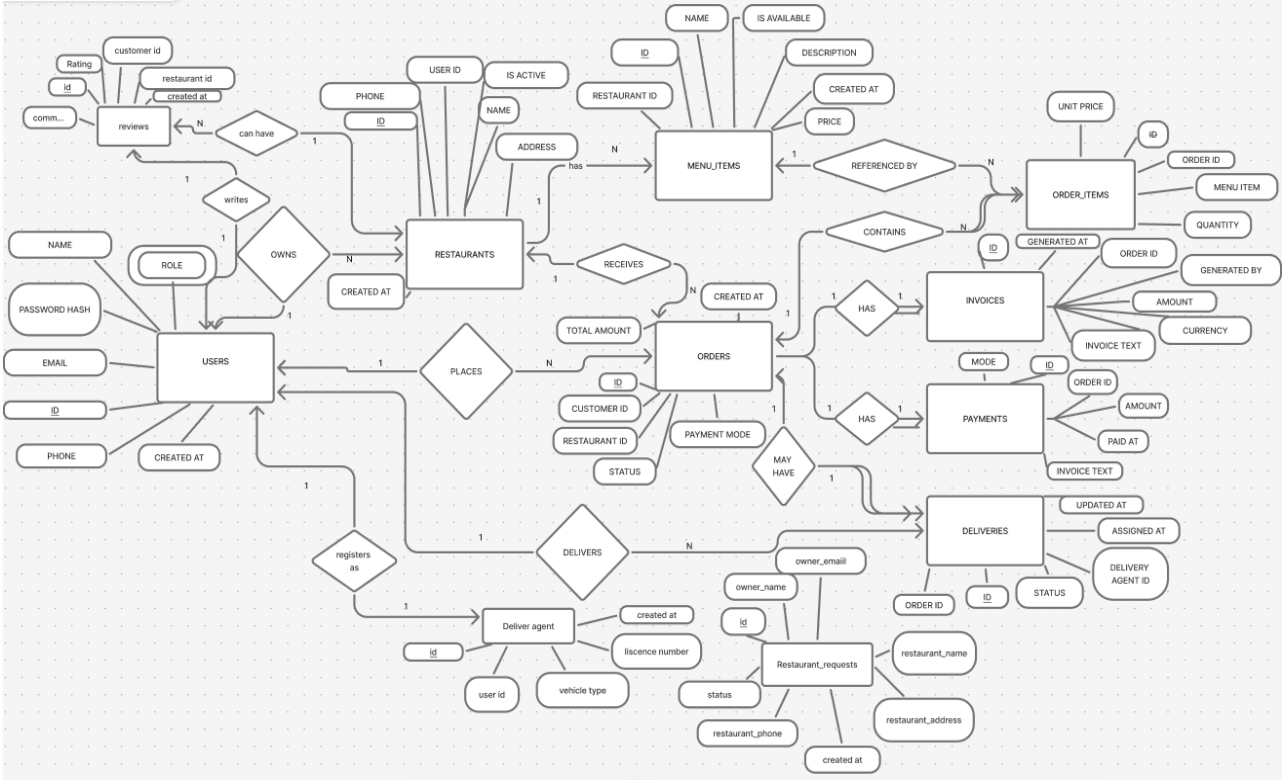

Figure 3.1.1 ER Diagram

3.2 Schema Diagram

The Relational Schema is the physical implementation resulting from the ER Diagram, defining the specific tables, columns, and rules within the PostgreSQL database. This structure strictly enforces data integrity and normalization principles.

- Primary Key–Foreign Key Structure: Each entity is stored as a separate table with a Primary Key, and Foreign Keys link related tables to enforce referential integrity.
- UNIQUE on Menu Items: A unique constraint on (restaurant_id, name) prevents duplicate menu items in the same restaurant.
- UNIQUE on Reviews: A unique constraint on (user_id, restaurant_id) ensures each customer can review a restaurant only once.
- Normalization: All tables follow 3NF to minimize redundancy and maintain a clean, scalable database design

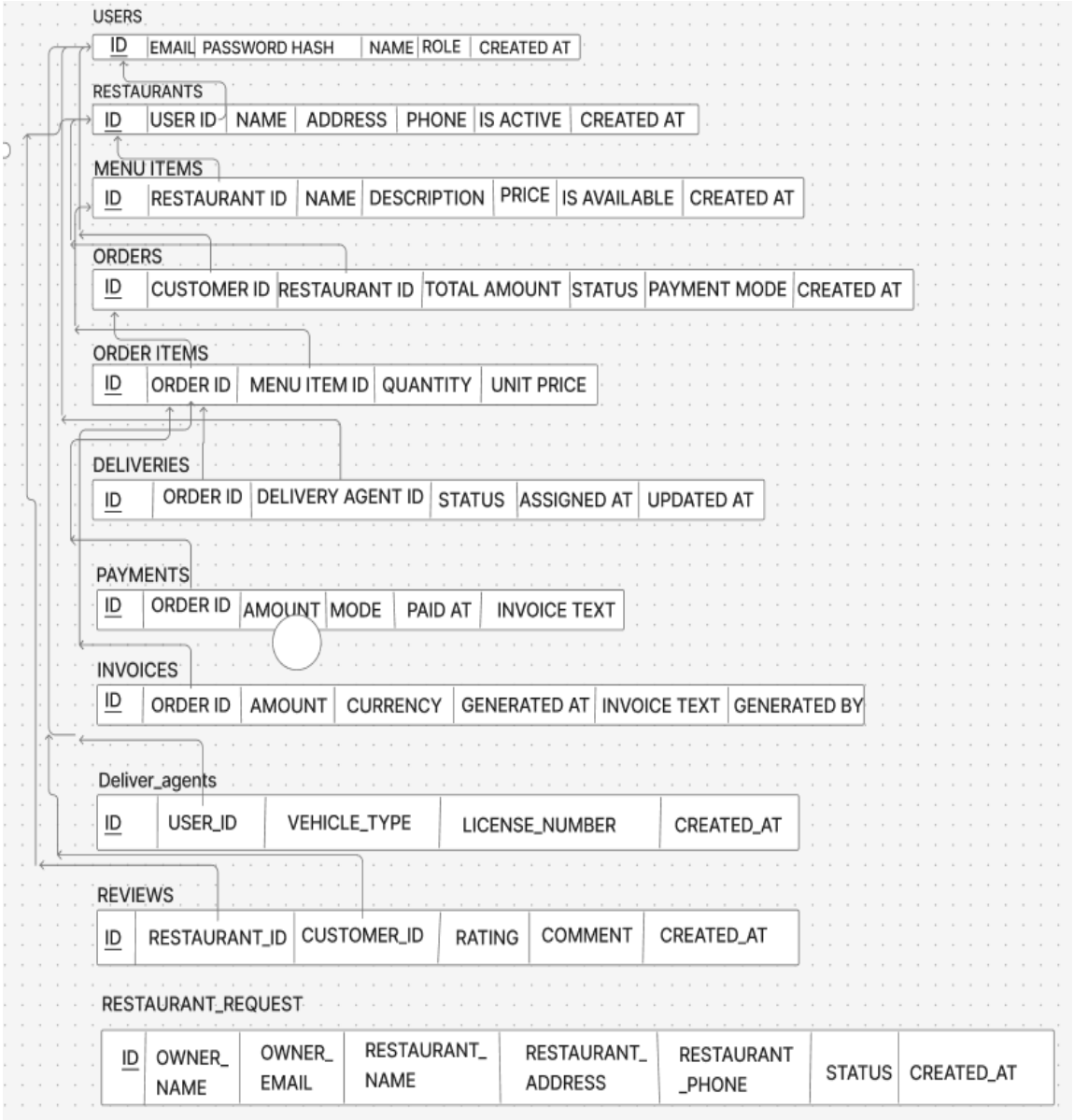

Figure 3.2.1 Schema Diagram

CHAPTER 4
NORMALIZATION

4.1 First Normal Form (1NF)

The design ensures atomicity, meaning every column in each table contains only a single, indivisible value. There are no repeating groups or multi-valued attributes; for example, the order_items table stores each ordered item as a separate record instead of keeping multiple items in a comma-separated list inside the orders table. This guarantees that the data is organized in a clean tabular structure, making queries simpler, faster, and more reliable for the database engine.

- Atomic Values: Each attribute holds only one value per cell, such as a single item_price or quantity, avoiding combined or list-type entries.
- No Repeating Groups: Multiple items in an order are stored as individual rows in the order_items table, ensuring each row represents one unique item within that order.
- Unique Primary Keys: Every table includes a primary key that uniquely identifies each row, preventing duplication and maintaining row-level integrity.

4.2 Second Normal Form (2NF)

Second Normal Form eliminates partial dependencies by ensuring that every non-key attribute depends on the whole primary key and not just part of it. In the Platora system, tables that use composite keys, such as order_items (which depends on both order_id and menu_item_id), store only attributes that relate to the full key. Any information that does not depend on the complete key is moved to its appropriate table. This prevents redundancy and keeps data properly organized.

- No Partial Dependencies: Attributes in a table with a composite primary key depend on both key fields, not just one of them.
- Proper Separation of Data: Details like menu item name or price are stored in the menu_items table instead of order_items, since they depend only on menu_item_id.
- Reduced Redundancy: Information appears only once in the correct table, avoiding repeated storage of the same details across multiple records.

4.3 Third Normal Form (3NF)

Third Normal Form removes transitive dependencies by ensuring that all non-key attributes depend only on the table's primary key and not on other non-key attributes. In the Platora system,

each table stores only the information directly related to its primary key. For example, restaurant details are stored only in the restaurants table, and not repeated in orders or menu_items, preventing indirect dependencies and keeping the database structure clean and consistent.

- No Transitive Dependencies: Non-key attributes rely only on the primary key and not on other non-key columns.
- Clear Separation of Entities: Information such as restaurant details, user information, or menu item properties is stored only in their respective tables.
- Improved Data Integrity: Updating a single value, like a restaurant address, requires changing it only once in its own table, reducing inconsistencies and ensuring better maintainability.

CHAPTER 5
DATABASE IMPLEMENTATION

5.1 Table Structure

5.1.1 Users Table

```
CREATE TABLE IF NOT EXISTS users (
  id SERIAL PRIMARY KEY,
  email VARCHAR(255) UNIQUE NOT NULL,
  password_hash TEXT NOT NULL,
  name VARCHAR(255) NOT NULL,
  phone VARCHAR(50),
  role VARCHAR(50) NOT NULL CHECK (role IN ('admin','customer','restaurant','delivery')),
  created_at TIMESTAMP DEFAULT now()
);
```

Field	Type	Null	Key	Default	Extra
id	bigint unsigned	NO	PRI	NULL	auto_increment
email	varchar(255)	NO	UNI	NULL	
password_hash	text	NO		NULL	
name	varchar(255)	NO		NULL	
phone	varchar(50)	YES		NULL	
role	varchar(50)	NO		NULL	
created_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

Figure 5.1.1.1 Users Table

5.1.2 Restaurant Table

```
CREATE TABLE IF NOT EXISTS restaurants (
  id SERIAL PRIMARY KEY,
  user_id INT REFERENCES users(id) ON DELETE CASCADE,
  name TEXT NOT NULL,
  address TEXT,
  phone VARCHAR(50),
  is_active BOOLEAN DEFAULT TRUE,
  total_orders INT DEFAULT 0,
  total_revenue NUMERIC(12,2) DEFAULT 0,
  average_rating NUMERIC(3,2) DEFAULT 0,
  total_reviews INT DEFAULT 0,
  created_at TIMESTAMP DEFAULT now()
);
```

	Field	Type	Null	Key	Default	Extra
▶	id	bigint unsigned	NO	PRI	NULL	auto_increment
	user_id	int	YES		NULL	
	name	text	NO		NULL	
	address	text	YES		NULL	
	phone	varchar(50)	YES		NULL	
	is_active	tinyint(1)	YES		1	
	total_orders	int	YES		0	
	total_revenue	decimal(12,2)	YES		0.00	
	average_rating	decimal(3,2)	YES		0.00	
	total_reviews	int	YES		0	
	created_at	timestamp	YES		CURR...	DEFAULT_GEN...

Figure 5.1.2.1 Restaurant Table

5.1.3 Menu-items Table

```
CREATE TABLE IF NOT EXISTS menu_items (
  id SERIAL PRIMARY KEY,
  restaurant_id INT REFERENCES restaurants(id) ON DELETE CASCADE,
  name TEXT NOT NULL,
  description TEXT,
  price NUMERIC(10,2) NOT NULL,
  is_available BOOLEAN DEFAULT TRUE,
  created_at TIMESTAMP DEFAULT now()
);
```

	Field	Type	Null	Key	Default	Extra
▶	id	bigint unsigned	NO	PRI	NULL	auto_increment
	restaurant_id	int	YES		NULL	
	name	text	NO		NULL	
	description	text	YES		NULL	
	price	decimal(10,2)	NO		NULL	
	is_available	tinyint(1)	YES		1	
	created_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

Figure 5.1.3.1 Menu-items Table

5.1.4 Orders Table

```
CREATE TABLE IF NOT EXISTS orders (
  id SERIAL PRIMARY KEY,
  customer_id INT REFERENCES users(id) ON DELETE SET NULL,
  restaurant_id INT REFERENCES restaurants(id) ON DELETE SET NULL,
  total_amount NUMERIC(10,2) DEFAULT 0,
  status VARCHAR(50) DEFAULT 'Pending',
```

```
  payment_mode VARCHAR(50),
  created_at TIMESTAMP DEFAULT now()
);
```

	Field	Type	Null	Key	Default	Extra
▶	id	bigint unsigned	NO	PRI	NULL	auto_increment
	customer_id	int	YES		NULL	
	restaurant_id	int	YES		NULL	
	total_amount	decimal(10,2)	YES		0.00	
	status	varchar(50)	YES		Pending	
	payment_mode	varchar(50)	YES		NULL	
	created_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

Figure 5.1.4.1 Orders Table

5.1.5 Deliveries Table

```
CREATE TABLE IF NOT EXISTS deliveries (
  id SERIAL PRIMARY KEY,
  order_id INT REFERENCES orders(id) ON DELETE CASCADE,
  delivery_agent_id INT REFERENCES users(id) ON DELETE SET NULL,
  status VARCHAR(50) DEFAULT 'Assigned',
  assigned_at TIMESTAMP DEFAULT now(),
  updated_at TIMESTAMP
);
```

	Field	Type	Null	Key	Default	Extra
▶	id	bigint unsigned	NO	PRI	NULL	auto_increment
	order_id	int	YES		NULL	
	delivery_agent_id	int	YES		NULL	
	status	varchar(50)	YES		Assigned	
	assigned_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED
	updated_at	timestamp	YES		NULL	

Figure 5.1.5.1 Deliveries Table

5.1.6 Payments Table

```
CREATE TABLE IF NOT EXISTS payments (
  id SERIAL PRIMARY KEY,
  order_id INT REFERENCES orders(id) ON DELETE CASCADE,
  amount NUMERIC(10,2) NOT NULL,
  mode VARCHAR(50),
  paid_at TIMESTAMP DEFAULT now(),
  invoice_text TEXT
```

);

Field	Type	Null	Key	Default	Extra
id	bigint unsigned	NO	PRI	NULL	auto_increment
order_id	int	YES		NULL	
amount	decimal(10,2)	NO		NULL	
mode	varchar(50)	YES		NULL	
paid_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED
invoice_text	text	YES		NULL	

Figure 5.1.6.1 Payment Table

5.1.7 Invoices Table

```
CREATE TABLE IF NOT EXISTS invoices (
  id SERIAL PRIMARY KEY,
  order_id INT REFERENCES orders(id) ON DELETE CASCADE,
  amount NUMERIC(10,2) NOT NULL,
  currency VARCHAR(10) DEFAULT 'INR',
  generated_at TIMESTAMP DEFAULT now(),
  invoice_text TEXT,
  generated_by VARCHAR(100)
);
```

Field	Type	Null	Key	Default	Extra
id	bigint unsigned	NO	PRI	NULL	auto_increment
order_id	int	YES		NULL	
amount	decimal(10,2)	NO		NULL	
currency	varchar(10)	YES		INR	
generated_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED
invoice_text	text	YES		NULL	
generated_by	varchar(100)	YES		NULL	

Figure 5.1.7.1 Invoices Table

5.1.8 Order-items Table

```
CREATE TABLE IF NOT EXISTS order_items (
  id SERIAL PRIMARY KEY,
  order_id INT REFERENCES orders(id) ON DELETE CASCADE,
  menu_item_id INT REFERENCES menu_items(id) ON DELETE SET NULL,
  quantity INT NOT NULL DEFAULT 1,
  unit_price NUMERIC(10,2) NOT NULL
);
```

	Field	Type	Null	Key	Default	Extra
▶	id	bigint unsigned	NO	PRI	NULL	auto_increment
	order_id	int	YES		NULL	
	menu_item_id	int	YES		NULL	
	quantity	int	NO		1	
	unit_price	decimal(10,2)	NO		NULL	

Figure 5.1.8.1 Order-items Table

5.1.9 Retaurant_request Table

```
CREATE TABLE IF NOT EXISTS restaurant_requests (
  id SERIAL PRIMARY KEY,
  owner_name VARCHAR(255) NOT NULL,
  owner_email VARCHAR(255) NOT NULL,
  restaurant_name TEXT NOT NULL,
  restaurant_address TEXT NOT NULL,
  restaurant_phone VARCHAR(50),
  status VARCHAR(50) DEFAULT 'Pending' CHECK (status IN ('Pending', 'Approved',
'Rejected')),
  created_at TIMESTAMP DEFAULT now()
);
```

	Field	Type	Null	Key	Default	Extra
▶	id	bigint unsigned	NO	PRI	NULL	auto_increment
	owner_name	varchar(255)	NO		NULL	
	owner_email	varchar(255)	NO		NULL	
	restaurant_name	text	NO		NULL	
	restaurant_address	text	NO		NULL	
	restaurant_phone	varchar(50)	YES		NULL	
	status	varchar(50)	YES		Pending	
	created_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

Figure 5.1.9.1 Restaurant_request Table

5.1.10 Reviews Table

```
CREATE TABLE IF NOT EXISTS reviews (
  id SERIAL PRIMARY KEY,
  restaurant_id INT REFERENCES restaurants(id) ON DELETE CASCADE,
  customer_id INT REFERENCES users(id) ON DELETE CASCADE,
  rating INT NOT NULL CHECK (rating >= 1 AND rating <= 5),
  comment TEXT,
  created_at TIMESTAMP DEFAULT now()
);
```

Field	Type	Null	Key	Default	Extra
id	bigint unsigned	NO	PRI	NULL	auto_increment
restaurant_id	int	YES		NULL	
customer_id	int	YES		NULL	
rating	int	NO		NULL	
comment	text	YES		NULL	
created_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

Figure 5.1.10.1 Reviews Table

5.1.11 Delivery_agents Table

```
CREATE TABLE IF NOT EXISTS delivery_agents (
  id SERIAL PRIMARY KEY,
  user_id INT REFERENCES users(id) ON DELETE CASCADE,
  vehicle_type VARCHAR(50),
  license_number VARCHAR(100),
  created_at TIMESTAMP DEFAULT now()
);
```

Field	Type	Null	Key	Default	Extra
id	bigint unsigned	NO	PRI	NULL	auto_increment
user_id	int	YES		NULL	
vehicle_type	varchar(50)	YES		NULL	
license_number	varchar(100)	YES		NULL	
created_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

Figure 5.1.11.1 Delivery_agents Table

5.2 Database Views, Triggers, and Stored Procedures

This section explains the database-level components implemented to improve data integrity, automate backend operations, and optimize information retrieval in the Platora system. It includes multiple analytical views that provide aggregated insights such as order history, restaurant performance, customer spending, and delivery efficiency. Triggers are used to automate routine processes like generating delivery records, updating timestamps, and recalculating restaurant ratings whenever new reviews are added. Additionally, stored procedures handle critical operations such as creating orders, assigning delivery agents, updating order statuses, and generating invoices all executed within controlled transactional blocks to ensure reliability, consistency, and accuracy across the food ordering workflow.

5.2.1 Order History View

It provides a consolidated record of all customer orders with essential details like items, amounts, and timestamps and used for quick retrieval of past orders without joining multiple tables.

```
CREATE OR REPLACE VIEW view_order_history AS
SELECT
  o.id as order_id,
  o.created_at,
  o.status,
  u.id as customer_id,
  u.name as customer_name,
  r.id as restaurant_id,
  r.name as restaurant_name,
  o.total_amount
FROM orders o
LEFT JOIN users u ON u.id = o.customer_id
LEFT JOIN restaurants r ON r.id = o.restaurant_id;
```

5.2.2 Daily Sales per Restaurant View

It shows each restaurant's total sales grouped by date and helps restaurant owners and admins monitor daily revenue trends.

```
CREATE OR REPLACE VIEW view_daily_sales_per_restaurant AS
SELECT
 r.id as restaurant_id,
r.name as restaurant_name,
date_trunc('day', o.created_at) as day,
SUM(o.total_amount) as total_sales,
COUNT(o.id)::INT as orders_count
FROM orders o
JOIN restaurants r ON r.id = o.restaurant_id
WHERE o.status <> 'Cancelled'
GROUP BY r.id, r.name, date_trunc('day', o.created_at)
ORDER BY day DESC;
```

5.2.3 Restaurant Performance View

It aggregates ratings, total reviews, total orders, and revenue for each restaurant and used for performance dashboards and analytical reports.

```
CREATE OR REPLACE VIEW view_restaurant_performance AS
SELECT
r.id as restaurant_id,
r.name as restaurant_name,
 r.total_orders,
r.total_revenue,
r.average_rating,
r.total_reviews,
COALESCE(ROUND(
CASE
WHEN r.total_orders > 0 THEN
 (r.total_revenue / r.total_orders)
ELSE 0
END, 2
), 0) as avg_order_value,
ROUND(
CASE
WHEN COUNT(o.id) > 0 THEN
 (COUNT(CASE WHEN o.status = 'Delivered' THEN 1 END) * 100.0 / COUNT(o.id))
ELSE 0
END, 2
) as delivery_success_rate
FROM restaurants r
LEFT JOIN orders o ON r.id = o.restaurant_id
GROUP BY r.id, r.name, r.total_orders, r.total_revenue, r.average_rating, r.total_reviews;
```

5.2.4 Customer Spending View

It displays the total money spent by each customer across all orders and is useful for loyalty insights and customer behavior analysis.

```
CREATE OR REPLACE VIEW view_customer_spending AS
SELECT
 u.id as customer_id,
 u.name as customer_name,
 COUNT(o.id)::INT as total_orders,
 COALESCE(SUM(o.total_amount), 0) as total_spent,
```

```
  COALESCE(AVG(o.total_amount), 0) as avg_order_value,
  MAX(o.created_at) as last_order_date
FROM users u
LEFT JOIN orders o ON u.id = o.customer_id
WHERE u.role = 'customer'
GROUP BY u.id, u.name;
```

5.2.5 Delivery Performance View

It summarizes each delivery agent's completed deliveries, status counts, and average delivery time.

```
CREATE OR REPLACE VIEW view_delivery_performance AS
SELECT
  u.id as agent_id,
  u.name as agent_name,
  COUNT(d.id)::INT as total_deliveries,
  COUNT(CASE WHEN d.status = 'Delivered' THEN 1 END)::INT as completed_deliveries,
  COALESCE(ROUND(
    CASE
      WHEN COUNT(d.id) > 0 THEN
        (COUNT(CASE WHEN d.status = 'Delivered' THEN 1 END) * 100.0 / COUNT(d.id))
      ELSE 0
    END, 2
  ), 0) as completion_rate
FROM users u
LEFT JOIN deliveries d ON u.id = d.delivery_agent_id
WHERE u.role = 'delivery'
GROUP BY u.id, u.name;
```

5.2.6 Create Delivery for Orders Stored Procedure

It automatically assigns an available delivery agent when a new order is placed and also ensures every order gets a delivery entry instantly without manual intervention.

```
CREATE OR REPLACE FUNCTION fn_create_delivery_for_order(p_order_id INT)
RETURNS VOID LANGUAGE plpgsql AS $$
DECLARE
selected_agent INT;
BEGIN
SELECT u.id INTO selected_agent
```

```
 FROM users u
 LEFT JOIN deliveries d ON d.delivery_agent_id = u.id AND d.status <> 'Delivered'
WHERE u.role = 'delivery'
GROUP BY u.id
 ORDER BY COUNT(d.id) ASC, u.id ASC
 LIMIT 1;
 -- Insert delivery; selected_agent may be NULL if no delivery agents exist.
 INSERT INTO deliveries(order_id, delivery_agent_id, status, assigned_at)
 VALUES (p_order_id, selected_agent, 'Assigned', now());
END;
$$;
```

5.2.7 Generate Invoice Stored Procedure

It creates a structured invoice containing order amount, timestamp, and invoice text and also prevents duplicate invoice generation and maintains consistent billing records.

```
CREATE OR REPLACE FUNCTION fn_generate_invoice(p_order_id INT, p_mode
VARCHAR)
RETURNS VOID LANGUAGE plpgsql AS $$
DECLARE
total NUMERIC(10,2);
 invoice TEXT;
inv_id INT;
BEGIN
SELECT COALESCE(SUM(oi.quantity * oi.unit_price),0) INTO total
FROM order_items oi
 WHERE oi.order_id = p_order_id;
 invoice := format('Invoice for order %s:\nTotal: %s\nGenerated at: %s', p_order_id, total,
now());
 -- Insert into payments for payment history
 INSERT INTO payments(order_id, amount, mode, paid_at, invoice_text)
VALUES (p_order_id, total, p_mode, now(), invoice);

 -- insert a simplified invoice record as well
INSERT INTO invoices(order_id, amount, currency, generated_at, invoice_text, generated_by)
 VALUES (p_order_id, total, 'INR', now(), invoice, p_mode)
 RETURNING id INTO inv_id;
```

```
-- update orders total_amount
UPDATE orders SET total_amount = total WHERE id = p_order_id;
END;
$$;
```

5.2.8 Get Deliveries for Agent Stored Procedure

It returns all deliveries assigned to a specific delivery agent and is used to populate the agent's dashboard for tracking active and completed tasks

```
CREATE OR REPLACE FUNCTION fn_get_deliveries_for_agent(p_agent_id INT)
RETURNS TABLE (
 delivery_id INT,
 order_id INT,
status TEXT,
 assigned_at TIMESTAMP,
 updated_at TIMESTAMP,
 total_amount NUMERIC,
 order_status TEXT,
 customer_name TEXT,
 customer_contact TEXT,
 delivery_address TEXT,
items JSON
) LANGUAGE plpgsql AS $$
BEGIN
 RETURN QUERY
SELECT
d.id as delivery_id,
 d.order_id,
 d.status::text,
 d.assigned_at,
 d.updated_at,
 o.total_amount,
 o.status::text as order_status,
 u.name::text as customer_name,
 u.phone::text as customer_contact,
 r.address::text as delivery_address,
```

```
 COALESCE(
 json_agg(
 json_build_object(
'order_item_id', oi.id,
'menu_item_id', oi.menu_item_id,
'name', mi.name,
'quantity', oi.quantity,
'unit_price', oi.unit_price
) ORDER BY oi.id
) FILTER (WHERE oi.id IS NOT NULL),
'[]'::json
) as items
  FROM deliveries d
 JOIN orders o ON o.id = d.order_id
  LEFT JOIN users u ON u.id = o.customer_id
  LEFT JOIN restaurants r ON r.id = o.restaurant_id
  LEFT JOIN order_items oi ON oi.order_id = o.id
 LEFT JOIN menu_items mi ON mi.id = oi.menu_item_id
 WHERE d.delivery_agent_id = p_agent_id
 GROUP BY d.id, d.order_id, d.status, d.assigned_at, d.updated_at,
 o.total_amount, o.status, u.name, u.phone, r.address
 ORDER BY d.assigned_at DESC;
 END;
 $$;
```

5.2.9 Update Order Status Stored Procedure

It updates the order status and performs linked actions such as timestamp updates or delivery updates and ensures consistent order workflow and smooth status transitions across the system.

```
CREATE OR REPLACE FUNCTION fn_update_order_status(p_order_id INT, p_status
VARCHAR)
RETURNS VOID LANGUAGE plpgsql AS $$
BEGIN
  -- Update order status
  UPDATE orders SET status = p_status WHERE id = p_order_id;
```

```
  -- If order is cancelled, refund payment if exists
  IF p_status = 'Cancelled' THEN
    -- Update payment status to refunded
    UPDATE payments SET mode = 'Refunded' WHERE order_id = p_order_id;

    -- Insert refund record
    INSERT INTO payments(order_id, amount, mode, paid_at, invoice_text)
    SELECT order_id, amount, 'Refund', now(), 'Refund for cancelled order ' || p_order_id
    FROM payments
    WHERE order_id = p_order_id AND mode != 'Refund' AND mode != 'Refunded'
    LIMIT 1;
  END IF;

  -- If order is delivered, update delivery status
  IF p_status = 'Delivered' THEN
    UPDATE deliveries SET status = 'Delivered', updated_at = now() WHERE order_id =
p_order_id;
  END IF;
END;
$$;
```

5.2.10 Calculate Restaurant Rating Stored Procedure

Recalculates a restaurant's average rating whenever a new review is added.
Maintains real-time accuracy of displayed ratings across all user interfaces.

```
CREATE OR REPLACE FUNCTION fn_calculate_restaurant_rating(p_restaurant_id INT)
RETURNS TABLE (
 restaurant_id INT,
avg_rating NUMERIC,
total_reviews INT
) LANGUAGE plpgsql AS $$
DECLARE
v_avg_rating NUMERIC := 0.0;
v_total_reviews INT := 0;
BEGIN
-- Calculate average rating and total reviews
```

```
SELECT
COALESCE(ROUND(AVG(r.rating), 2), 0.0),
COUNT(r.id)::INT
INTO
v_avg_rating, v_total_reviews
FROM reviews r
WHERE r.restaurant_id = p_restaurant_id;

-- Return the results
RETURN QUERY
SELECT
p_restaurant_id as restaurant_id,
v_avg_rating as avg_rating,
v_total_reviews as total_reviews;
END;
$$;
```

5.2.11 Update Restaurant Status Trigger

It automatically updates restaurant statistics whenever an order's status changes and ensures totals like revenue, delivered orders, and ratings remain accurate and updated.

```
CREATE OR REPLACE TRIGGER trigger_update_order_status
  AFTER UPDATE OF status ON orders
  FOR EACH ROW
  EXECUTE FUNCTION fn_update_restaurant_stats();

-- Trigger function to automatically create delivery when order is confirmed
CREATE OR REPLACE FUNCTION fn_auto_create_delivery()
RETURNS TRIGGER LANGUAGE plpgsql AS $$
BEGIN
  -- Only create delivery for new orders with 'Pending' status
  IF TG_OP = 'INSERT' AND NEW.status = 'Pending' THEN
    -- Call the existing function to create delivery
    PERFORM fn_create_delivery_for_order(NEW.id);
  END IF;
  RETURN NEW;
END;
$$;
```

5.2.12 Auto create Delivery Trigger

It creates a delivery record immediately when a new order is placed and guarantees every order is assigned for delivery without requiring manual steps.

```
CREATE OR REPLACE TRIGGER trigger_auto_create_delivery
  AFTER INSERT ON orders
  FOR EACH ROW
  EXECUTE FUNCTION fn_auto_create_delivery();

-- Trigger function to update timestamps
CREATE OR REPLACE FUNCTION fn_update_timestamp()
RETURNS TRIGGER LANGUAGE plpgsql AS $$
BEGIN
  NEW.updated_at = now();
  RETURN NEW;
END;
$$;
```

5.2.13 Update delivery timestamp Trigger

It updates the updated_at timestamp whenever a delivery record is modified and maintains accurate tracking of delivery progress and latest status changes.

```
CREATE OR REPLACE TRIGGER trigger_update_delivery_timestamp
  BEFORE UPDATE ON deliveries
  FOR EACH ROW
  EXECUTE FUNCTION fn_update_timestamp();
```

CHAPTER 6
SCREENSHOTS

6.1 Landing Page

This screen displays the Platora landing page, showcasing the platform's purpose for ordering food online from nearby restaurants. It highlights key features such as fast delivery, curated restaurants, and provides quick access to Login and Sign-up options.

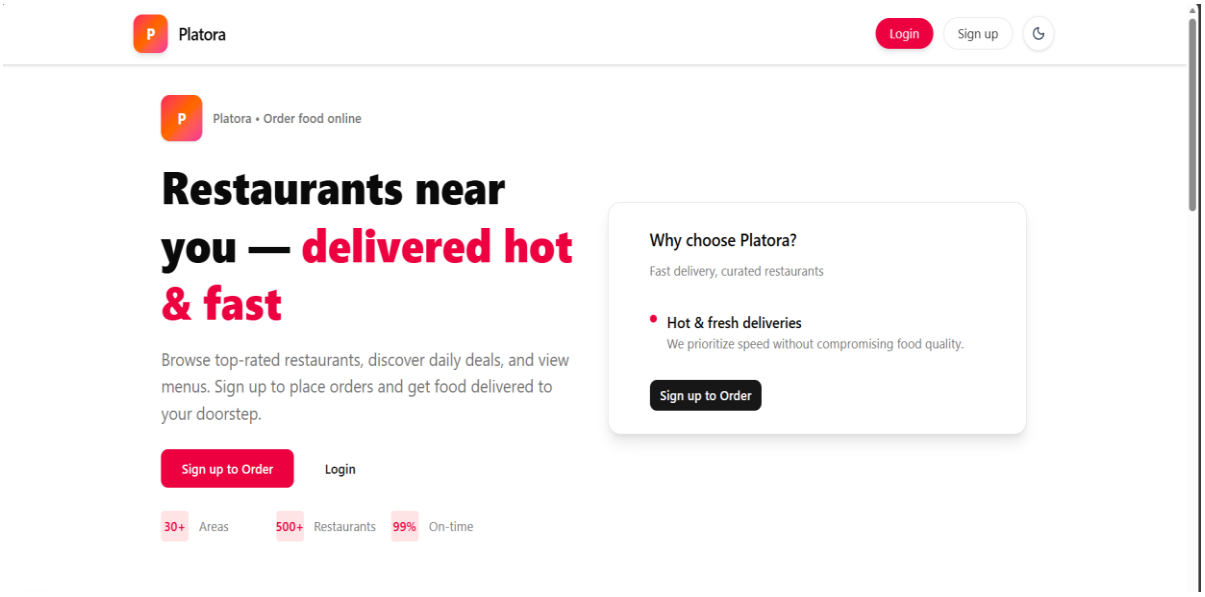

Figure 6.1.1 Landing Page

6.2 Customer Dashboard Page

The customer dashboard offers a personalized view where users can manage orders, explore restaurants, and update their profile. It also provides quick statistics like catalog progress and top restaurant picks.

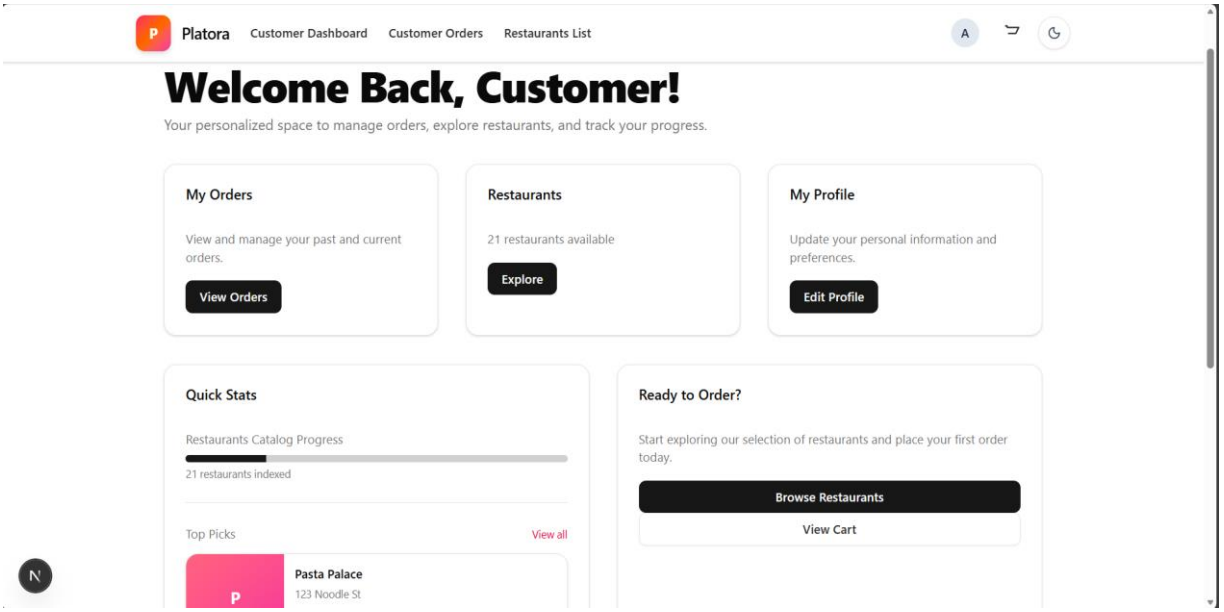

Figure 6.2.1 Customer Dashboard Page

6.3 Restaurant Dashboard Page

This screen shows the restaurant owner's dashboard, where owners can view and edit their restaurant profile. Key details like restaurant name, contact information, address, and status are displayed clearly for management. This page also allows restaurant owners to manage orders, update menus, and view customer reviews. It centralizes essential restaurant operations, helping owners monitor performance and customer feedback.

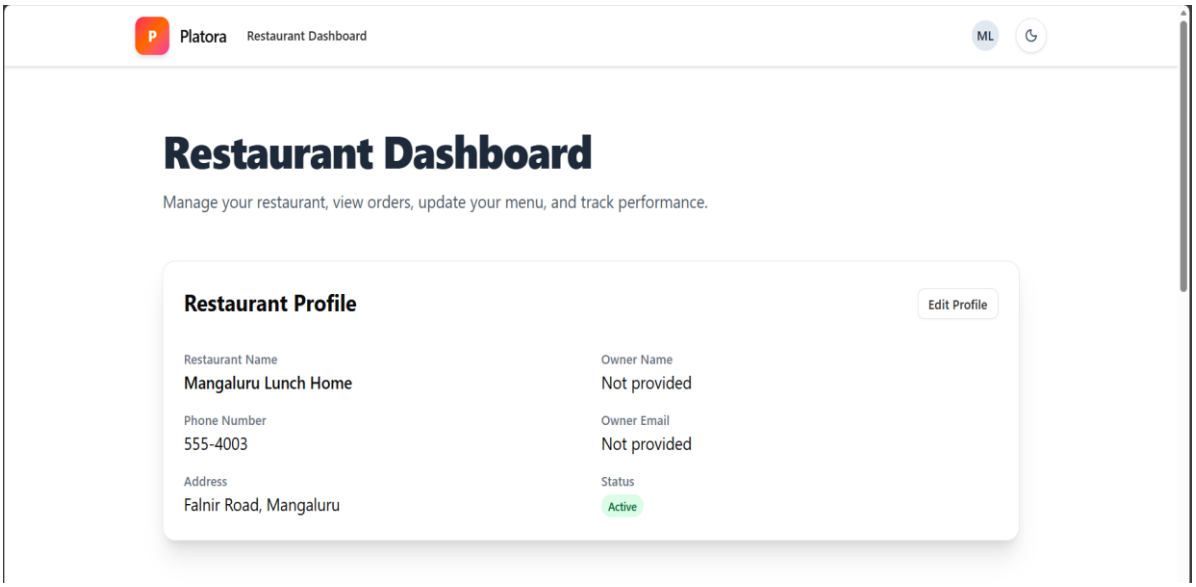

Figure 6.3.1 Restaurant Dashboard Page

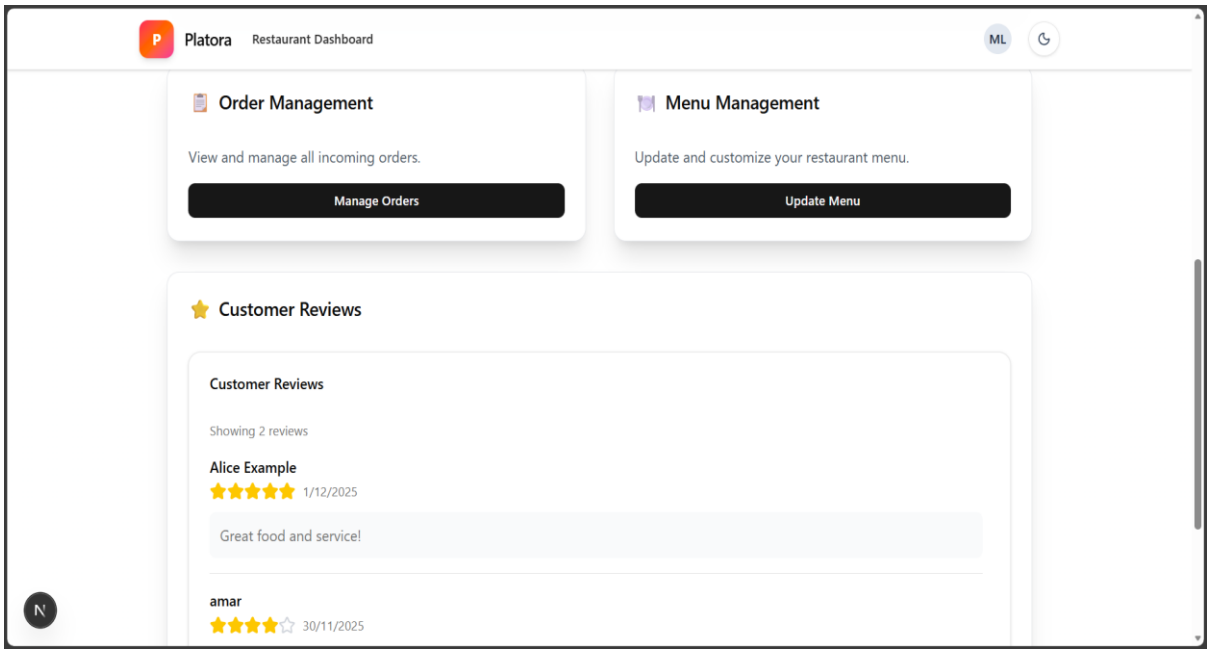

Figure 6.3.2 Restaurant Dashboard Page

6.4 Admin Dashboard Page

The admin dashboard enables platform administrators to manage users, handle restaurant registration requests, and configure platform settings. It also includes access to analytics and platform-wide statistics.

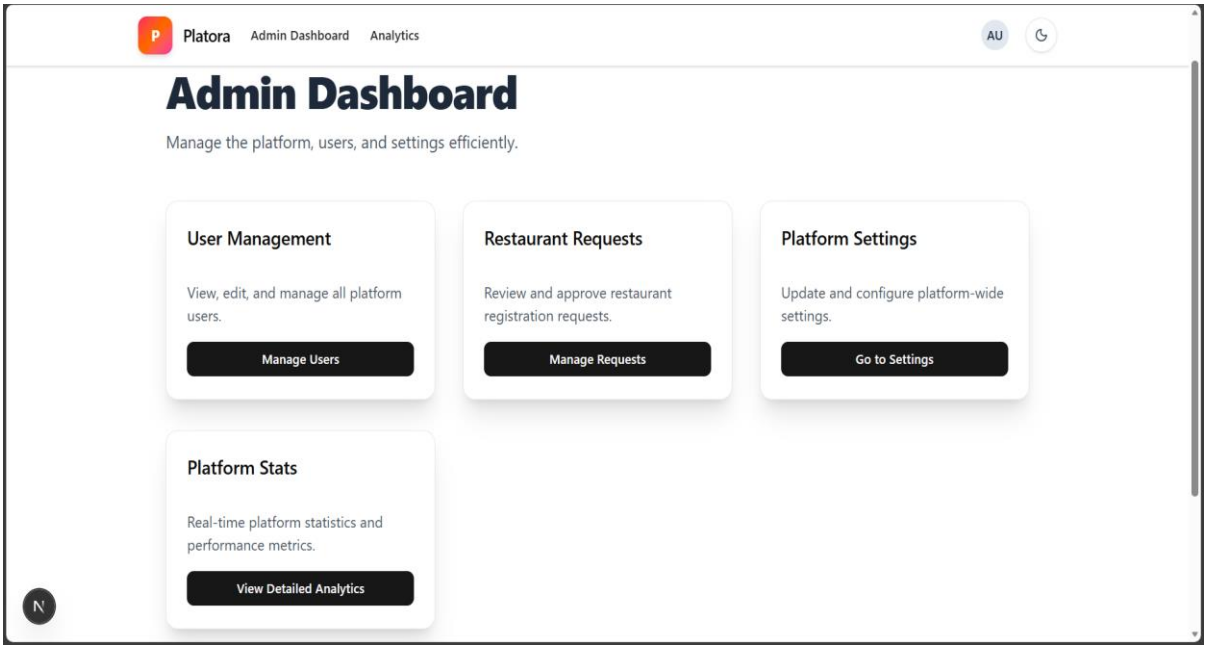

Figure 6.4.1 Admin Dashboard Page

6.5 Delivery Agent Dashboard Page

This screen is used by delivery agents to track their assigned deliveries. It displays daily, pending, and completed deliveries, along with a list of current delivery tasks.

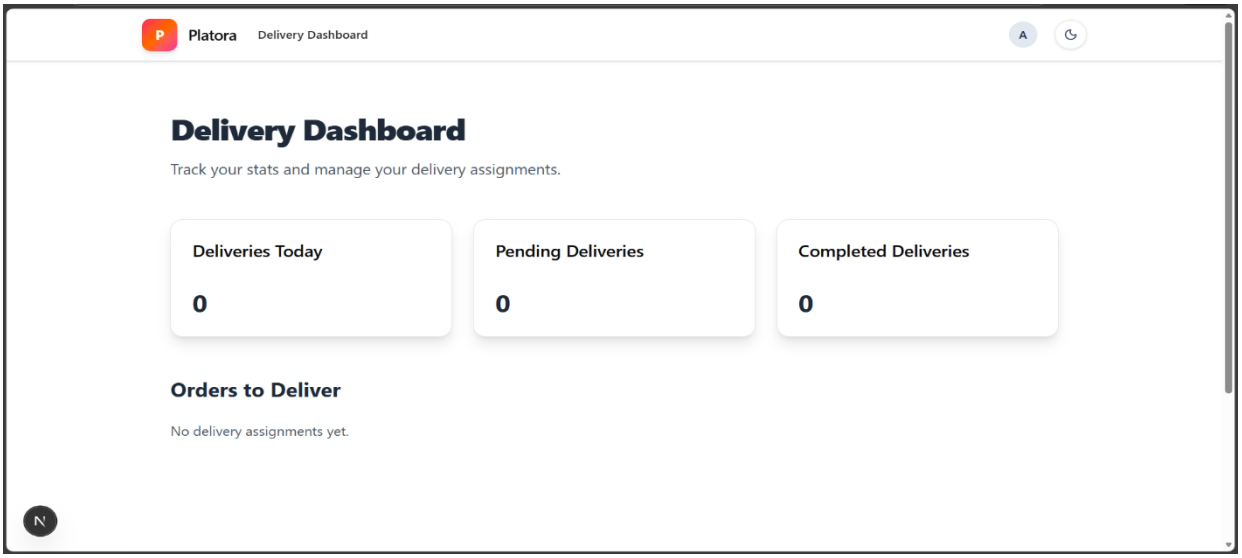

Figure 6.5.1 Delivery agent Dashboard Page

CHAPTER 7
CONCLUSION

7.1 Conclusion

The project successfully developed a robust and highly consistent DBMS backend for the Platora Online Food Delivery System. The database design emphasizes reliability, accuracy, and efficiency, enabling smooth handling of core operations such as order placement, menu management, restaurant onboarding, delivery assignments, payments, and customer reviews. By structuring all tables according to Third Normal Form, redundancy was eliminated and data relationships were clearly modeled, ensuring long-term maintainability and scalability of the platform.

A key strength of the system lies in the use of PostgreSQL stored procedures, which encapsulate complex multi-step workflows such as order creation, automatic delivery assignment, invoice generation, and status transitions. These procedures leverage transactional control and row-level locking to prevent conflicts, ensuring that operations remain safe and consistent even when multiple users place orders simultaneously. This significantly enhances the reliability of the backend and supports high-traffic, real-world usage scenarios common in food delivery applications.

Additionally, automated triggers were implemented to maintain dynamic information such as restaurant ratings, delivery timestamps, and order-related updates. Whenever a customer submits a review or a delivery agent updates their status, the triggers automatically adjust relevant fields to ensure accurate, real-time data across all modules. This automation removes the need for manual recalculations and greatly strengthens data consistency.

To improve query performance and support fast access to aggregated insights, optimized database views were created. Views such as order history, restaurant performance, customer spending patterns, and delivery efficiency simplify complex joins and allow the application to fetch frequently used information quickly. Together, these components form a database layer that is reliable, transactional, and optimized—providing a strong foundation for a scalable and production-ready food delivery system.

7.2 Future Enhancements

1. Add real-time GPS-based delivery tracking for accurate live order updates.
2. Implement an AI recommendation engine to suggest dishes and restaurants.
3. Introduce a dynamic pricing and automated offer-generation system.
4. Enable multi-restaurant ordering and scheduled future deliveries.
5. Develop an advanced admin dashboard with predictive analytics and insights.

References

1. PostgreSQL Global Development Group. (n.d.). *PostgreSQL documentation*. https://www.postgresql.org/docs/
2. Connolly, T., & Begg, C. (2015). *Database systems: A practical approach to design, implementation, and management* (6th ed.). Pearson. — (Used as a standard reference for relational database design principles.)
3. Chen, P. P. (1976). The entity-relationship model—Toward a unified view of data. *ACM Transactions on Database Systems, 1*(1), 9–36. https://doi.org/10.1145/320434.320440