DESIGN AND IMPLEMENTATION OF A MODULAR MULTI-TENANT FOOD DELIVERY SYSTEM

BY

OJIMMA CHIMA PETER

PSC2105375

DEPARTMENT OF COMPUTER SCIENCE

FACULTY OF COMPUTING

UNIVERSITY OF BENIN

BENIN CITY

NOVEMBER 2025

DESIGN AND IMPLEMENTATION OF A MODULAR MULTI-TENANT FOOD DELIVERY SYSTEM

BY

OJIMMA CHIMA PETER
PSC2105375

A FINAL YEAR PROJECT REPORT SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF BENIN, EDO STATE, IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF BACHELOR'S DEGREE IN COMPUTER SCIENCE.

NOVEMBER 2025

CERTIFICATION

This is to certify that OJIMMA CHIMA PETER, with Matriculation number PSC2105375, carried out this project work under my supervision. It is adequate and satisfactory, both in scope and content, for the award of Bachelor of Science (B.Sc.) Degree in Computer Science from the University of Benin.

APPROVAL

This project work, titled "**DESIGN AND IMPLEMENTATION OF A MODULAR MULTI-TENANT FOOD DELIVERY SYSTEM** ", carried out by OJIMMA CHIMA PETER with Matriculation Number PSC2105375, is hereby approved in partial fulfillment of the requirements for the award of Bachelor of Science (B.Sc) Degree in Computer Science from the University of Benin.

DEDICATION

This project is dedicated to Almighty God for His infinite wisdom, guidance, and strength throughout this journey. I also dedicate it to my loving family, the Ojimmas, whose selfless love and unwavering support have shaped me into the person I am today. To my friends and coursemates, I express my sincere gratitude for your encouragement, understanding, and constant support. Your positive energy has made this journey both fulfilling and worthwhile. This work stands as a reflection of the collective strength and inspiration I have drawn from each of you.

ACKNOWLEDGEMENT

My utmost acknowledgement goes to God Almighty for giving me the strength, wisdom and knowledge throughout my academic journey. I would like to express my gratitude to my family for their consistent guidance, support and encouragement. I would also like to give special thanks to my project supervisor, Mr Imiefo, for his guidance and support towards the successful completion of this project. I would also like to specially thank my good friends Eveshoginameh Samuel Daisi, Osaguona Eseosa and Joanna Osaghae for their steady support and encouragement guided towards the completion of this project.

TABLE OF CONTENT

CERTIFICATION ....
APPROVAL ....
DEDICATION ....
ACKNOWLEDGEMENT ....
TABLE OF CONTENT ....
ABSTRACT ....
CHAPTER ONE ....
INTRODUCTION ....
1.1 Background of Study ....
1.2 Problem Definition ....
1.3 Research Aims and Objectives ....
1.4 Scope of Research ....
1.5 Research Methodology ....
1.6 Research Significance ....
1.7 Limitations ....
CHAPTER TWO ....
LITERATURE REVIEW ....
2.1 Introduction ....
2.2 Conceptual Overview ....
2.2.1 Overview of Food Delivery Systems ....
2.2.2 Evolution of Multi-Tenant SaaS Platforms ....
2.2.3 Modular Monolithic vs Microservices Architecture ....
2.3 Theoretical Framework ....
2.3.1 Multi-Tenancy as an Architectural Principle ....
2.3.2 Data Isolation Models in Multi-Tenancy ....
2.3.3 Resource Sharing and Scalability ....
2.3.4 Tenant Context and Request Handling ....
Summary of Theoretical Framework ....
2.4 Empirical Review ....
2.4.1 Studies on Multi-Tenancy in SaaS Applications ....
2.4.2 Data Security and Privacy in Multi-Tenant Systems ....
2.4.3 Applications in Food Delivery Platforms ....
2.4.4 Performance and Scalability ....
Summary ....
2.5 Comparative Analysis of Related Studies ....

2.5.1 Strengths and Weaknesses of Existing Systems ....
2.5.2 Comparative Table of Related Studies ....
2.5.3 Gap Analysis ....
2.6 Summary ....
CHAPTER THREE ....
SYSTEM ANALYSIS AND DESIGN ....
3.1 Introduction ....
3.2 System Design Approach ....
3.2.1 Overview of the Proposed Approach ....
3.2.1 Overview of the Proposed Approach ....
3.3 Design Aims and Objectives ....
3.4 System Architecture ....
3.4.1 Architectural Layers ....
3.4.2 Core Database Design ....
3.4.3 Relationships and Normalization ....
3.4.4 Entity-Relationship Diagram ....
3.5 Module Design ....
3.5.1 Authentication and User Management Module ....
3.5.2 Restaurant and Menu Management Module ....
3.5.3 Order Management Module ....
3.5.4 Payment Processing Module ....
3.5.5 Transactions Module ....
3.5.6 Notification Module ....
3.5.7 Analytics and Reporting Module ....
3.5.8 Data Flow Diagram (DFD) ....
3.6 Software Development Life Cycle (SDLC) ....
3.6.1 Requirement Analysis Phase ....
3.6.2 Design Phase ....
3.6.3 Implementation Phase ....
3.6.4 Testing Phase ....
3.6.5 Deployment Phase ....
3.7 Technology Stack ....
3.8 Deployment and Infrastructure Design ....
3.8.1 Scalability and Performance ....
3.8.2 Security Considerations ....
3.9 System Workflow ....
3.9 Summary ....
CHAPTER FOUR ....

SYSTEM IMPLEMENTATION ....
4.1 Introduction ....
4.2 Development Environment and Tools ....
4.3 Implementation Overview ....
4.4 Core Module Implementations ....
4.4.1 Authentication and Tenant Login ....
4.4.2 Restaurant and Menu Management ....
4.4.3 Order Management ....
4.4.4 Payment Integration ....
4.4.5 Notifications and Queue Workers ....
4.4.6 Analytics and Reporting ....
4.5 Database Configuration ....
4.6 Deployment ....
4.7 Summary ....
CHAPTER FIVE ....
SUMMARY, CONCLUSION AND RECOMMENDATION ....
5.1 Summary ....
5.2 Conclusion ....
5.3 Recommendations ....
5.4 Summary of the Chapter ....
LIST OF ACRONYMS ....
REFERENCES ....

ABSTRACT

The rapid growth of online food delivery platforms has introduced challenges in supporting multiple restaurants on a single system. This research proposes a **multi-tenant backend system for food delivery platforms**, enabling each restaurant to operate independently while sharing the same backend infrastructure. The system ensures that tenants maintain full control over orders, menus, and operational data, eliminating the risk of conflicts or data overlap.

Unlike traditional single-tenant or loosely structured multi-vendor systems, this implementation ensures strong tenant isolation by design which means each restaurant experiences the system as if it were built specifically for them. All operations including order processing, background jobs, inventory updates, and notifications are executed in a tenant-aware context.

This work contributes a practical approach for anyone building a multi-vendor platform and not just in food delivery but also in areas where clean isolation, predictable behavior, and efficient resource sharing are required.

CHAPTER ONE

INTRODUCTION

1.1 Background of Study

The global food delivery industry has experienced massive growth over the past decade, largely driven by increased internet penetration, smartphone adoption, and the demand for convenience in urban lifestyles. In this space, multi-tenant platforms such as Uber Eats, DoorDash, and Jumia Food have emerged, allowing various restaurants to operate under a single umbrella while reaching a wider customer base (Statista, 2023). These platforms rely on backend systems that can handle multiple vendors (or tenants), customer orders, and logistics workflows concurrently.

However, most legacy systems used in food delivery or even e-commerce were not designed with multi-tenancy in mind. Instead, they often follow a tightly coupled monolithic structure where vendors share the same data models, and logic is abstracted at the application level. This introduces issues with **data leakage**, **cross-tenant interference**, and **order collision**, all of which can affect the integrity of the business (Amazon Web Services, 2023). The lack of isolated order management also limits customizations for each restaurant, making it difficult to scale or localize features.

Multi-tenancy, as a software architectural principle, enables a single system to serve multiple tenants while logically isolating their data and workflows. It allows a platform to maintain one codebase and infrastructure, while each tenant feels like they have their own dedicated environment (Bezemer et al., 2019). In the context of food delivery, this means orders, menus, customer data, and delivery statuses are scoped to individual restaurants without overlap or dependency on other vendors' data.

In addition, **tenant-aware order management** is key to reliability and accountability. When a customer places an order, the system must resolve the correct tenant context and handle the order from start to finish including inventory updates, notifications, and payment processing all within that tenant's domain. This is not just a technical need, but also a **business-critical feature** for ensuring trust between the platform and vendors (Gartner, 2022).

Recent work in distributed systems has made tenant isolation easier to achieve, even in shared platforms. Features like dynamic tenant resolution, scoped database connections, and isolated deployment enable horizontal scaling while preventing one tenant's issues from impacting others. This project builds on those ideas and applies them to a **modular monolithic food delivery system**, focusing on how to keep each vendor's data and operations fully isolated while making the system reliable, easy to maintain, and ready to grow.

1.2 Problem Definition

Modern food delivery platforms face significant architectural and operational challenges when scaling to serve multiple vendors within a single application environment. Many systems rely on tightly coupled, monolithic designs where all vendors share the same database tables, logic, and services. This shared structure often leads to problems such as data leakage, interference between vendor operations, and limited flexibility in vendor-specific customizations (Bezemer et al., 2019; AWS, 2023).

In a typical multi-vendor setup, orders from different restaurants are processed through the same pipelines without any strict logical separation. As a result, issues like incorrect order routing, unauthorized data access, and performance bottlenecks become increasingly common, especially as the number of vendors and users grows (Gartner, 2022). Moreover, onboarding new restaurants or deploying tenant-specific updates becomes complex and error-prone in the absence of clear isolation mechanisms.

There is a need for a system that treats each restaurant (tenant) as an independent entity within a shared platform, ensuring full data isolation, autonomous order management, and scalable service delivery while still making it affordable and suitable for startup companies and small to medium business enterprises. The problem, therefore, is how to design and implement a food delivery system that supports multiple tenants while enforcing strict boundaries between them at both the application and infrastructure levels.

This research proposes a multi-tenant food delivery system with isolated tenant management, using a modular monolith and tenant-aware routing to allow each vendor to operate independently while sharing the same infrastructure.

1.3 Research Aims and Objectives

The main aim of this research is to build a **multi-tenant food delivery system** where multiple restaurants can operate independently on a shared system, without affecting each other's data, order flow, or customer interactions. The system will enforce **strict tenant isolation** within a single codebase by using a **modular architecture**, allowing features to be organized into separate, self-contained units.

Instead of splitting the system into independent microservices, this project will use a **modular monolithic approach**, where each feature (auth, orders, payments, etc.) is built as a standalone module. This keeps the system simple to deploy and maintain while still allowing for separation of logic, data boundaries, and easier scaling when needed.

To achieve this, the study is guided by the following objectives:

1. **Analyze the structure of existing food delivery platforms**, especially how they handle multiple vendors, and identify common issues with data overlap and scalability.

2. **Design the system using modular architecture**, where each domain such as authentication, order processing, restaurant management, and payment handling is built as a self-contained module.

3. **Implement tenant isolation within the monolith**, so that all operations (orders, menus, customer data) are scoped to a specific tenant using techniques like tenant ID binding.

4. **Develop dynamic tenant resolution**, where the system detects which restaurant is making a request (using subdomains or tokens) and applies the correct tenant context throughout the app lifecycle.

5. **Apply security practices** such as token-based authentication and validation middleware to prevent unauthorized access and enforce tenant boundaries.

6. **Evaluate the system** in terms of tenant isolation, onboarding ease, and overall performance, especially under conditions where multiple restaurants are actively processing orders.

This modular monolith design allows for future migration to microservices if needed, but keeps the current system simple, faster to build, and easier to manage in a single codebase.

1.4 Scope of Research

This research focuses on building a **multi-tenant food delivery system** that allows different restaurants to operate independently on a shared platform. The main focus is on **tenant isolation** to ensure that orders, menus, and customer data from one restaurant cannot interfere with or affect another.

The project will be built using a **modular monolith architecture**, where features like authentication, order management, restaurant setup, and payments are separated into modules within a single application. This structure keeps the codebase organized and makes it easier to maintain boundaries between features and tenants.

This research does **not** focus on advanced logistics like rider tracking, third-party integrations, or production-ready performance scaling. The goal is to **prove that tenant isolation and modular separation can be enforced within a single deployable system**, while still allowing the app to grow and scale horizontally when needed.

This project is primarily focused on the backend implementation of a food delivery system using a modular monolithic architecture. The work will involve the design, development, and testing of backend modules. The project scope however, does not extend to the development of a user interface, as the emphasis is on ensuring a robust and scalable backend system.

1.5 Research Methodology

This research will follow a practical and implementation-driven approach, with the main goal being to **design, build, and test** a working application where multiple vendors (tenants) will be able to manage their orders and menus independently within the same codebase and deployment environment.

Literature Review:

We will begin by reviewing existing multi-tenant architectures, common problems with shared

systems, and techniques used in SaaS platforms to isolate tenants. This will guide the decision to adopt a modular monolith structure and employ isolated deployments as the best fit for the use case.

System Design:
We will design the system using clearly defined modules: auth, orders, restaurants, and payments. Each module will be scoped in such a way that tenant-specific operations are contained and cannot affect other tenants. Proper boundaries will be set at both the code and database level.

Implementation:
We will implement the system using a backend framework **NestJS**, along with **PostgreSQL** for structured data storage. Every request will carry a **tenant ID**, which will be resolved at runtime based on the request source (e.g., subdomain, headers).

Testing:
We will create multiple test tenants and simulate user activities such as placing orders, updating menus, and logging in. The system will be tested to ensure that no tenant is able to access or interfere with another tenant's data. We will also test how the system behaves under concurrent requests and high load.

1.6 Research Significance

This research is important because it tackles a real-world problem: how to isolate tenant data and operations within a shared system reliably and efficiently. As food delivery platforms grow and onboard more vendors, especially in developing tech ecosystems like Nigeria's, trust and data security become critical.

By designing a modular food delivery system with tenant-aware logic and scoped database access, this study will provide a blueprint that balances common infrastructure with strong separation between vendors. Modular design within a single deployable application offers a practical middle ground making it easier to manage than microservices, but safer and more scalable than traditional monolithic apps (Pushpan, 2024; Abdul et al., 2018).

This work is also relevant to SaaS developers and platform engineers because it demonstrates real-world application of dynamic tenant resolution and scoped access controls which are techniques recommended by AWS for effective tenant isolation (Amazon Web Services, 2023). Applying these proven strategies in a food delivery context fills a practical gap in existing research and offers lessons for low-resource startups and developers working in constrained environments.

Finally, the findings will serve as a learning resource for students, early-stage developers, and smaller teams aiming to build scalable systems. Instead of complex microservices, the modular monolith approach deployed on Render demonstrates how strong isolation and predictable behavior can be achieved with a lean architecture.

1.7 Limitations

Despite the practical value of this research, a few limitations are expected:

1. **Limited Scalability Testing:**
   While the system will be built to support horizontal scaling, the research will not involve large-scale deployment or stress testing under real-world traffic. As a result, conclusions about performance under production-scale loads may be theoretical or based on simulations rather than actual high-traffic environments.

2. **Vendor Scope:**
   The research will focus primarily on isolating restaurants (vendors) in a multi-tenant setup. It will not explore advanced concepts like customer segmentation, delivery partner isolation, or enterprise-level tenant configurations.

3. **Security Focus Will Be Minimal:**
   While isolation improves security indirectly, this project will not dive deeply into broader cybersecurity topics such as token hijacking, rate limiting, intrusion detection, or end-to-end encryption beyond basic authentication and authorization.

4. **Tech Stack Bias:**
   The system will be implemented using a specific stack (Node.js, NestJS, PostgreSQL,

NextJS). While the architectural principles will be transferable, some techniques and optimizations may not apply directly to other ecosystems like Django, Laravel, or Spring Boot.

CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

This chapter reviews existing research and technological developments related to multi-tenant architectures, software-as-a-service (SaaS) platforms, and food delivery systems. The goal is to establish the theoretical foundation for building a modular monolithic multi-tenant backend system that supports isolated restaurant operations on a shared infrastructure. The review begins with a conceptual discussion of food delivery systems and multi-tenant SaaS platforms, followed by the theoretical framework that defines core architectural principles. It continues with an empirical review of prior studies and commercial platforms, and concludes with a comparative analysis highlighting existing gaps that this study aims to address.

2.2 Conceptual Overview

The conceptual overview explains key ideas and systems that influence the development of multi-tenant food delivery platforms. It examines food delivery systems, the evolution of multi-tenancy in SaaS environments, and the distinction between modular monolithic and microservices architectures.

2.2.1 Overview of Food Delivery Systems

Food delivery systems are digital platforms that connect restaurants with customers to process and manage food orders. They combine customer interfaces, restaurant dashboards, and backend infrastructures that handle order routing, menu management, and payment processing (Ray, Dhir, Bala, & Kaur, 2019). In early models, orders were received via phone or email and fulfilled manually. The rise of internet technologies and mobile applications transformed this process into a fully automated digital experience (Kumar & Anbanandam, 2020).

Modern food delivery systems such as Uber Eats, DoorDash, Deliveroo, and Jumia Food rely on centralized platforms that host multiple restaurants while providing real-time updates, payment

integration, and data analytics. Despite these improvements, these platforms face persistent challenges in scalability, vendor onboarding, and cross-tenant interference (Liu, Ma, Wang, & Zhao, 2020). When multiple vendors share the same application instance, issues such as data leakage or order overlap can occur if the system is not properly designed for isolation.

2.2.2 Evolution of Multi-Tenant SaaS Platforms

Multi-tenancy originated as a core architectural concept in SaaS environments, where a single software instance serves multiple clients, or tenants, with logical separation of data and configuration (Bezemer, Zaidman, Platzbeecker, & Lago, 2019). This approach allows vendors to share computing resources while maintaining individual access control and data privacy. The primary motivations for multi-tenancy include cost efficiency, simplified maintenance, and scalable resource allocation (Chong & Carraro, 2006).

However, multi-tenancy introduces design challenges, including the need to ensure strict data isolation, prevent performance degradation under heavy tenant loads (commonly referred to as "noisy neighbour" effects), and provide meaningful tenant-level customization. Walraven, Truyen, and Joosen (2011) note that standard middleware for multi-tenant applications often lacks support for flexible tenant-specific configurations and data separation. To address these challenges, practitioners have implemented data-isolation strategies such as database-per-tenant, schema-per-tenant, and shared schema with tenant identifiers (Azeez et al., 2010). More recent studies have focused on lightweight multi-tenant architectures suitable for startups and small to medium enterprises, with Pushpan (2024) presenting a framework for scalable SaaS in resource-constrained environments.

2.2.3 Modular Monolithic vs Microservices Architecture

Architectural style significantly affects scalability and maintainability in multi-tenant systems. The two most common approaches are microservices and modular monoliths.

Microservices architecture decomposes a system into independently deployable services that communicate through lightweight APIs (Newman, 2019). It offers flexibility, fault isolation, and

horizontal scalability but often increases complexity due to service orchestration, distributed data management, and network latency (Balalaie, Heydarnoori, & Jamshidi, 2016).

Modular monolithic architecture, by contrast, maintains a single deployable unit while internally dividing the system into independent modules. Each module encapsulates specific functionality and can evolve with minimal interference to others (Taibi, Lenarduzzi, & Pahl, 2017). This approach provides a balance between simplicity and separation of concerns. Recent studies have shown that modular monoliths can achieve similar scalability to microservices when combined with containerized deployments and efficient caching (Kaurova & Murzabulatova, 2021; Abdul, Bass, Ghavimi, & MacRae, 2018).

For food delivery platforms targeting small to medium enterprises, modular monoliths offer a practical advantage. They allow multi-tenant isolation within one codebase, lower hosting costs, and easier debugging compared to fully distributed microservice ecosystems.

2.3 Theoretical Framework

The theoretical framework defines the underlying principles guiding this research. It draws from four key theoretical constructs: multi-tenancy as an architectural principle, data isolation models, resource sharing and scalability, and tenant-aware request handling.

2.3.1 Multi-Tenancy as an Architectural Principle

Multi-tenancy is the ability of a single system to serve multiple tenants while preserving logical isolation between them. It promotes scalability and efficient use of shared resources (Walraven et al., 2011). In a food delivery context, each restaurant acts as a tenant that manages its orders, menus, and customers independently. The theory of multi-tenancy supports the development of backend systems where different clients coexist without operational conflict.

2.3.2 Data Isolation Models in Multi-Tenancy

According to Azeez et al. (2010), data isolation can be achieved through three common models:

1. **Database-per-tenant** – Each tenant has its own database. This ensures maximum security but increases overhead.

2. **Schema-per-tenant** – Tenants share one database but maintain separate schemas. This balances isolation and resource use.
3. **Shared schema with tenant identifiers** – All tenants share the same schema, and data separation is enforced through tenant IDs or unique keys.

This study applies the third model because it offers efficiency and scalability while preserving tenant boundaries through strict query filtering and request-level validation.

2.3.3 Resource Sharing and Scalability

Chong and Carraro (2006) describe scalability as the balance between shared resource efficiency and tenant independence. In shared environments, resources such as servers and databases must be dynamically allocated to meet varying tenant demands. This principle ensures that while restaurants share infrastructure, one tenant's heavy load does not degrade the performance of others. The proposed system builds on this theory by using modular resource separation and asynchronous background jobs for equitable performance.

2.3.4 Tenant Context and Request Handling

Bezemer et al. (2019) emphasize the need for runtime tenant resolution, where each request is identified and processed in its correct tenant context. This approach prevents data leakage and ensures accountability. Common strategies include subdomain-based routing, token-based tenant mapping, and user-to-tenant associations through authentication tokens. The system in this research adopts token-based resolution, ensuring that every request is tagged with a tenant identifier before any database interaction.

Summary of Theoretical Framework

These theories collectively support the foundation of the proposed system: multi-tenancy ensures multi-client operation, data isolation secures information boundaries, scalability principles ensure stable performance, and tenant-aware request handling enforces contextual integrity. Together, they enable a modular monolithic backend that behaves as an independent system for each restaurant.

2.4 Empirical Review

The empirical review evaluates the implementation and outcomes of multi-tenant architectures in software-as-a-service (SaaS) applications and their potential relevance to food delivery systems. This section synthesizes findings from existing studies, industry reports, and platform analyses. It is organized into four main subsections: multi-tenancy in SaaS, data security and privacy in multi-tenant systems, applications in food delivery platforms, and performance and scalability considerations. Each subsection critically examines the literature and identifies gaps that inform the current research.

2.4.1 Studies on Multi-Tenancy in SaaS Applications

Multi-tenancy allows a single software instance to serve multiple tenants while maintaining logical separation of data and configuration. Walraven, Truyen, and Joosen (2011) propose a middleware layer that uses dependency injection and tenant-aware service layers to enable flexible multi-tenant applications. Their prototype demonstrated that tenant-specific configurations could be managed efficiently, reducing infrastructure costs without significant overhead. This foundational study illustrates the role of middleware in supporting tenant isolation and flexibility.

Azeez et al. (2010) extended these principles to service-oriented architectures (SOA) in cloud computing, presenting a middleware framework that allows legacy applications to operate in a multi-tenant environment. The authors demonstrate how shared application instances can reduce deployment and operational costs. Both studies confirm that middleware-based approaches can enhance flexibility and cost efficiency, but they are primarily conceptual or prototypical, with limited direct evidence in commercial food delivery systems or SME contexts.

Newman (2019) provides practical guidance on transitioning from monolithic to modular or microservice architectures. His work emphasizes that modular monoliths can support multi-tenant operations by isolating logical modules within a single deployable unit. Taibi, Lenarduzzi, and Pahl (2017) complement this by investigating motivations and challenges for migrating to microservices, highlighting that SMEs often adopt hybrid approaches to balance maintainability and operational simplicity. Pushpan (2024) specifically addresses lightweight multi-tenant

architectures, providing a framework for scalable SaaS in resource-constrained environments, which aligns closely with the objectives of SME-targeted food delivery systems.

Chong and Carraro (2006) advocate for architectures capable of supporting the “long tail” of customers, emphasizing the need for flexible, efficient, and cost-effective multi-tenancy. Collectively, these studies demonstrate a progression from conceptual middleware frameworks to practical architecture patterns suitable for small to medium-sized enterprises, providing a foundation for developing SME-focused multi-tenant food delivery backends.

2.4.2 Data Security and Privacy in Multi-Tenant Systems

Data isolation and security are critical challenges in multi-tenant systems. Walraven et al. (2011) highlight that ensuring tenant-specific data separation and configuration flexibility is essential for maintaining system integrity. Their middleware approach offers a mechanism for isolating tenant-specific metadata and application state, mitigating risks associated with shared environments.

Similarly, Azeez et al. (2010) discuss how multi-tenant SOA middleware enables multiple tenants to share application instances while maintaining logical separation of resources. The study emphasizes the importance of middleware-enforced boundaries, which is particularly relevant when handling sensitive data such as restaurant orders or customer information.

Bezemer, Zaidman, Platzbeecker, and Lago (2019) examine the maintenance implications of multi-tenant SaaS applications, finding that the strength of isolation mechanisms directly influences system maintainability and operational reliability. Strong isolation reduces the risk of cascading failures and simplifies debugging, whereas weak isolation can result in unpredictable performance and increased operational complexity.

While these studies provide a theoretical and prototype-level understanding of tenant isolation, empirical evidence regarding data security and privacy in operational food delivery systems remains sparse. Specifically, the literature lacks detailed measurements of potential data leakage, the effectiveness of middleware in production systems, or tenant-level access control in SME-targeted platforms.

2.4.3 Applications in Food Delivery Platforms

Food delivery platforms represent practical implementations of multi-tenant principles. Although peer-reviewed academic studies directly examining multi-tenant food delivery systems are limited, industry reports and platform analyses provide insight into operational models. Uber Eats manages billions of orders across independent restaurant tenants using centralized, tenant-aware systems (Fu & Soman, 2021). DoorDash implements a multi-tenant logistics system that optimizes delivery assignments without cross-tenant data interference (Liu, Ma, Wang, & Zhao, 2020).

Jumia Food, operating in emerging markets, adapts these principles to local constraints, balancing infrastructure costs, connectivity limitations, and vendor-specific requirements (Nwaiwu, 2020). These platforms demonstrate that multi-tenancy is practical and scalable at large volumes; however, their architectures rely on distributed microservices, complex DevOps practices, and substantial infrastructure, which may not be feasible for SMEs.

Studies examining consumer behavior, such as Ray, Dhir, Bala, and Kaur (2019), indicate that the effectiveness of these platforms depends not only on technical performance but also on usability and reliability of vendor dashboards, order tracking, and notifications. Kumar and Anbanandam (2020) further emphasize the operational risks and interdependencies in food delivery supply chains, highlighting the need for systems that maintain performance and isolation even under high load.

2.4.4 Performance and Scalability

Empirical evaluation of multi-tenant system performance remains critical. Walraven et al. (2011) demonstrate that prototype multi-tenant middleware reduces CPU consumption compared to separate single-tenant instances under their test scenarios. Azeez et al. (2010) similarly show that shared SOA instances reduce resource usage and deployment costs. These findings support the potential efficiency gains of multi-tenant architectures.

However, modular monoliths have not been widely evaluated in operational food delivery systems. Newman (2019) and Kaurova and Murzabulatova (2021) suggest that modular

monoliths can achieve comparable scalability and maintainability to microservices while reducing deployment complexity. Pushpan (2024) confirms that lightweight modular designs can maintain predictable performance under multi-tenant workloads, making them suitable for SMEs that cannot afford large-scale infrastructure.

Despite these insights, few studies quantify operational metrics relevant to restaurant dashboards, such as order processing latency, vendor reporting responsiveness, or tenant-specific performance isolation. Consequently, there is a research gap in evaluating modular multi-tenant designs in applied food delivery contexts.

Summary

The literature reviewed demonstrates that:

1. Multi-tenant SaaS frameworks reduce cost, enable scalability, and support flexible tenant configurations.
2. Middleware and SOA-based approaches provide mechanisms for tenant isolation, though empirical validation in SME-specific contexts is limited.
3. Large-scale food delivery platforms successfully implement multi-tenancy, but typically rely on complex microservices and distributed infrastructures.
4. Modular monolithic architectures present a promising alternative for SMEs, offering strong isolation, manageable complexity, and predictable performance.

This review identifies clear gaps in empirical research on modular multi-tenant food delivery systems, particularly for SME-focused platforms with integrated dashboards. These gaps justify the current study's focus on designing a modular multi-tenant food delivery platform that combines cost efficiency, performance, and secure tenant isolation.

Table 2.1 – Key Empirical Studies:

#	Study	Architecture Focus	Key Findings	Limitations / Gap
1	Walraven, Truyen, & Joosen (2011)	Middleware SaaS multi-tenancy	Flexible middleware, reduces CPU usage	Prototype, general SaaS, not food delivery
2	Azeez et al. (2010)	SOA middleware	Shared application instances reduce cost	Enterprise focus, no food delivery context
3	Newman (2019)	Modular monoliths	Modularity improves maintainability and scalability	Conceptual guidance, limited empirical validation
4	Taibi et al. (2017)	Microservices migration	Migration improves scalability	SME adoption specifics limited
5	Pushpan (2024)	SME-oriented modular SaaS	Lightweight, predictable performance	Limited real-world food delivery validation
6	Fu & Soman	Uber Eats	Tenant-aware centralized	Operational details limited in academic

	(2021)	platform	management	literature
7	Liu et al. (2020)	DoorDash logistics	Tenant-aware routing, performance optimization	Metrics not publicly available
8	Nwaiwu (2020)	Jumia Food	Multi-tenant adapted for emerging markets	Focused on market adaptation
9	Walraven et al. (2011)	Multi-tenant isolation	Middleware enforces tenant separation	Prototype, not validated in real SME food delivery
10	Kaurova & Murzabulatova (2021)	Modular monoliths	Comparable isolation/performance to microservices	Conceptual, limited real-world data

2.5 Comparative Analysis of Related Studies

The comparative analysis evaluates existing literature and real-world implementations, identifying strengths, weaknesses, and research gaps relevant to this study.

2.5.1 Strengths and Weaknesses of Existing Systems

Existing literature has advanced the understanding of multi-tenancy and SaaS scalability. Walraven et al. (2011) and Azeez et al. (2010) provided the foundational isolation models still used today. Uber Eats and DoorDash demonstrated the feasibility of tenant-aware delivery

systems with strong logistics and order routing capabilities (Fu & Soman, 2021; Liu et al., 2020). Jumia Food showcased how these ideas adapt to developing markets (Nwaiwu, 2020).

However, most prior systems assume access to large budgets, extensive DevOps teams, and distributed microservice infrastructures. They are optimized for global operations rather than lightweight deployments. Few studies address the architectural needs of startups and small enterprises that require cost-effective, isolated multi-tenant backends without full microservices overhead.

2.5.2 Comparative Table of Related Studies

Study / Platform	**Architecture Type**	**Isolation Strength**	**Scalability**	**Cost Efficiency**	**Limitation**
Uber Eats (Fu & Soman, 2021)	Microservices	Strong	Very High	Low	Requires complex orchestration
DoorDash (Liu et al., 2020)	Microservices	Strong	High	Medium	High infrastructure cost
Jumia Food (Nwaiwu, 2020)	Monolithic	Weak	Medium	High	Poor isolation
Walraven et al. (2011)	Middleware SaaS	Strong	High	Medium	Lacks SME focus
Pushpan (2024)	Modular Monolith	Strong	High	High	Limited real-world validation
Proposed System	Modular Monolith	Strong	High	High	Scalable within shared infrastructure

2.5.3 Gap Analysis

While numerous studies explore multi-tenancy and SaaS scalability, there is limited focus on affordable, SME-friendly architectures that maintain tenant isolation within a modular monolithic setup. Most prior works emphasize cloud-native microservices and overlook practical implementations suitable for emerging markets. This research addresses that gap by designing and implementing a modular monolithic backend for food delivery platforms that provides strong isolation, predictable performance, and manageable complexity for startups.

2.6 Summary

This chapter has reviewed relevant theories, technologies, and studies on multi-tenancy, data isolation, and food delivery systems. It identified how existing research focuses heavily on complex microservices and enterprise platforms, leaving a gap for modular, cost-efficient solutions suitable for startups. The theoretical and empirical insights presented here form the foundation for the design and implementation of the proposed modular monolithic system, which aims to provide strong tenant isolation, scalability, and maintainability for food delivery platforms in emerging technology environments.

CHAPTER THREE

SYSTEM ANALYSIS AND DESIGN

3.1 Introduction

Building on the gaps identified in Chapter Two, this chapter presents the system design for the proposed **multi-tenant food delivery platform**. Previous studies showed that most existing systems rely on microservice-heavy architectures that are expensive and complex for startups. This chapter therefore describes a **modular monolithic architecture** that provides tenant isolation, scalability, and maintainability within a single deployable unit.

The chapter covers the overall system architecture, design goals, module interactions, database schema, and deployment strategy. Each component is designed to address limitations such as weak tenant isolation, high maintenance cost, and poor adaptability identified in the literature review.

3.2 System Design Approach

3.2.1 Overview of the Proposed Approach

The design follows a modular monolithic architecture, which organizes the system into discrete modules such as authentication, tenant management, order processing, payments, notifications, and analytics while retaining a unified codebase. This balances simplicity and scalability, allowing each module to evolve independently.

For a multi-tenant environment, this modular strategy enables restaurants to operate under isolated configurations while sharing backend infrastructure. Each tenant can customize menu settings, pricing, and branding without affecting other tenants, ensuring operational independence within a shared deployment.

3.2.1 Overview of the Proposed Approach

Architecture Style	Description	Advantages	Limitations
Monolithic	Entire system built as one unit	Simple to start and easy to deploy	Difficult to scale and risky to maintain (Balalaie et al., 2016)
Microservices	Independent services communicating via APIs	High scalability and independent deployment (Dragoni et al., 2017)	Complex orchestration and distributed data management
Modular Monolith (Adopted)	Single deployable system divided into self-contained modules	Simplifies maintenance and supports gradual scaling (Kaurova & Murzabulatova, 2021)	Requires careful module boundaries

The chosen architecture provides a practical middle ground that maintains modular boundaries for scalability while avoiding the heavy DevOps overhead associated with microservices.

3.3 Design Aims and Objectives

The system design is guided by four core objectives:

1. **Scalability:** The system must handle increasing tenants and order volumes using techniques such as Redis caching, asynchronous job queues, and database indexing.
2. **Tenant Isolation:** Enforce strict logical data separation through a shared-schema design with tenant identifiers (Aulbach et al., 2008).
3. **Maintainability:** Modularize components so that upgrades or fixes in one area, such as payments, do not affect others.

4. **Reliability:** Incorporate message queues and retry mechanisms for consistent performance even under heavy loads.

These aims directly address gaps in existing systems, particularly the lack of affordable, maintainable, and tenant-isolated architectures for small and medium-scale food delivery providers.

3.4 System Architecture

The proposed system adopts a layered modular monolithic structure consisting of five main layers: **Presentation**, **Application**, **Shared Services**, **Data**, and **Infrastructure**. Each layer encapsulates a distinct set of responsibilities while maintaining tenant awareness and flexibility.

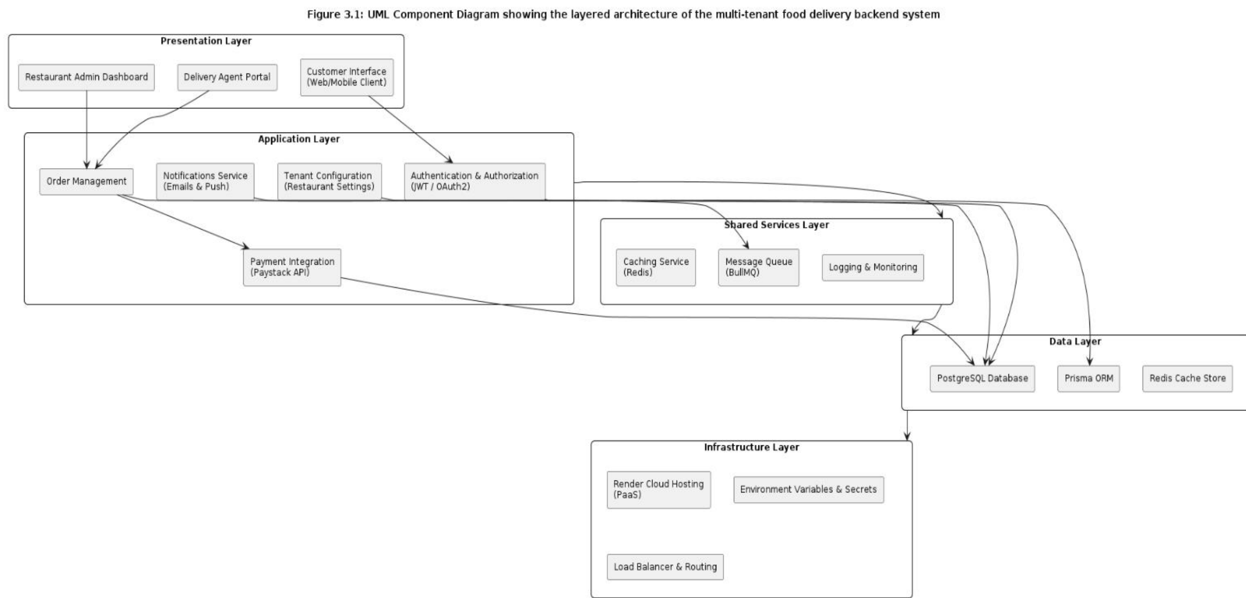

***Figure 3.1**: System Architecture Diagram illustrating the layered structure and interaction between core components.*

3.4.1 Architectural Layers

1. **Presentation Layer**
   This layer serves as the interface for customers, vendors, and delivery agents. Tenant-specific frontends communicate with backend APIs to ensure customized branding while sharing common backend logic (Chong & Carraro, 2006).

2. **Application Layer**

   This layer contains the core modules:

   - **Authentication & Authorization Module** – Manages user sign-up, login, and role-based access control using secure token-based methods (JWT).
   - **Order Management Module** – Handles order creation, tracking, and updates.
   - **Payment Module** – Integrates multiple gateways, allowing tenants to choose preferred processors
   - **Notifications Module** – Sends real-time updates using asynchronous queues.
   - **Tenant Configuration Module** – Manages tenant-specific settings such as pricing rules, delivery zones, and branding, supporting rapid onboarding of new restaurants.

3. **Shared Services Layer**

   Provides reusable services such as caching with Redis, background jobs using BullMQ, and centralized logging and monitoring.

4. **Data Layer**

   Uses PostgreSQL for primary storage with logical tenant isolation, complemented by Redis caching for high-speed retrieval.

5. **Infrastructure Layer**

   The infrastructure is hosted on **Render PaaS** which automates routing, scaling, SSL, and deployment (Render, 2025).

This layered approach ensures clear separation of concerns, tenant-aware operations, and simplified scalability.

3.4.2 Core Database Design

The database design is built on a single database, shared schema model. Each record carries a **restaurantAcronym** to enforce tenant separation. PostgreSQL is chosen for its ACID compliance, JSON support, and indexing performance (Aulbach et al., 2008; Abdul et al., 2018).

Core Table Schemas

Table 1: Users

Field	Type	Description
userID	UUID (PK)	Unique identifier for each user
name	VARCHAR(100)	User’s full name
email	VARCHAR(150)	Login email address
password	TEXT	Encrypted password
role	VARCHAR(20)	Defines user type such as customer, vendor, or admin
restaurantAcronym	VARCHAR(20)	Tenant identifier linking user to a restaurant
created_at	TIMESTAMP	Registration timestamp

Table 2: Restaurants

Field	Type	Description
restaurantID	UUID (PK)	Unique restaurant identifier
restaurantAcronym	VARCHAR(20)	Tenant code used across the system
name	VARCHAR(100)	Restaurant name
address	TEXT	Physical address
regNo	TEXT	Restaurant registration number
domain	TEXT	Client-side restaurant-specific website url

created_at	TIMESTAMP	Registration date

Table 3: Meals

Field	**Type**	**Description**
mealID	UUID (PK)	Menu item identifier
acronym	VARCHAR(20)	Tenant link
category	VARCHAR(50)	Menu category
name	VARCHAR(100)	Item name
price	DECIMAL(10,2)	Item price
availability	BOOLEAN	Availability status
created_at	TIMESTAMP	Creation date

Table 4: Orders

Field	**Type**	**Description**
orderID	UUID (PK)	Unique order identifier
restaurantAcronym	VARCHAR(20)	Tenant link
customerID	UUID (FK)	Linked to the user placing the order
totalAmount	DECIMAL(10,2)	Total cost of the order
status	VARCHAR(30)	Order lifecycle status

created_at	TIMESTAMP	Order creation date
updated_at	TIMESTAMP	Last status update

Table 5: Transactions

Field	**Type**	**Description**
transactionID	UUID (PK)	Transaction identifier
orderID	UUID (FK)	Linked order
restaurantAcronym	VARCHAR(20)	Tenant link
payment_method	VARCHAR(50)	Payment type such as Paystack or AnyPay(crypto)
amount	DECIMAL(10,2)	Transaction amount
reference	TEXT	The user-friendly ID is sent to the customer and restaurant for referencing the transaction.
payment_status	VARCHAR (20)	Pending, Successful, or Failed
created_at	TIMESTAMP	Transaction timestamp

All tables include the restaurantAcronym field to enforce logical isolation across tenants.

3.4.3 Relationships and Normalization

The schema follows Third Normal Form (3NF) to minimize redundancy and maintain referential integrity. Foreign key relationships, such as **order_id** linking Orders to Transactions, ensure consistency. Indexes are applied to fields like **restaurantAcronym**, **order_status**, and **payment_status** to optimize query speed.

3.4.4 Entity-Relationship Diagram

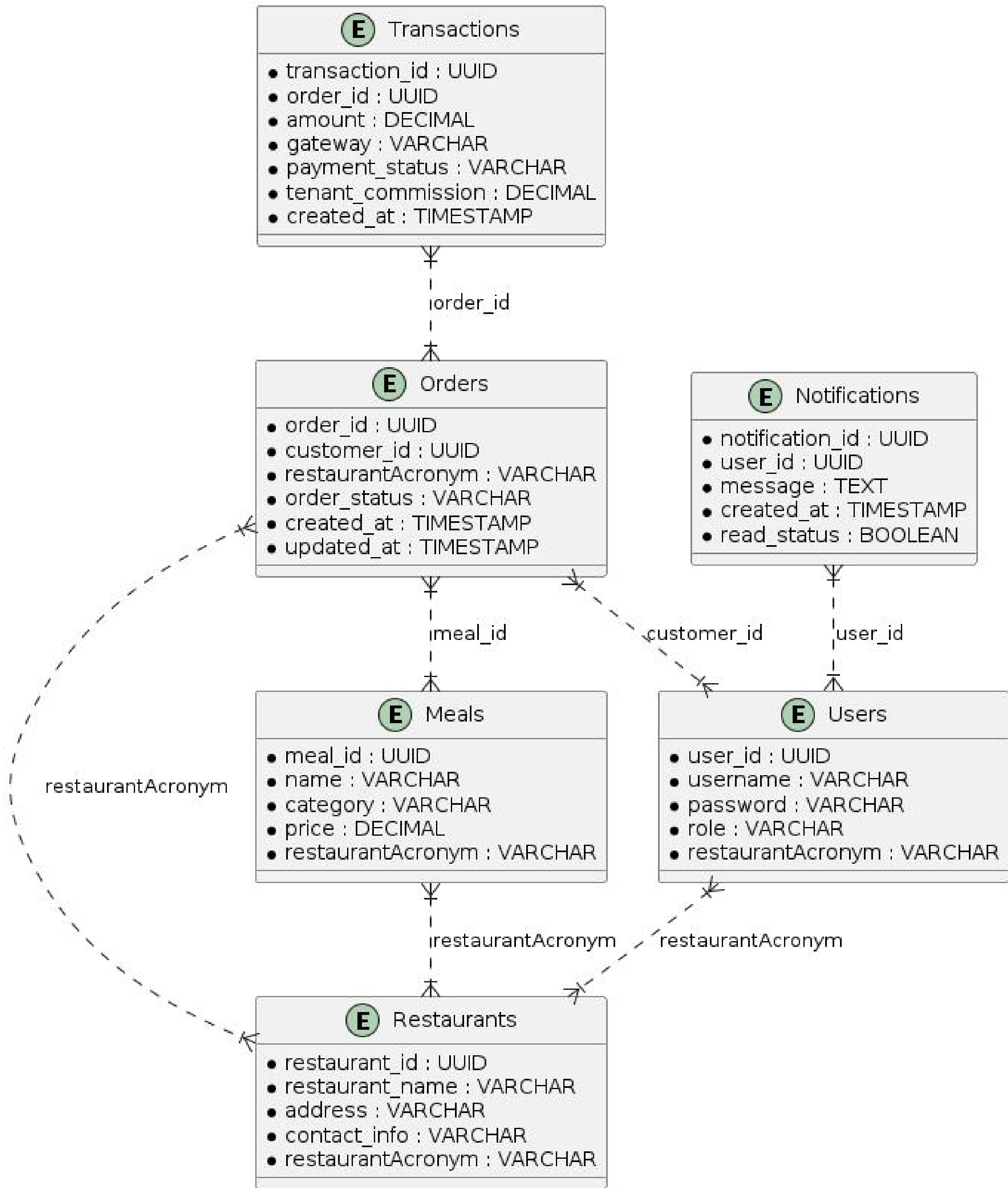

Figure 3.2: *Entity-Relationship Diagram showing the logical relationships among Users, Restaurants, Meals, Orders, Transactions, and Tenant Configurations.*

3.5 Module Design

The proposed multi-tenant food delivery system is designed as a set of loosely coupled, yet highly cohesive modules. Each module performs a specific set of responsibilities, ensuring that the system remains scalable, maintainable, and easy to extend. By adopting this modular design, data is able to flow seamlessly across components while preserving tenant isolation. According to Sommerville (2016), modular decomposition enhances maintainability and reliability in large-scale systems, while Fowler (2015) emphasizes its role in supporting scalability in distributed environments.

The following modules form the backbone of the system:

3.5.1 Authentication and User Management Module

This module handles registration, login, authentication, and authorization of users, including customers, restaurant owners, and delivery agents. It supports multiple tenants by associating users with specific tenant IDs. Authentication tokens (JWT) ensure secure access to tenant-specific data.

Data flow:

1. Customers register/login → credentials validated → JWT token generated.
2. Tenant ID embedded in JWT → directs user access rights to appropriate tenant database partitions.
3. Requests are routed to downstream modules (e.g., Order Management) with authentication headers.

3.5.2 Restaurant and Menu Management Module

Restaurant owners can create and update their digital storefront, including menus, categories, and item availability. Tenant isolation ensures that one restaurant's menu cannot be accessed or altered by another tenant.

Data flow:

1. Restaurant admin sends requests → data validated by backend → menu stored in PostgreSQL (with tenant ID).
2. Customers browsing the app request menu → API retrieves only menu items belonging to the requested tenant.

3.5.3 Order Management Module

This is the central module responsible for creating, updating, and tracking customer orders. It integrates with both the Menu Management and Payment modules. Orders transition through statuses: *Pending → Confirmed → Out for Delivery → Completed.*

Data flow:

1. Customer selects menu items → cart created → order record generated in database.
2. Order details passed to Payment Module for verification.
3. Once confirmed, order status updates trigger events to both Transactions and Notification Module.

3.5.4 Payment Processing Module

This module handles secure payments through tenant-preferred gateways, giving tenants the flexibility to accept fiat payments via AnyPay or cryptocurrency payments via Paystack, based on their configuration. It ensures that payment transactions are securely processed, verified, and recorded while maintaining tenant-specific isolation.

Data flow:

1. Customer places an order → order data (amount, tenant ID, customer ID, items bought) is sent to the Payment Processing Module.
2. Payment module routes the request to the appropriate gateway (AnyPay for fiat, Paystack for crypto).

3. Gateway returns a success or failure response, which is logged in the database.
4. On success → the Order Management Module updates the order status to confirmed.
5. On failure → the system triggers a rollback and notifies the customer of the failed transaction.

3.5.5 Transactions Module

The Transactions Module is responsible for recording and managing all financial activities within the system, including order payments, refunds, and tenant commission calculations. It ensures that every transaction is securely logged and tied to the appropriate tenant and customer account, supporting both fiat and cryptocurrency payments.

Data flow:

1. Order status update event published by Order Management.
2. Transaction Module subscribes → records the transaction in the database and calculates tenant commissions or system fees.
3. Transaction status updates → triggers notification to tenant and customer via the Notification Module.
4. Any failed or disputed transactions → flagged for review and rollback if necessary.

3.5.6 Notification Module

Facilitates real-time updates to customers and vendors using push notifications or emails. It uses queue management with BullMQ and Redis to decouple notifications from core processing.

Data flow:

1. Order status update event published by Order Management.
2. Notification module subscribes → sends real-time update to customer/vendor.
3. Failures queued for retry without affecting main transaction.

3.5.7 Analytics and Reporting Module

Generates insights for tenants (e.g., sales reports, customer preferences).

Data flow:

1. Raw transaction/order data pulled from PostgreSQL.
2. Data aggregated per tenant → visualized through dashboards.
3. Reports delivered to restaurant admins periodically.

3.5.8 Data Flow Diagram (DFD)

To illustrate the flow of information, **a Level-1 Data Flow Diagram (DFD)** is presented below showing how customer orders flow through Authentication, Order Management, Transactions, and Notification modules, while ensuring tenant-aware data handling and isolation.

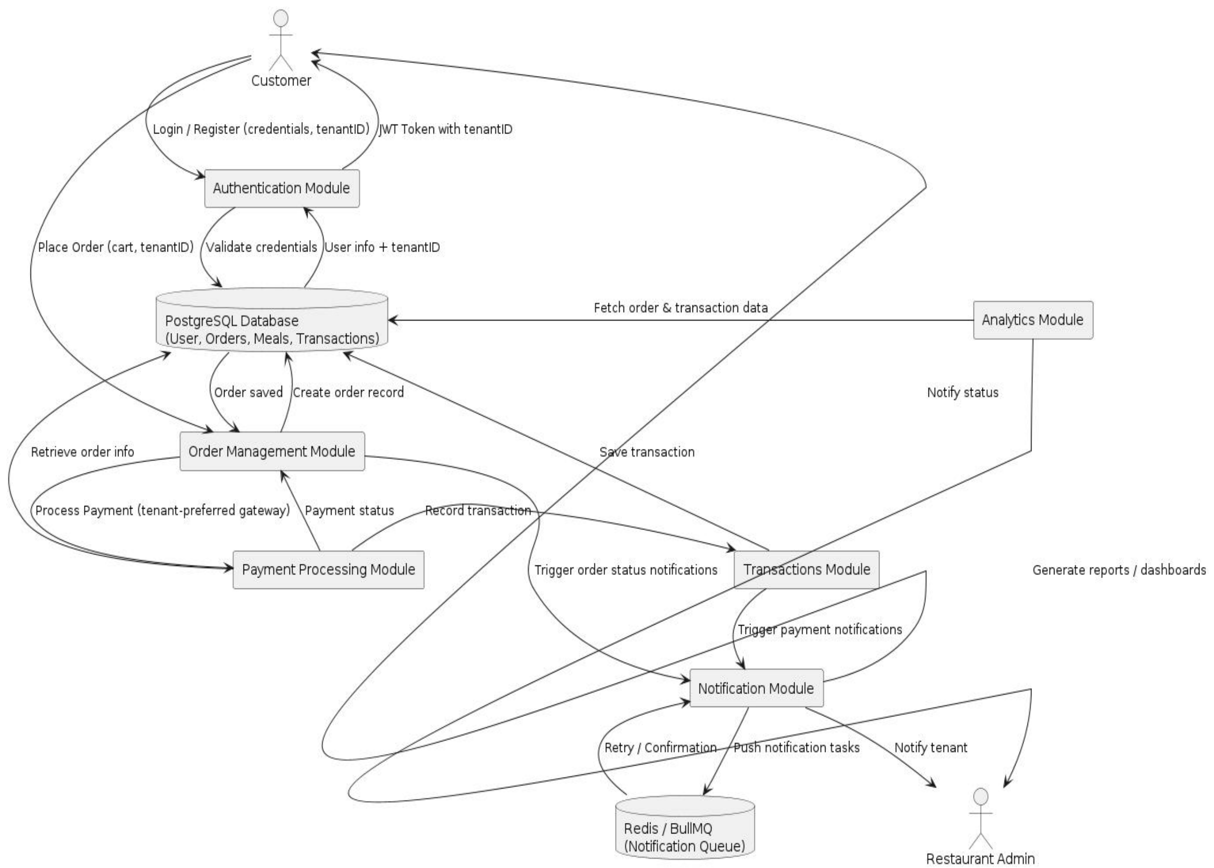

Figure 3.3: Level-1 Data Flow Diagram (DFD) showing interactions among major modules and data stores in the multi-tenant food delivery backend system.

3.6 Software Development Life Cycle (SDLC)

Given the dynamic nature of multi-tenant platforms, where tenants may require **custom configurations, rapid onboarding, and frequent feature updates**, the **Agile methodology** has been selected for the software development process. Agile emphasizes **iterative development, frequent feedback, and incremental delivery**, which aligns well with the **modular architecture** of the platform.

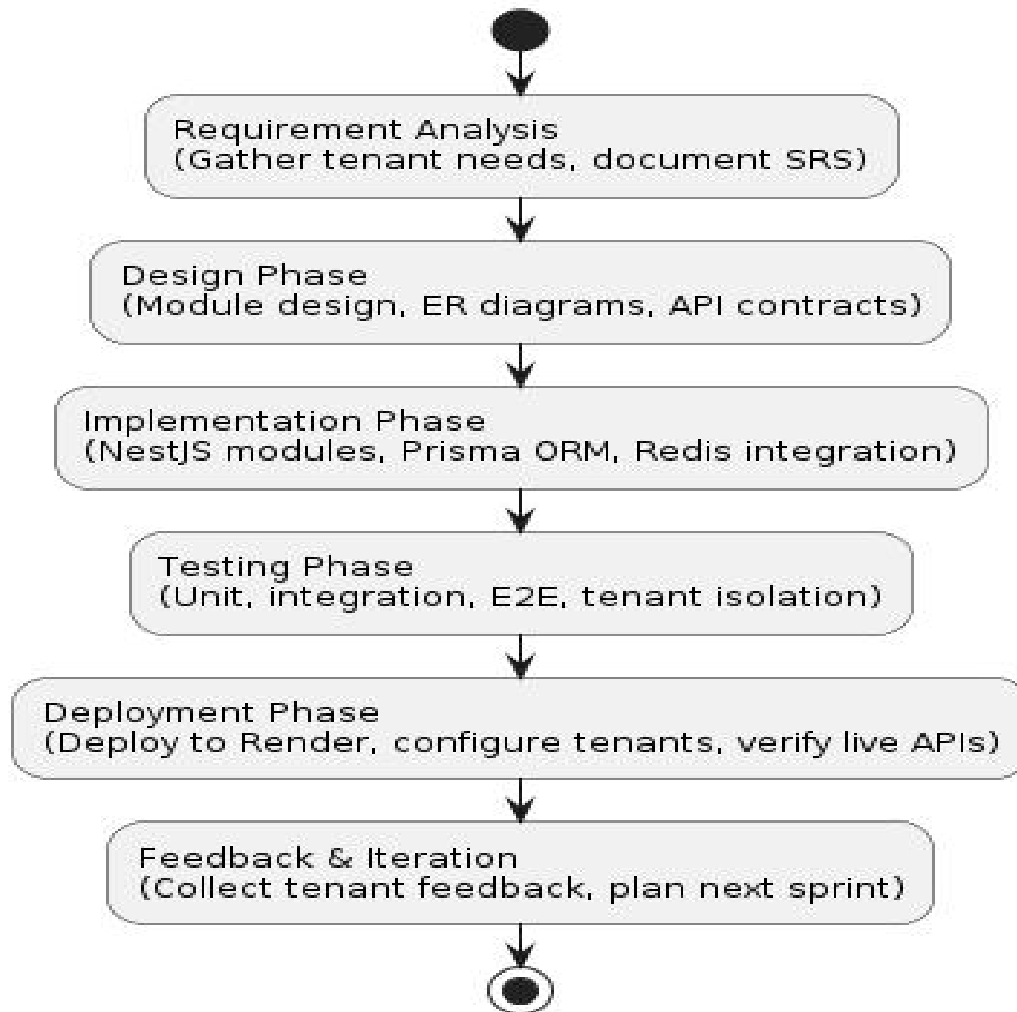

Figure 3.4: High-level Agile SDLC illustrating all phases from Requirement Analysis to Deployment, highlighting iterative feedback loops.

3.6.1 Requirement Analysis Phase

1. Gather functional and non-functional requirements from potential tenants and users.
2. Identify key modules such as authentication, order management, payment, notifications, and tenant configuration.
3. Define performance, scalability, and isolation constraints.

Output: Software Requirements Specification (SRS).

3.6.2 Design Phase

1. Create architecture, module interaction, and ER diagrams.
2. Define APIs and data flow between modules.
3. Choose technology stack such as NestJS, PostgreSQL, Redis, and Render.

Output: Design Document Specification (DDS).

3.6.3 Implementation Phase

1. Develop modules using NestJS with TypeScript.
2. Implement PostgreSQL schema with Prisma ORM.
3. Integrate Redis for caching and BullMQ for background jobs.
4. Ensure tenant-aware data handling in all modules.

3.6.4 Testing Phase

1. Conduct unit and integration tests.
2. Perform end-to-end tests for order, payment, and notification workflows.
3. Verify tenant isolation security.

3.6.5 Deployment Phase

1. Deploy the system on Render PaaS.
2. Ensure environment variable security and tenant routing.
3. Automate builds and scaling through CI/CD pipelines.

3.7 Technology Stack

Layer	**Technology**	**Purpose**
Backend Framework	NestJS (TypeScript)	Modular backend structure with strong typing
Database	PostgreSQL with Prisma	Secure, relational data storage with tenant

	ORM	isolation
Caching and Jobs	Redis with BullMQ	High-speed caching and background processing
Authentication	JWT	Secure multi-tenant access control
Deployment	Render PaaS	Automated scaling, routing, and SSL
Version Control	Git with CI/CD	Continuous integration and collaboration

The selected technology stack supports the **modular, multi-tenant design**, ensures **high performance, security, and scalability**, and allows **rapid development and deployment** of tenant-specific features.

3.8 Deployment and Infrastructure Design

This section outlines how the modular multi-tenant food delivery system is deployed, managed, and maintained in production environments. The backend is hosted on **Render**, a modern Platform-as-a-Service (PaaS) solution that abstracts server and container management, while the restaurant dashboard frontend is hosted on **Vercel**, which provides automatic build optimization, CDN distribution, and edge caching.

Render handles server provisioning, container orchestration, automatic routing, and SSL termination, allowing the backend to scale seamlessly without manual intervention. Continuous Integration and Continuous Deployment (CI/CD) pipelines are configured to push updates directly from the Git repository, ensuring version-controlled and reliable releases. Vercel automatically builds and deploys the dashboard whenever updates are pushed to the repository, providing near-instantaneous availability to restaurant users.

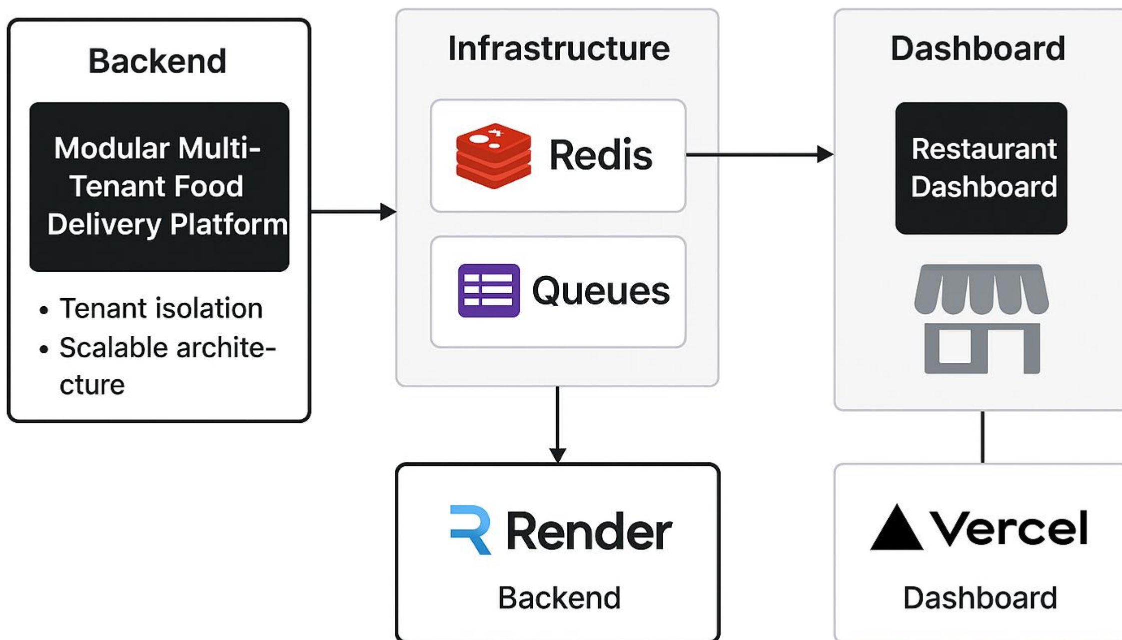

Figure 3.5: Deployment diagram showing Render hosting the backend, Vercel hosting the dashboard, Redis and queues, and the tenant isolation flow

3.8.1 Scalability and Performance

The system implements multiple strategies to maintain responsiveness under variable workloads:

1. **Automatic scaling**: Render dynamically adjusts backend resources to handle traffic spikes, ensuring low latency during peak order periods.
2. **Caching**: Redis is used to cache frequently accessed data, reducing database load.
3. **Asynchronous processing**: Queues handle long-running tasks such as notifications and reporting in the background, preventing request blocking.
4. **Tenant isolation**: Each restaurant tenant is logically separated using a `restaurantAcronym` identifier, preventing cross-tenant interference and ensuring consistent performance across tenants.

3.8.2 Security Considerations

Security is enforced at multiple layers to protect tenant data and system integrity:

1. **HTTPS enforcement**: All traffic is secured via SSL/TLS certificates provided by Render and Vercel.
2. **Tenant access control**: JWT-based authentication ensures that requests are correctly associated with the corresponding restaurant tenant.
3. **Role-based authorization**: Users are assigned specific roles (e.g., restaurant admin, staff) to control access to functionality.
4. **Environment isolation**: Separate staging and production environments are maintained for the backend and dashboard, minimizing the risk of unintended data exposure or deployment errors.

3.9 System Workflow

The workflow of the proposed multi-tenant food delivery system demonstrates how modules interact seamlessly to manage orders, payments, and notifications while maintaining tenant isolation. Unlike the module-level workflows described in Section 3.3, this section focuses on the **integrated end-to-end flow**, illustrating how customer actions propagate through the system in a modular monolith architecture.

When a customer places an order, the Authentication Module first validates their credentials and ensures correct tenant association. Once authenticated, the order is recorded by the Order Management Module, embedding the tenant identifier to prevent cross-tenant data access. Payment is processed via the Transactions Module, which routes the request to the tenant-preferred gateway which can be AnyPay for cryptocurrency or Paystack for fiat payments. The payment result, whether successful or failed, is reflected in the Order Management Module, which simultaneously triggers notifications to inform both the customer and the tenant. The tenant then confirms the order, completing the end-to-end workflow.

These interactions are visualized in **Figure 3.5**, a high-level UML sequence diagram that captures the chronological flow of messages among the customer, tenant, and system modules. The diagram emphasizes **tenant-specific handling** and the order in which operations occur, providing a clear representation of the integrated workflow.

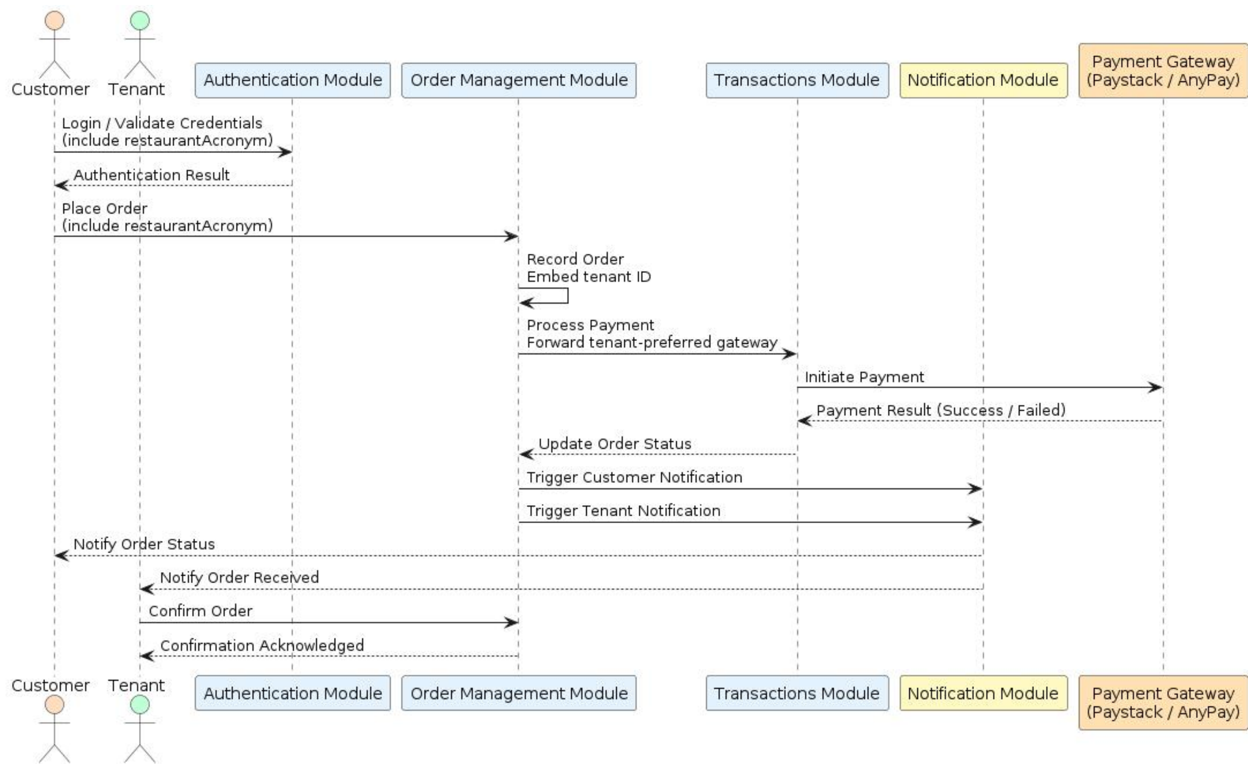

Figure 3.6: *High-level UML sequence diagram showing integrated workflow across all modules of the multi-tenant food delivery* system*, from customer order placement to payment processing, tenant confirmation, and notifications.*

To highlight parallel and asynchronous processes, an **activity diagram** is presented in **Figure 3.6**. While the Transactions Module interacts with payment gateways and updates order records, the Notifications Module operates concurrently to deliver real-time updates to the customer and tenant. This concurrency ensures responsiveness and reliability, even during high order volumes, and reinforces the efficiency of the modular monolith approach.

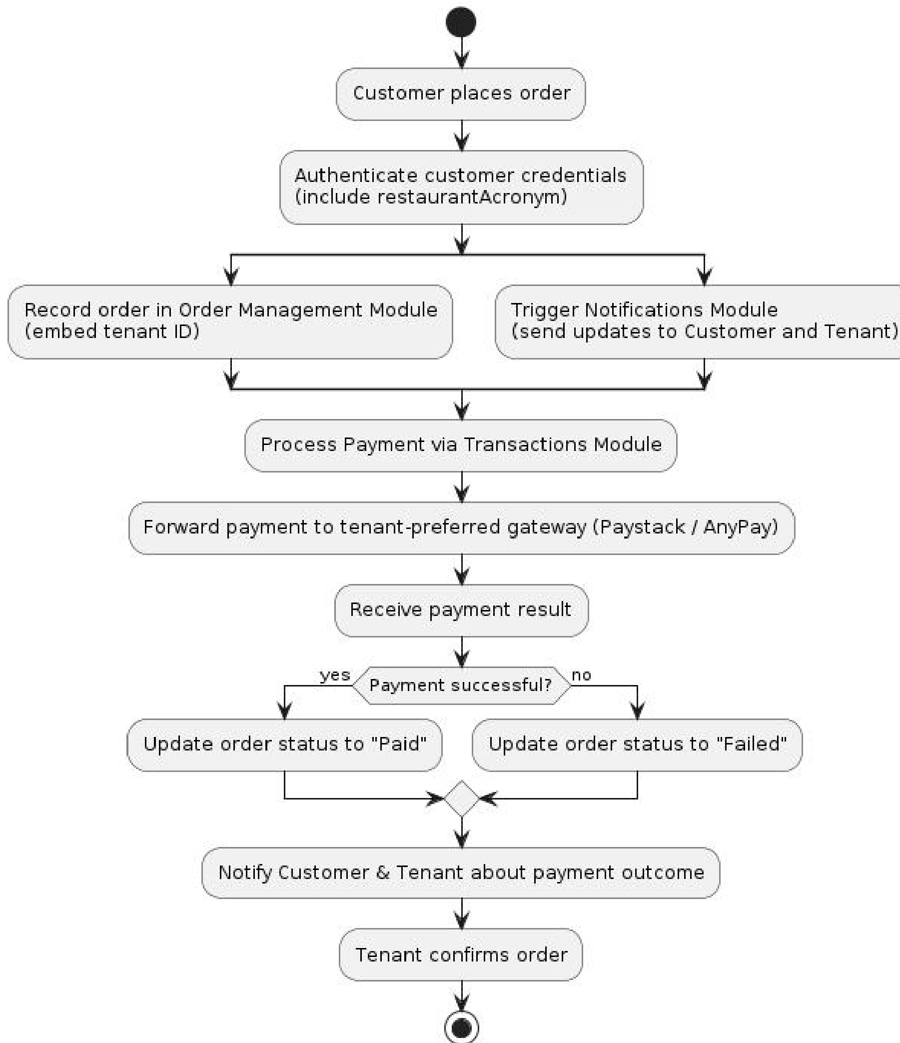

Figure 3.7: *Activity diagram illustrating concurrent module interactions, showing asynchronous notifications and parallel payment processing in the integrated system workflow.*

Overall, the system workflow demonstrates how **modular components coordinate efficiently**, ensuring data integrity, tenant isolation, and real-time responsiveness. By integrating sequence and activity diagrams, this section provides a comprehensive view of end-to-end operations, bridging the module-level design from Section 3.5 with the deployment strategies discussed in Section 3.8. The visual representations support understanding of how the system maintains **scalability, maintainability, and robust multi-tenant management**.

3.9 Summary

This chapter presented the system design for the proposed multi-tenant modular food delivery platform. The design resolves the gaps identified in existing systems by integrating modular architecture, shared-schema isolation, and scalable cloud deployment. Each module, from authentication to analytics, operates independently yet cohesively within the modular monolith.

Through its architectural layers, normalized database schema, and Agile development lifecycle, the design ensures scalability, maintainability, and strong tenant isolation. The next chapter will focus on the implementation and testing of the designed system.

CHAPTER FOUR

SYSTEM IMPLEMENTATION

4.1 Introduction

This chapter presents the implementation of the proposed multi-tenant food delivery system. It focuses on how the design developed in Chapter Three was translated into a functional, tenant-aware web application. The system was implemented using **NestJS**, **TypeScript**, and **PostgreSQL** for the backend, while the dashboard interface was developed with **Next.js**.

Each implementation step focused on modularity, scalability, and tenant isolation. The chapter covers the development environment, code structure, module implementation, database configuration, workflows, testing, and deployment. Visual evidence from the working system, including screenshots of the restaurant dashboard and other user interfaces, is presented throughout.

4.2 Development Environment and Tools

The implementation was carried out in a controlled environment using modern web technologies optimized for performance and modular design. The table below summarizes the key tools and technologies used in the project.

Tool/Technology	**Purpose**
NestJS (TypeScript)	Backend framework providing modular structure and dependency injection.
PostgreSQL + Prisma ORM	Relational database for structured, tenant-isolated data storage.
Next.js (React)	Frontend framework for building server-side rendered and interactive interfaces.
Redis + BullMQ	Caching and asynchronous task management for background operations.
Paystack API	Handles tenant-based payment processing and verification.
Cloudinary	Manages image uploads for menus and restaurant profiles.
Render Cloud Platform	Deployment environment for backend and database hosting.
Postman	Used for API testing and endpoint validation.

(Table 4.1: Summary of development tools and their functions.)

The backend folder structure under the ***src/*** directory was organized by modules (e.g., Auth, Orders, Payments, Notifications), ensuring clarity and scalability.

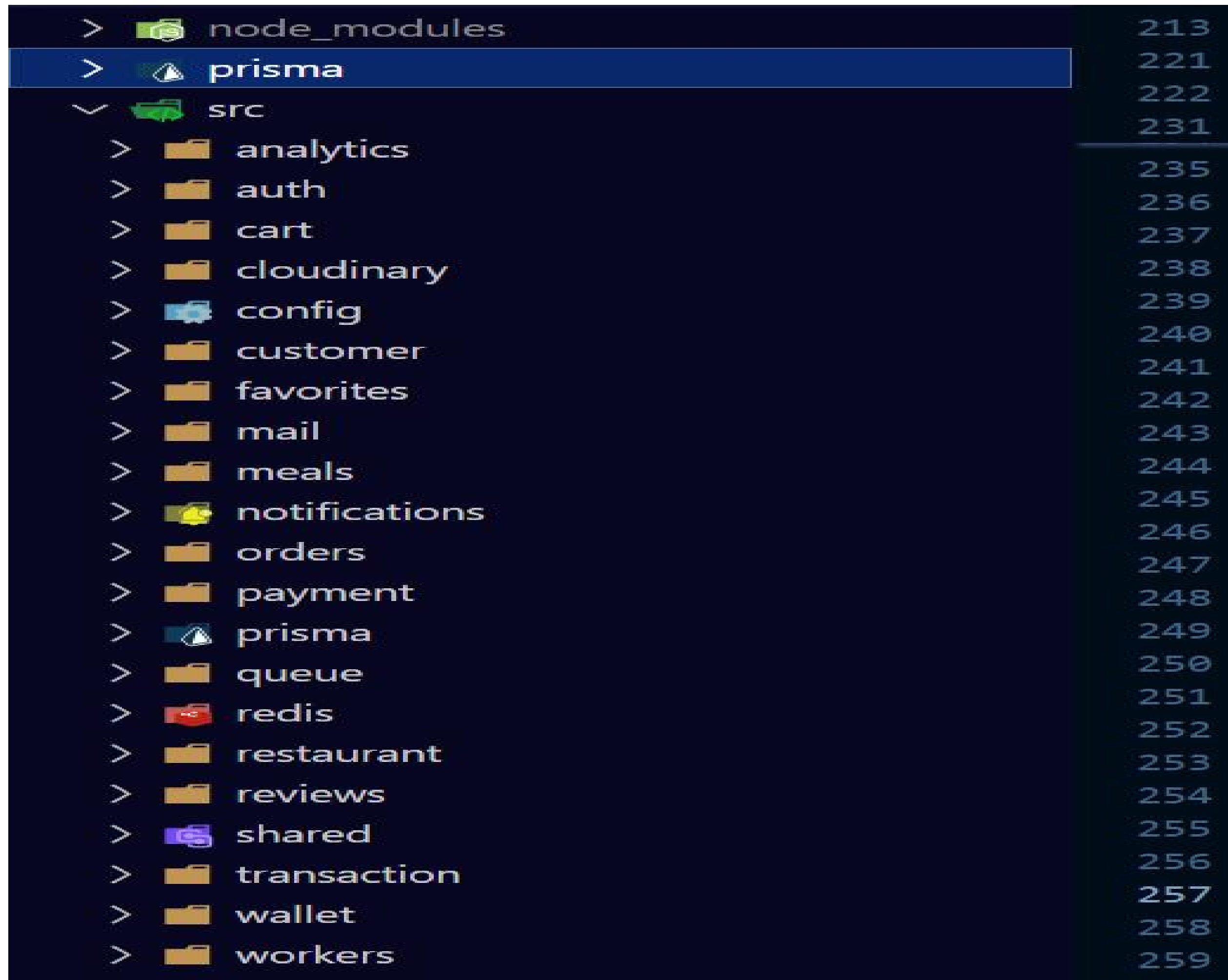

Figure 4.1: *Screenshot of project folder structure showing modular organization under the* ***src/*** *directory.*

4.3 Implementation Overview

The backend follows the **modular monolithic architecture** described in Chapter Three. Each module contains three major layers:

1. **Controller Layer:** Handles incoming HTTP requests.
2. **Service Layer:** Implements business logic and coordinates database operations.
3. **Data Access Layer:** Uses Prisma ORM to enforce tenant-based data access.

Each frontend component communicates with backend APIs scoped to a tenant, ensuring data isolation. Shared services such as notifications and payment gateways operate asynchronously to improve system responsiveness.

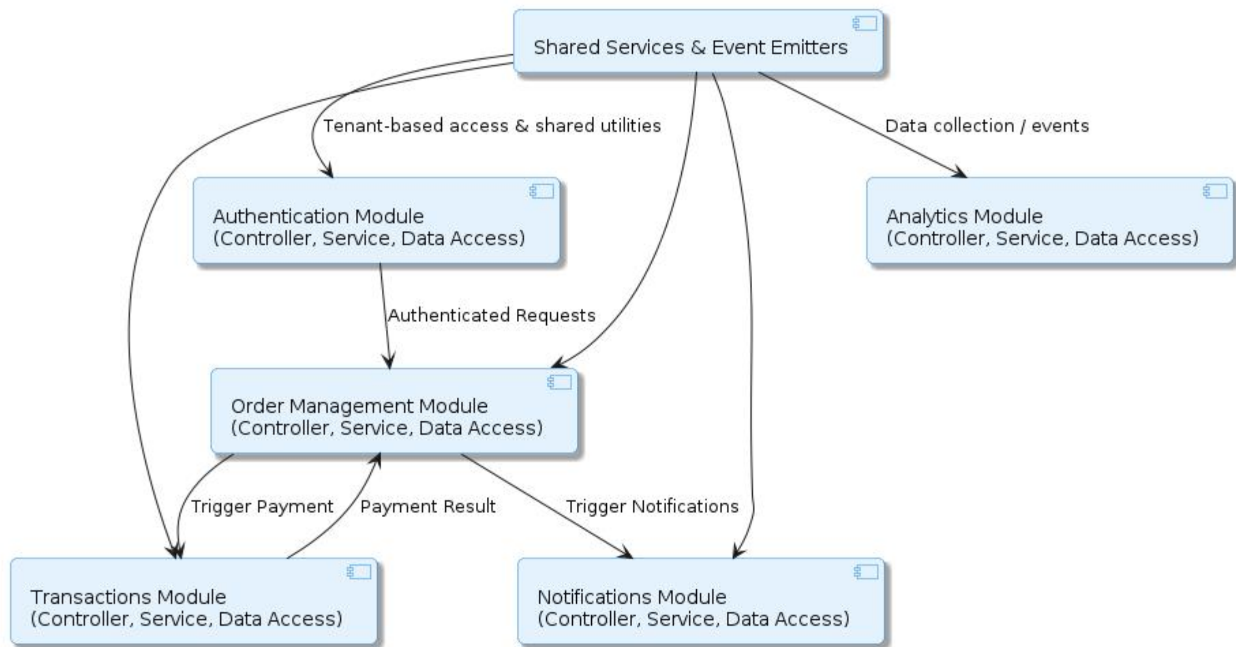

Figure 4.2: *UML Component Diagram showing module structure and interactions between core components (Auth, Orders, Payments, Notifications, Analytics).*

4.4 Core Module Implementations

4.4.1 Authentication and Tenant Login

The **Authentication Module** handles user registration, login, and token-based access control. Each JWT token embeds the tenantId, which is later used to isolate all subsequent operations for that tenant.

Code Snippet 4.1: *User authentication and tenant context binding.*

```
async validateUser(email: string, pass: string): Promise< restaurant | null> {
const restaurant = await this.prisma.restaurant.findUniqueOrThrow({  where: { email } });
```

```
const passwordMatch = await bcrypt.compare(dto.password, restaurant.password);
    if (!passwordMatch) {
      throw new UnauthorizedException('Invalid password.');
    }
const { password, ...data } = restaurant;
return await this.authService.generateJWTPayload({ id: user.restaurantID, data,email: user.email, name: user.name, message: 'Login successful!', role: user.role }); }
```

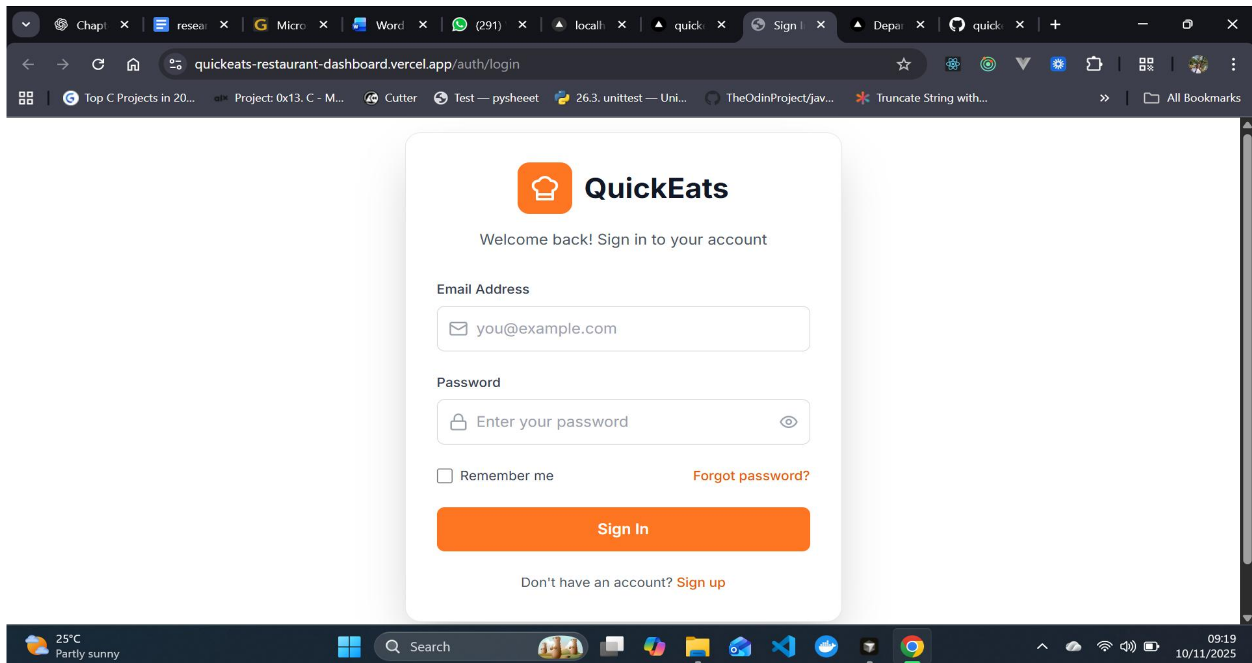

Figure 4.3: *Screenshot of the login interface showing tenant-based access control.*

4.4.2 Restaurant and Menu Management

The restaurant dashboard enables vendors to manage their profiles, menus, and orders. Each restaurant operates independently under its **restaurantAcronym**.

The interface includes forms for adding menu items, setting prices, and managing categories.

Code Snippet 4.3: *Tenant-aware menu creation logic.*

```
async createMeal(restaurantAcronym: string, dto: CreateMealDto) {
  try {
    const meal = await this.prisma.meal.create({
```

```
data: {
  ...dto,
  restaurant: {connect: {restaurantAcronym}}
}});
```

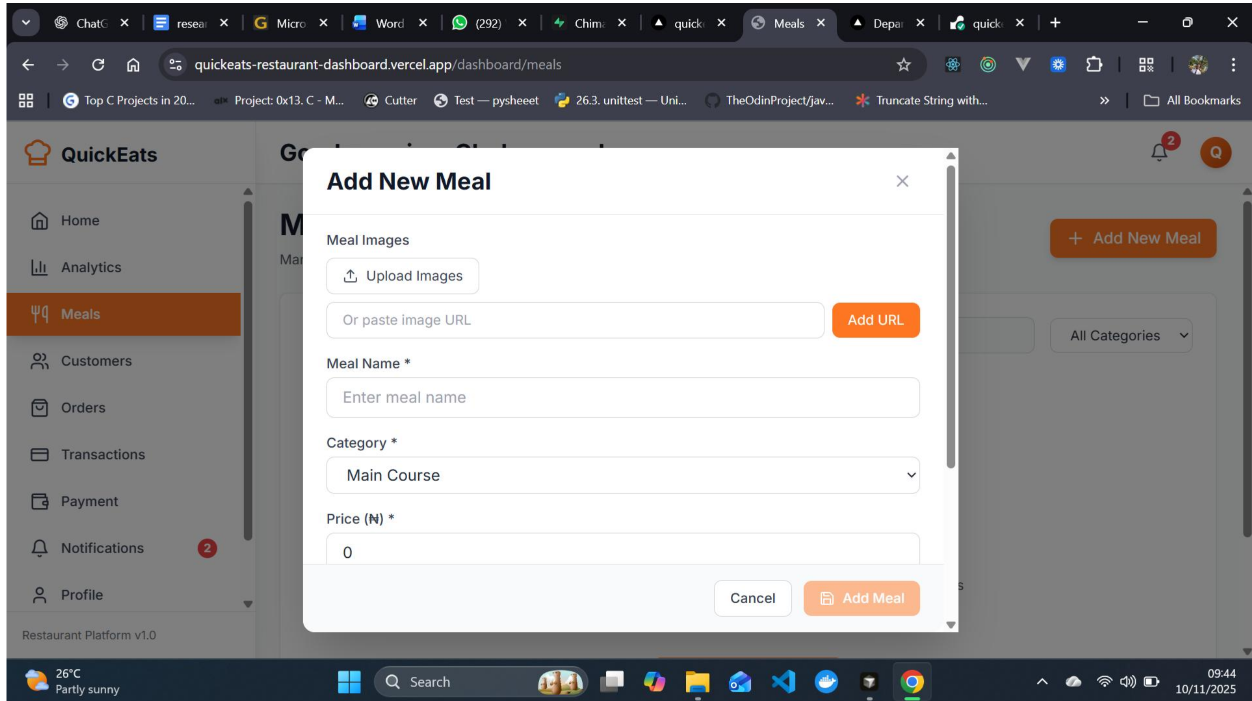

Figure 4.4: Screenshot of the restaurant dashboard showing the menu management interface.

4.4.3 Order Management

The **Orders Module** handles order creation, status updates, and coordination with the **Transactions Module**.

Tenant isolation ensures that orders created under one restaurant are never accessible by another

Code Snippet 4.4: *Order creation scoped to tenant.*

```
// orders.service.ts
async createOrder(userID: string, acronym: string) {
        const cart = await this.prisma.cart.findUnique({
                where: { customerID: userID },
                select: {
```

```
        items: true,
        totalAmount: true,
      },
    });

  return this.prisma.order.create({
    data: {
          customer: { connect: { customerID: userID } },
          status: 'pending',
          totalAmount: cart.totalAmount,
          restaurants: { connect: { acronym } },
          items: {
            create: cart.items.map((item) => ({
              amount: item.amount,
              quantity: item.quantity,
              mealID: item.mealID,
            })),
          },},
})}
```

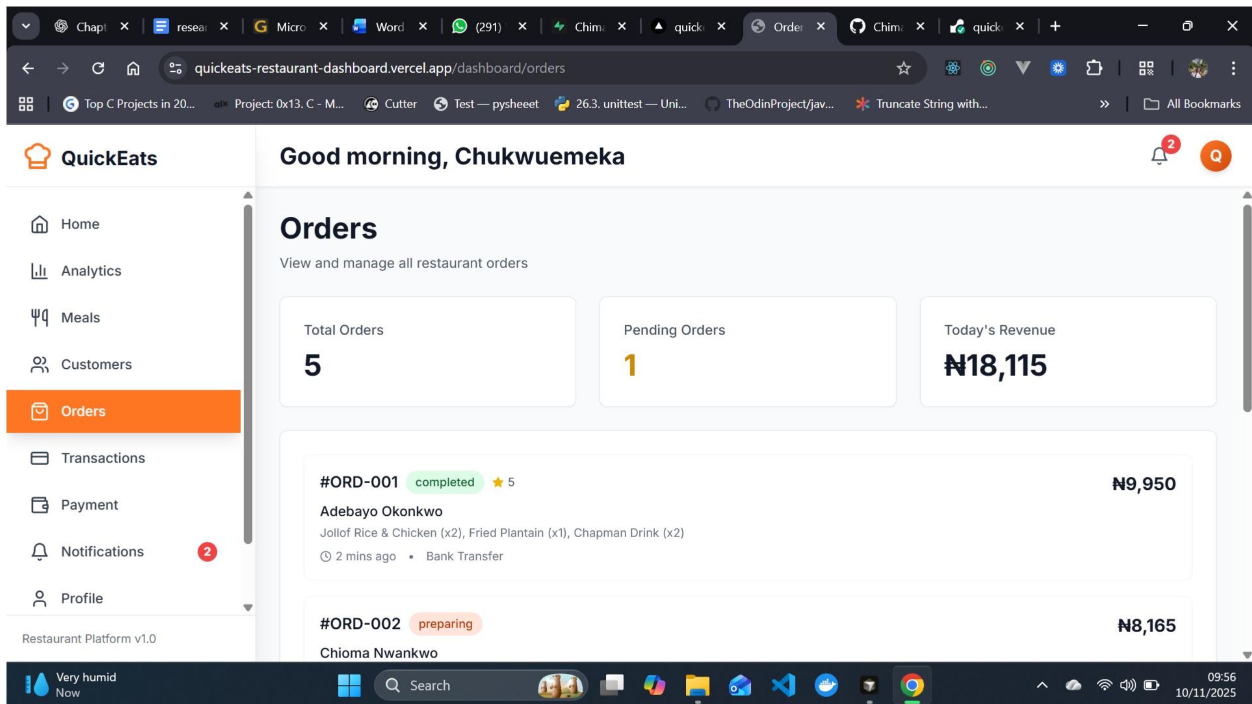

Figure 4.5: *Screenshot of the restaurant dashboard showing the list of active and completed orders.*

4.4.4 Payment Integration

The **Payment Module** integrates with Paystack and handles secure verification.

The Paystack secret key is stored as an environment variable which is dynamically retrieved when needed.

Code Snippet 4.5: *Order creation scoped to tenant.*

```
// payment.service.ts
async verifyPayment(reference: string, tenantId: string) {
        const tenant = await this.prisma.restaurant.findUnique({ where: { acronym:
      restaurantAcronym} });
        const response = await
      axios.get(`https://api.paystack.co/transaction/verify/${reference}`, {
          headers: { Authorization: `Bearer ${process.env.paystackSecretKey}` },
        });
        if (response.data.status) {
```

```
                await this.prisma.transaction.create({
                    data: {
                      reference,
                      tenantId,
                      status: 'SUCCESS',
                      amount: response.data.data.amount / 100,
                    },
                });
        }
}
```

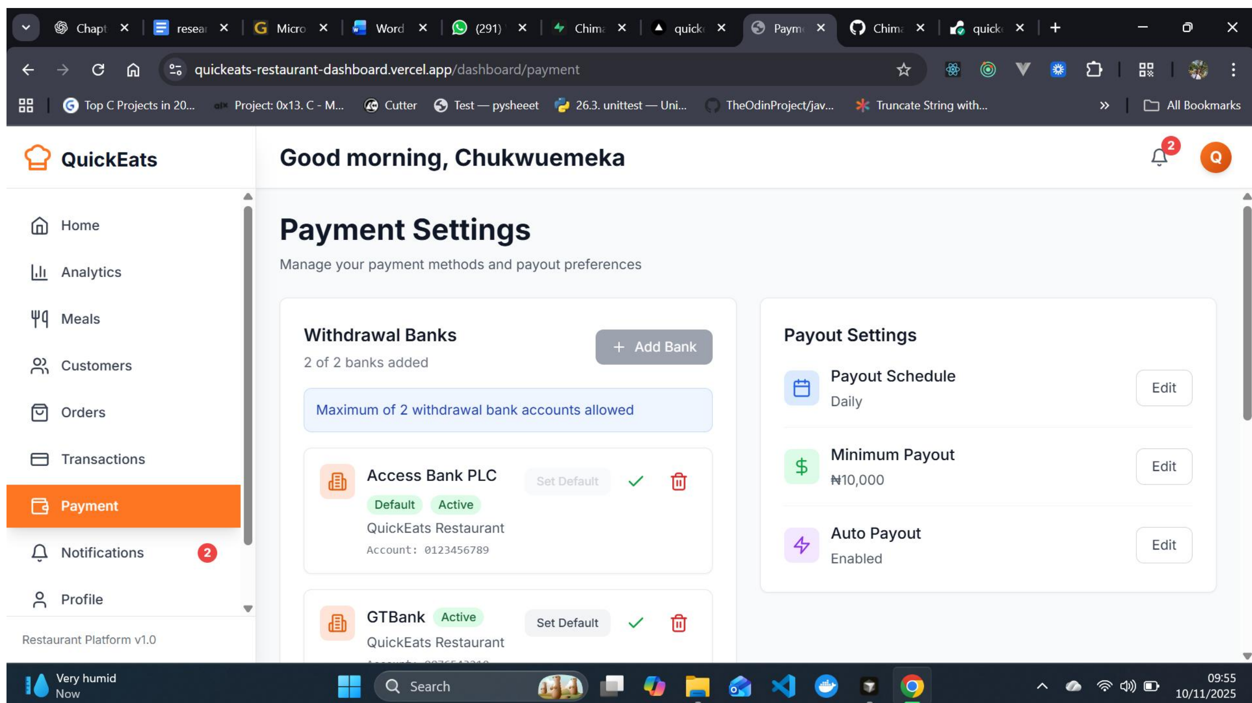

Figure 4.6: *Screenshot of payment page.*

4.4.5 Notifications and Queue Workers

The **Notifications Module** handles asynchronous email and in-app notifications using **BullMQ** and **Redis** queues.

This ensures that main application flows remain fast and non-blocking.

Code Snippet 4.6: *Worker queue handling asynchronous notifications.*

```
// mail.worker.ts
this.queue.process(async (job) => {
  await this.mailService.sendMail({
    to: job.data.email,
    subject: job.data.subject,
    text: job.data.message,
  });
});
```

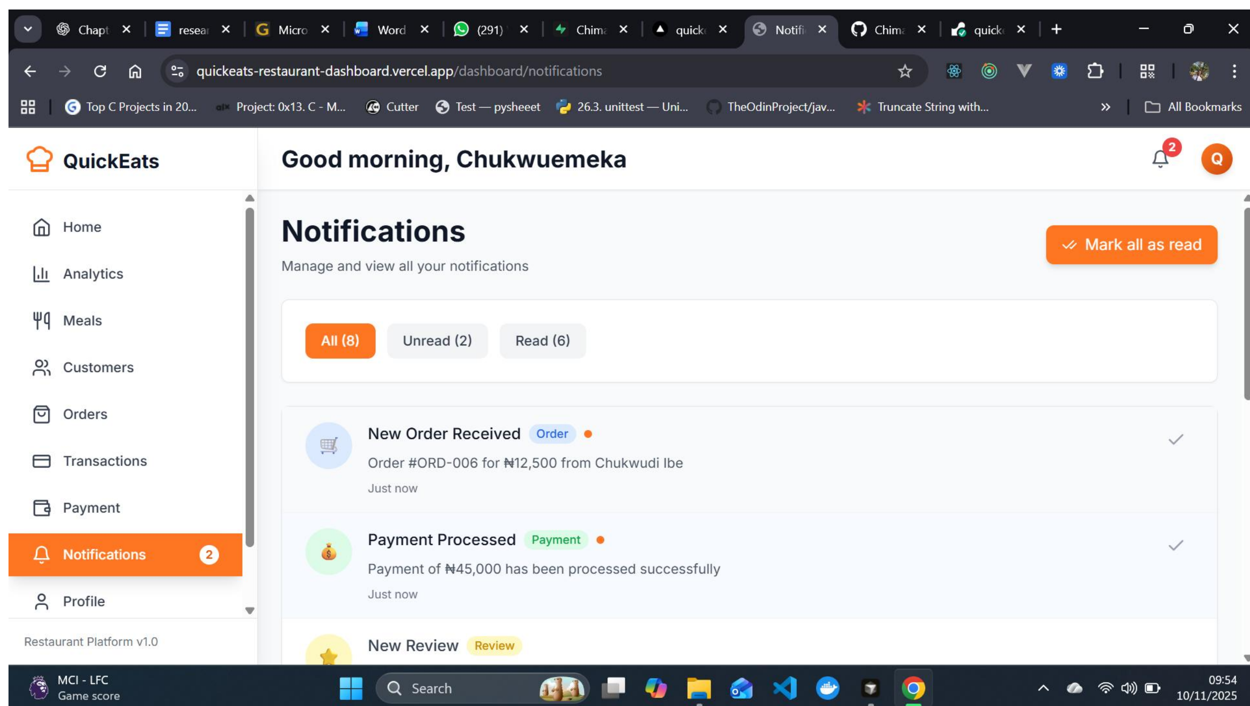

Figure 4.7: Screenshot of notification alerts showing order updates and confirmations.

4.4.6 Analytics and Reporting

The **Analytics Module** provides restaurant-level insights such as best-selling meals weekly and all-time, worst-selling meals, sales by category, daily transactions, and popular meals.

Code Snippet 4.7: *Tenant-specific daily sales aggregation.*

```
async getDailySales(acronym: string) {
```

```
  return this.prisma.transaction.groupBy({
    by: ['createdAt'],
    where: { acronym, status: 'SUCCESS' },
    _sum: { amount: true },
  });
}
```

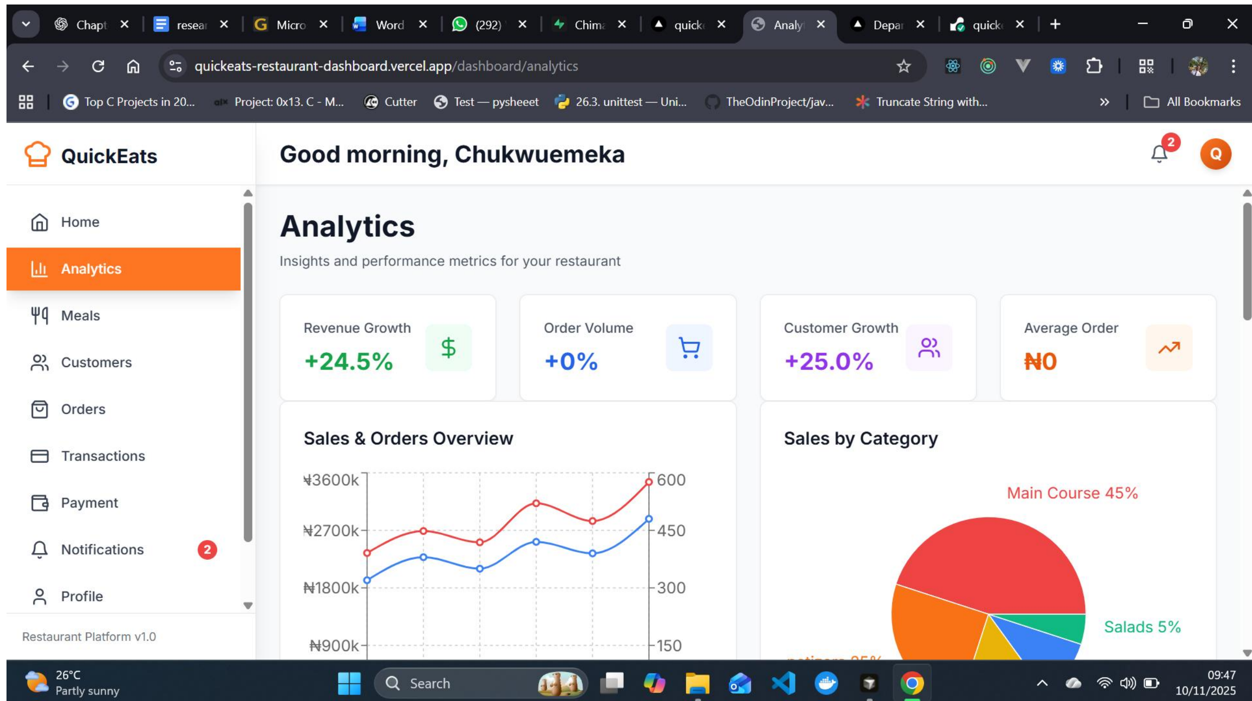

Figure 4.8: *Screenshot of analytics dashboard showing total sales, transactions, and popular menu items.*

4.5 Database Configuration

The database adopts a **shared schema with tenant identifiers** model. Each table includes a column to identify tenants for logical data partitioning.

```
model Meal {
  mealID          String   @id @default(uuid())
  images          String[]
  name            String
  thumbnail       String
  description     String
  viewCount       Int      @default(0)
  likeCount       Int      @default(0)
  dislikeCount    Int      @default(0)
  price_per_potion Float
  discount        Float    @default(0)
  ingredients     String[]
  category        String
  createdAt       DateTime @default(now())
  updatedAt       DateTime @updatedAt

  restaurantID String
  restaurant   Restaurant @relation(fields: [restaurantID], references: [restaurantID])

  favorite      Favorite[]
  favoriteCount Int         @default(0)
  ordersCount   Int         @default(0)
  reviews       Review[]
  items         CartItems[]

  orderItems OrderItems[]

  @@unique([name, restaurantID])
  @@index([restaurantID, category])
}

model Order {
  orderID     String     @id @default(uuid())
  totalAmount Int        @default(0)
  status      StatusEnum @default(pending)

  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt

  items OrderItems[]

  transaction   Transaction? @relation(fields: [transactionID], references: [transactionID])
  transactionID String?      @unique

  customer          Customer    @relation(fields: [customerID], references: [customerID], onDelete: Cascade)
  customerID        String
  restaurants       Restaurant? @relation(fields: [restaurantAcronym], references: [acronym], onDelete: Cascade)
  restaurantAcronym String
}
```

Figure 4.9: *Screenshot* of the schema of the system showing key tables and relationships.

4.6 Deployment

Security is enforced at multiple layers to protect tenant data and system integrity:

1. **HTTPS Enforcement:** All traffic is secured via SSL/TLS certificates. Render and Vercel automatically provide HTTPS for deployed applications, ensuring encrypted communication between clients and servers.

2. **Tenant Access Control:** JWT-based authentication ensures that each request is correctly associated with the corresponding restaurant tenant. Tokens are signed and verified to prevent tampering.
3. **Role-Based Authorization:** Users are assigned specific roles (e.g., restaurant admin, staff) to control access to functionality, enforcing the principle of least privilege.
4. **Environment Isolation:** Separate staging and production environments are maintained for both backend and dashboard. Vercel provides preview deployments for each branch, allowing safe testing without affecting production, while Render manages environment-specific variables to reduce the risk of unintended data exposure or deployment errors.

5. **Platform Security Features:**
    a. **Vercel:** Automatic DDoS protection, continuous security updates, and isolation between deployments. Preview deployments are ephemeral and isolated from production.
    b. **Render:** Secure default configurations, automatic OS-level updates, and network-level isolation for services.

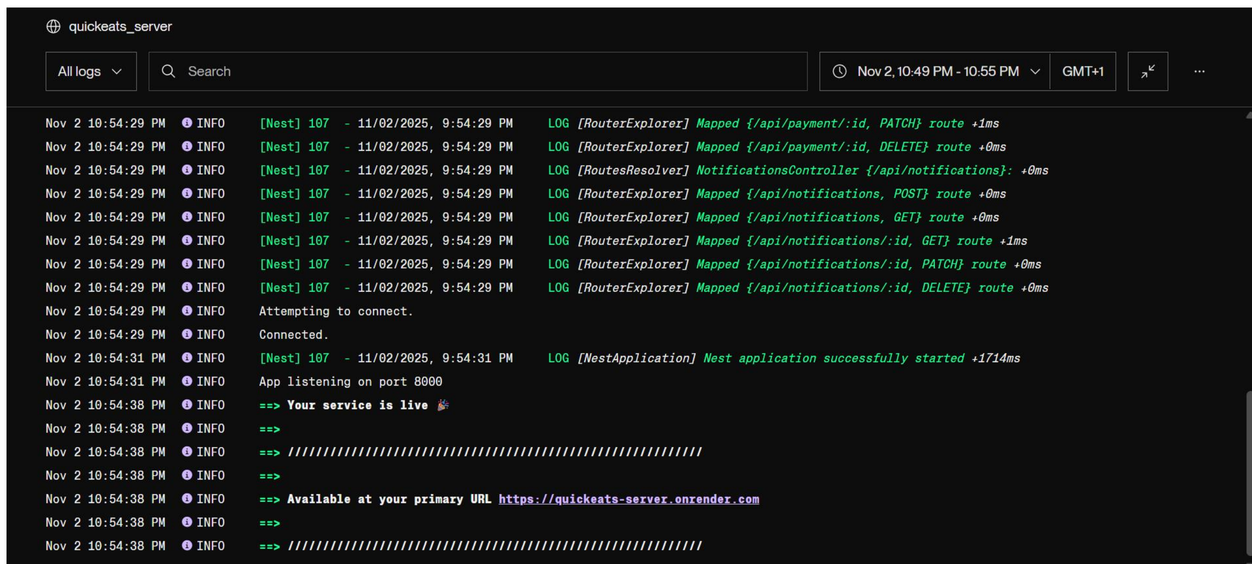

Figure 4.10 *Screenshot of Render dashboard showing successful service deployment.*

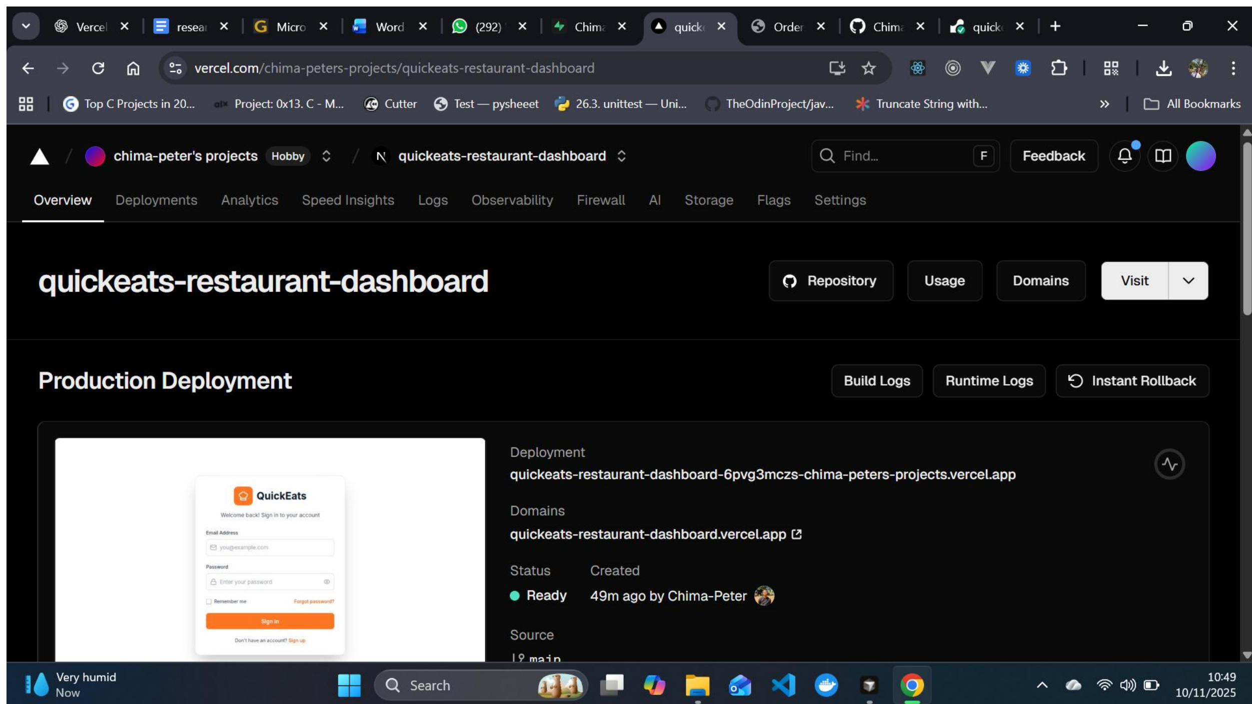

Figure 4.11 *Screenshot of Vercel overview showing successful restaurant dashboard deployment.*

4.7 Summary

This chapter presented the implementation of the multi-tenant food delivery system, integrating backend APIs and a restaurant frontend dashboard under a modular architecture. The combination of NestJS, PostgreSQL, and Next.js enabled a tenant-aware application with secure authentication, order management, and payment workflows.

Screenshots from the working dashboard demonstrated real-time interaction between modules, while database configuration and testing confirmed proper tenant isolation. Deployment on Render Cloud and Vercel ensures continuous delivery, reliability, and scalability.

The next chapter provides summary, conclusion, and recommendations derived from the overall study. It consolidates the major findings from the system design and implementation phases, highlights the contributions of this work to the field, and presents practical suggestions for future improvement and research.

CHAPTER FIVE

SUMMARY, CONCLUSION AND RECOMMENDATION

5.1 Summary

This research presented the design and implementation of a multi-tenant food delivery system aimed at providing a scalable, secure, and modular backend system for restaurant order management. The system was developed to address the problem of inefficient order handling and the lack of tenant-level data isolation that often exists in traditional food delivery applications.

The study began with an introduction to the challenges faced by multi-restaurant delivery platforms and emphasized the need for a unified system that allows multiple restaurants to operate independently within a shared infrastructure. Chapter One defined the objectives of the study, which included building a modular backend, ensuring tenant data separation, and integrating a secure payment gateway.

Chapter Two provides a comprehensive review of existing literature and related systems such as Uber Eats and DoorDash. It examined previous academic works on multi-tenancy, order management, and distributed systems. The findings from this review highlighted major limitations in scalability and data segregation within existing systems. These insights guided the design and implementation of the proposed solution.

Chapter Three focused on system analysis and design. It described the architecture, module structures, and database schema that formed the foundation of the system. UML diagrams, data

flow diagrams, and entity relationship models were used to illustrate interactions among the authentication, restaurant, order, payment, and analytics modules.

Chapter Four presented the actual implementation of the system using NestJS, TypeScript, and PostgreSQL. The system was built following a modular monolithic architecture to ensure maintainability, scalability, and efficient performance. Core modules such as authentication, restaurant management, order processing, payment integration, notifications, and analytics were implemented with strict tenant-aware data control. Integration with Paystack provided secure payment processing, while Redis and BullMQ handled asynchronous notification jobs. The entire system was deployed on the Render Cloud Platform for testing and demonstration.

In summary, the system successfully met its design objectives by combining modular architecture, tenant isolation, and cloud-based deployment to deliver a reliable and extensible backend platform for multi-restaurant food delivery operations.

5.2 Conclusion

The implementation of the multi-tenant food delivery system achieved all the key objectives of the research. The system demonstrated that it is possible to design a unified backend capable of serving multiple independent restaurant tenants without data overlap or interference. The use of a modular monolithic structure proved to be a practical choice for achieving scalability and maintainability within a single deployable unit.

The integration of Paystack confirmed the reliability of the system in processing secure online transactions, while Prisma ORM ensured efficient data modeling and integrity. The use of Redis-based queue workers optimized performance by offloading time-consuming background tasks such as notifications. Each component of the system contributed to overall efficiency, resulting in a stable, scalable, and secure backend suitable for commercial deployment.

Beyond the technical implementation, this research contributes to academic and professional discourse by demonstrating how modular monolithic systems can effectively support multi-tenancy. It also shows that tenant-aware resource management can be achieved without the complexity of full microservices, making it a practical alternative for startups and medium-scale businesses seeking cost efficiency.

5.3 Recommendations

Based on the findings and implementation outcomes, the following recommendations are made:

1. **Integration of Additional Payment Gateways:**
   Future versions of the system can incorporate other payment providers such as Flutterwave or Stripe to enhance payment flexibility and regional compatibility.
2. **Containerization and Orchestration:**
   Although the system was successfully deployed on Render, using Docker and Kubernetes for containerization would provide better scalability and easier horizontal scaling for large-scale operations.
3. **Incorporation of Delivery and Rider Tracking:**
   Extending the system to include delivery personnel management and GPS-based order tracking would enhance the real-world applicability of the system.
4. **Enhanced Monitoring and Logging:**
   Implementing a monitoring tool such as Prometheus or Grafana would allow for improved visibility into system performance and quicker detection of issues.
5. **User Interface Development:**
   A dedicated web or mobile frontend could be developed to allow customers and restaurants to interact directly with the system, transforming it into a full-stack multi-tenant food delivery solution.
6. **Further Research:**
   Additional research could compare the performance of modular monolithic and microservices architectures in multi-tenant environments to provide empirical insights for future software engineering projects.

5.4 Summary of the Chapter

This chapter provided a concise overview of the research work, from system conception and design to implementation and deployment. It concluded that the project achieved its stated objectives of modularity, scalability, and tenant isolation. The recommendations presented serve as a guide for future developers, researchers, and organizations interested in building upon the foundation established by this study.

LIST OF ACRONYMS

Acronym	Meaning
ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
AnyPay	Payment gateway for fiat/crypto (proprietary system)
CI/CD	Continuous Integration / Continuous Deployment
DDS	Design Document Specification
DFD	Data Flow Diagram
JWT	JSON Web Token
ORM	Object-Relational Mapping
PaaS	Platform-as-a-Service
Redis	Remote Dictionary Server (in-memory data store/cache)
SaaS	Software-as-a-Service
SDLC	Software Development Life Cycle
SRS	Software Requirements Specification
SSL/TLS	Secure Sockets Layer / Transport Layer Security
UUID	Universally Unique Identifier
NestJS	Node.js framework for scalable backend development
Next.js	React framework for server-side rendering and frontend development
BullMQ	Node.js queue library for background jobs and async processing
Prisma	ORM for TypeScript and Node.js to interact with relational databases
Paystack	Payment gateway for online transactions
Cloudinary	Cloud-based media management platform for images and media
JSON	JavaScript Object Notation
HTTP	Hypertext Transfer Protocol

HTTPS	Hypertext Transfer Protocol Secure
SQL	Structured Query Language
PK	Primary Key
FK	Foreign Key
CRUD	Create, Read, Update, Delete
CI	Continuous Integration
CD	Continuous Deployment
ERD	Entity-Relationship Diagram
UI	User Interface
UX	User Experience
CPU	Central Processing Unit
RAM	Random Access Memory

REFERENCES

Abdul, A. O., Bass, J. M., Ghavimi, H., & MacRae, N. (2018). Multi-tenancy design patterns in SaaS applications: A performance evaluation case study. *International Journal for Digital Society, 9*(1), 1367–1375.

Amazon Web Services (AWS). (2023). SaaS tenant isolation strategies.

Aulbach, S., Grust, T., Jacobs, D., Kemper, A., & Rittinger, J. (2008). *Multi-tenant databases for software as a service: Schema-mapping techniques*.

Azeez, A., Perera, S., Siriwardana, P., Gamage, D., Linton, R., & Weerawarana, S. (2010). Multi-tenant SOA middleware for cloud computing. *IEEE 3rd International Conference on Cloud Computing.*

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). *Microservices architecture enables DevOps: Migration to a cloud-native architecture*. IEEE Software, 33(3), 42–52.

Bezemer, C.-P., Zaidman, A., Platzbeecker, B., & Lago, P. (2019). Multi-tenant SaaS applications: Maintenance dream or nightmare? IEEE Software, 36(4), 52–59.

Chong, F., & Carraro, G. (2006). *Architecture strategies for catching the long tail*. Microsoft Corporation.

Dragoni, N., Giazzi, S., Lago, P., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). *Microservices: Yesterday, Today, and Tomorrow*. In *Present and Ulterior Software Engineering* (pp. 195–216). Springer.

Fowler, M. (2015). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Fu, Y., & Soman, C. (2021). Real-time data infrastructure at Uber.

Gartner. (2022). Best practices for designing multi-tenant SaaS architectures. Gartner Research Reports.

Kaurova, O., & Murzabulatova, A. (2021). *Analysis of modular monolithic architecture in software development*.

Kumar, A., & Anbanandam, R. (2020). Analyzing the inter-relationships of risks in food delivery supply chain using interpretive structural modeling. *Journal of Retailing and Consumer Service.*

Li, H., & Srinivasan, K. (2019). Online platforms and marketplace design. *Marketing Science.*

Liu, Y., Ma, H., Wang, S., & Zhao, L. (2020). On-demand food delivery: Platform logistics, order assignment, and last-mile optimization. *Transportation Research Part E: Logistics and Transportation Review.*

Newman, S. (2019). *Monolith to Microservices*. O'Reilly Media

Nwaiwu, F. (2020). Digital disruption in emerging markets: A case study of Jumia Food in Africa. *Journal of Business and Retail Management Research.*

Pressman, R. S., & Maxim, B. R. (2014). *Software Engineering: A Practitioner's Approach*. McGraw-Hill.

Pushpan, S. (2024). Multi-tenant architecture: A comprehensive framework for building scalable SaaS applications. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*.

Ray, A., Dhir, A., Bala, P. K., & Kaur, P. (2019). Why do people use food delivery apps? *International Journal of Hospitality Management.*

Render. (2025). *Scaling Render Services*

Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson.

Statista. (2023). Online food delivery - worldwide.

Taibi, D., Lenarduzzi, V., & Pahl, C. (2017). Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation.

Walraven, S., Truyen, E., & Joosen, W. (2011). A middleware layer for flexible and cost-efficient multi-tenant applications. *Middleware 2011: ACM/IFIP/USENIX 12th International Middleware Conference.*