AASTU

UNIVERSITY FOR INDUSTRY

Addis Ababa Science and Technology University

College of Engineering

Department of Software Engineering

ConnectX - A Centralized, Scalable Backend Platform for E-Commerce

General Comments

- please avoid redundancy
- work on formatting (ensure consistency through out the document)
- make sure every thing you write in this report reflect your current work only (designing and developing ConnectX).

Addis Ababa Science and Technology University

College of Engineering

Department of Software Engineering

Title: ConnectX - A Centralized, Scalable Backend Platform for E-Commerce

Group Members	ID
1. Abdulmajid Awol	ETS0016/13
2. Abel Atkelet	ETS0020/13
3. Adane Moges	ETS0079/13
4. Amanuel Mandefrow	ETS0122/13
5. Elias Balude	ETS0237/12

Advisor Name:Chere Lemma(MSc) Signature_____

May 20205, Addis Ababa

Acknowledgment

We would like to extend our sincere appreciation to **Addis Ababa Science and Technology University**, and in particular to the **College of Engineering** and the **Department of Software Engineering**, for providing the academic environment and institutional support that made this project possible.

Our deepest gratitude is owed to our advisor, **Mr. Chere Lemma**, for his exceptional guidance, constructive feedback, and unwavering support throughout the duration of this work. His mentorship has been invaluable to the successful completion of this project.

We also acknowledge the contributions of various scholarly resources, research publications, and industry case studies that have informed and inspired the development of **ConnectX**. These sources have played a significant role in shaping the technical and conceptual foundation of our solution

Table Content

List of Tables, Figures/Illustrations, Plates/Photographs

List of figures

Abstract

ConnectX is a developer-centric backend platform designed to simplify and accelerate the development of e-commerce applications. It addresses the challenges faced by student developers, small businesses, and early-stage startups who struggle with the technical complexity, high development costs, and limited scalability of building back-end of e-commerce from scratch.

At its core, ConnectX offers a centralized, multi-tenant backend platform powered by a robust set of RESTful APIs that handle essential e-commerce operations such as product management, user authentication, order processing, and payment integration. This API-first architecture enables developers to integrate backend functionality directly into their web or mobile applications, allowing them to focus on crafting rich frontend experiences without reinventing backend logic. To further reduce the time to launch, ConnectX includes customizable, plug-and-play storefront templates that work seamlessly with its backend APIs. These templates serve as ready-to-use blueprints for developers to rapidly build and deploy e-commerce platforms.

Additionally, the platform features a powerful and intuitive management dashboard that provides users with centralized control over their operations. From managing product listings and tracking customer orders to viewing analytics, the dashboard empowers users to efficiently oversee their e-commerce activities.

This document provides a comprehensive overview of the system's architecture, key components, and design decisions. It outlines the core functionalities of the API suite, the role of prebuilt templates, and the structure of the admin dashboard. Ultimately, ConnectX aims to make e-commerce development more accessible, flexible, and efficient—especially for teams with limited resources or technical capacity.

ConnectX

Chapter One: Introduction

The digital transformation of commerce has reshaped how businesses operate, how consumers shop, and how entrepreneurs launch their ventures. E-commerce, once limited to large corporations with substantial technical and financial resources, has become more accessible due to advances in internet infrastructure, mobile technology, and cloud-based platforms. Yet, despite this progress, significant barriers remain for a large group of aspiring entrepreneurs, particularly students, small-scale merchants, and developers with limited financial means or technical experience.

Launching an e-commerce platform typically requires a robust backend system that handles core operations such as product management, order processing, payment integration, user authentication, and product tracking. Building and maintaining such a backend is complex, time-consuming, and costly. Moreover, the lack of a secure and scalable infrastructure makes it difficult for new players to compete with well-funded and established platforms. These challenges often discourage innovation and exclude capable individuals who have creative ideas but lack the resources to execute them.

As a group of software engineering students, we experienced these challenges firsthand. During our third year, we attempted to create an e-commerce platform as a side project. While we successfully developed the frontend and backend components, we were unable to acquire inventory due to financial constraints and faced difficulty in gaining trust and credibility in the market. That project failed not because of a lack of technical ability or motivation, but because we lacked the business infrastructure and operational support needed to launch and sustain it. This experience became the inspiration for ConnectX.

ConnectX is a centralized backend-as-a-service (BaaS) platform specifically designed to lower the barriers to entry for aspiring e-commerce entrepreneurs. It aims to serve developers, students, merchants, and small businesses who want to launch an online shop without the burden of building a backend from scratch. By offering ready-made APIs, and management dashboards, ConnectX empowers users to focus on frontend development, branding, and customer engagement, while the platform takes care of backend operations. ConnectX is heavily inspired by platforms such as Snowflake and Stripe. Snowflake revolutionized data warehousing by providing a shared, scalable infrastructure, allowing organizations to focus on insights rather than infrastructure management. Similarly, Stripe transformed online payments by offering developer-friendly APIs that enable fast and secure integration of payment systems. ConnectX adapts these philosophies to the e-commerce domain by delivering modular backend services with a plug-and-play experience.

A key innovation in ConnectX is its dual product listing system. Users can either:

- Sell pre-managed, public products available on the platform (ideal for merchants without inventory),
- Or list their own private or public products, allowing verified businesses to expand the marketplace collaboratively.

This structure encourages resource sharing, competitive diversity, and faster go-to-market strategies.Verified sellers can contribute to the ecosystem while benefiting from collective growth.

Additionally, ConnectX introduces a performance-based pricing model, where users are only charged a small percentage of each successful sale, depending on their subscription tier. This means users only pay when they actually generate revenue, ensuring they are charged based on real value and not fixed costs. This model significantly lowers financial barriers, making the platform affordable and risk-free for student entrepreneurs, small merchants, and early-stage startups who may not have the capital to invest upfront. By integrating backend services like product management, secure user authentication, payment handling, product analytics, and team management into a single shared infrastructure, ConnectX enables anyone to launch a professional e-commerce platform, whether they choose to build a custom frontend or use the mobile/web templates provided. Users can manage their stores using a customizable dashboard, access API keys for integration, and verify their business identity to gain market trust.

1.1 Statement of the Problem

Despite the rapid growth of global e-commerce, aspiring entrepreneurs, students, and small-scale merchants continue to face significant challenges when attempting to launch their own online platforms. These challenges, both technical and financial, create high barriers to entry and prevent many viable business ideas from becoming successful ventures. The primary issues identified include:

1. **High Backend Development Costs:-** Building a reliable, secure, and scalable backend system for an e-commerce platform is expensive. A typical backend setup, including product management, order handling, user authentication, and payment integration, requires significant financial investment. For small businesses, development costs often start at $2,000; for medium-sized platforms, they can reach $25,000; and for enterprise-level solutions, costs may exceed $100,000 [7]. These expenses are unaffordable for students, early-stage startups, and bootstrapped entrepreneurs, effectively preventing them from entering the digital market.
2. **Lack of Technical Skills and Development Capacity:-** Many potential e-commerce entrepreneurs lack the technical skills required to build and maintain backend systems. Backend development involves complex areas such as secure authentication, product

database design, API development, and server infrastructure management [3][6][11]. Without experience in backend engineering or access to skilled developers, most individuals are unable to implement systems that are scalable, reliable, and secure, resulting in fragile platforms that cannot support long-term growth.

3. **Product Management and Operational Complexity:-** Managing the operational lifecycle of products, such as listing, updating, tracking availability, and handling orders requires robust backend tools. Startups and students, especially those without technical teams or resources, struggle to build or maintain these systems effectively. Manual processes or poorly built platforms lead to inefficiencies, inconsistencies, and user dissatisfaction, ultimately hindering business scalability.

4. **Lack of Access to Sellable Products:-** One of the core barriers faced by entrepreneurs particularly students is the inability to access or afford products to sell. Even if a developer successfully builds an e-commerce site, they often do not have the financial means to source inventory or establish supplier relationships. This lack of product availability stalls their ability to generate revenue and sustain their platform.

5. **Credibility and Market Trust Issues:-** New businesses often struggle with credibility, especially when they lack verifiable business registration or cannot provide reliable product fulfillment. This is especially problematic in emerging markets, where customer trust plays a major role in online purchasing behavior. Entrepreneurs with limited operational history may be overlooked in favor of more established competitors, making it hard for them to build a loyal customer base.

6. **Time-Consuming Development Cycles:-** Many developers spend the majority of their time building backend functionality from scratch such as setting up databases, writing authentication flows, or integrating payment gateways [7]. This delays market entry, reduces focus on business growth, and increases the risk of failure due to burnout or loss of momentum.

7. **Lack of Collaboration and Resource Sharing:-** Current e-commerce platforms offer little to no support for shared marketplaces or collaborative product listing ecosystems [12][13][14]. Developers and merchants operate in silos, each having to source their own products and manage their own operations independently. This lack of cooperation limits resource efficiency, increases duplication of effort, and stifles innovation within the entrepreneurial community.

8. **Inflexible and Rigid Pricing Models:-** Many existing backend platforms and SaaS providers use fixed or subscription-based pricing models that require users to pay regardless of whether they are making sales [7][11]. Platforms like Shopify or BigCommerce typically charge fixed monthly subscription fees starting from $30/month, regardless of traffic or sales performance. This structure can be especially difficult for student entrepreneurs or early-stage startups that haven’t yet built consistent revenue.

1.2 Objectives

1.2.1 General Objectives

The main objective of this project is to design and implement a centralized, scalable, and developer-friendly backend-as-a-service (BaaS) platform and enable aspiring entrepreneurs, small businesses, and students to rapidly launch and manage e-commerce platforms with minimal cost and within a short time.

1.2.2 Specific Objectives

-

1.3 Scope and Limitation

1.3.1 Scope

The scope of ConnectX includes the design, development, and deployment of a centralized backend-as-a-service (BaaS) platform tailored to e-commerce businesses.The major components of the project include:

1. Backend Platform Development

The goal is to develop a scalable, multi-tenant backend that supports multiple e-commerce stores running on a shared infrastructure. This involves implementing essential services such as product management, order processing, user authentication, and payment integration. The system must ensure scalability, fault tolerance, and data isolation to maintain security and reliability.

2. API Provision

- Provide a suite of well-documented RESTful APIs to allow integration of backend services into both web and mobile applications.
- Expose endpoints for user management, product operations, order lifecycle, analytics, and secure payment workflows.

The objective is to provide a suite of well-documented RESTful APIs to allow integration of backend services into both web and mobile applications. These APIs will expose endpoints for user management, product operations, order lifecycle, analytics, and secure payment workflows.

3. Build Product Listing and Selling System

- Implement a dual product listing model where users can:
    - Sell pre-listed, publicly available products provided through the platform.
    - Upload and manage their own products with the option to mark listings as public or private.
- Support categorization for niche-specific storefronts (e.g., fashion, electronics, beauty products).

4. Implement Merchant Verification System

- Implement a business verification mechanism that allows sellers to submit legal business documentation.
- Verified merchants gain access to advanced features such as public product listings, branding visibility, and higher platform credibility.
- Verification requests are reviewed and approved through the admin dashboard to maintain trust and regulatory compliance.

5. Web and Mobile Frontend Template Provision

- Provide reusable frontend templates for both web and mobile platforms, allowing users to launch e-commerce sites quickly.
- Offer flexibility for users to either use the provided templates or integrate ConnectX APIs into their own custom applications using API keys.

6. Building Admin and Merchant Dashboards

- Create a dashboard for merchants to manage products, track sales and orders, manage team roles, and monitor performance metrics.
- Provide an administrative interface for platform operators to manage business verifications, monitor platform usage, and maintain ecosystem quality.

7. Consumption-Based Pricing Model

- Adopt a commission-based pricing model where users only pay a small percentage of each successful sale, depending on their subscription plan.
- This model ensures affordability and fairness by aligning platform costs with user success and revenue generation.

1.3.1 Limitation

While ConnectX delivers a strong foundation for scalable and accessible e-commerce backend services, certain advanced features and integrations remain out of scope for the current projet work and are identified as areas for future development:

1. ConnectX does not currently offer integrated logistics management, such as shipment tracking or automated delivery coordination. Incorporating such features could improve order fulfillment workflows in future iterations.
2. Intelligent features like personalized product recommendations, sales forecasting, and behavior-based insights are not yet included. Future improvements could explore machine learning techniques to enhance user experience and business performance.
3. Integration with the platform is currently available only through RESTful APIs. The future addition of SDKs for popular frontend and mobile frameworks could enhance developer experience and speed up integration.
4. The analytics provided focus on essential metrics. More advanced insights such as customer behavior trends and conversion tracking—may be added in future updates to support data-driven decision-making.

1.4 Methodology

To ensure the successful development and deployment of the ConnectX platform, an Agile software development model was employed. This methodology was chosen due to its iterative, incremental, adaptability to change, and continuous user involvement. From the initial requirements gathering phase, the team established specific and clearly defined deliverables for each sprint, allowing effective use of time and resources across the software development life cycle.

A structured interview process was conducted with potential users, including marketplace administrators and vendors, to capture and refine the system requirements. The gathered insights were organized into functional and non-functional requirements and documented in the Software Requirements Specification (SRS).

During the system design and architecture phase, an iterative design approach was adopted. This process followed four key activities:

1. Proposing an initial high-level solution to the problem statement.
2. Modeling the proposed solution using architectural diagrams and technical documentation.
3. Evaluating the model against original requirements to ensure alignment.
4. Elaborating the design into a detailed specification that served as a blueprint for development.

The backend system design was based on a monolithic architecture, later structured into a modular monolith for scalability and maintainability. The Model-View-Controller (MVC) design pattern, provided by Django, was used to maintain a clean separation of concerns. PostgreSQL was selected as the database due to its support for advanced features such as transactions, referential integrity, and

scalability. Multi-tenancy was addressed by implementing schema-based or row-level data isolation techniques to ensure client data security and separation.

To further enhance maintainability, scalability, and modular development, the system architecture was guided by the principles of Clean Architecture. This approach was applied consistently across the backend, as well as the web and mobile frontends. By clearly separating concerns between the domain, data, and presentation layers, the architecture ensures that core business logic remains independent of external frameworks and technologies. This allowed parallel development across teams, simplified testing and debugging, and laid the groundwork for future extensibility without major refactoring.

APIs were developed using Django REST Framework (DRF) to expose application functionality, particularly in areas such as user authentication.

ion, order processing, and product listings. The APIs were designed with scalability in mind, incorporating features such as pagination, filtering, and sorting for efficient data handling.

1.4.1 Data Collection Methodology

The data required for the system design and validation was gathered through multiple methods, including structured interviews, online research, and direct observation. Interviews were held with marketplace stakeholders, particularly vendors and administrative staff, as well as with operators of popular Telegram market channels and startup teams currently working on e-commerce platforms. These discussions provided valuable insights into real-world pain points, system expectations, and common limitations in existing platforms. Observational data on current manual marketplace operations were also recorded to identify workflow inefficiencies. Additionally, similar e-commerce systems were analyzed to extract best practices and feature requirements that align with the intended solution.

1.4.2 Deciding on the Development Tools

The tools used for documentation, design, development, and deployment were carefully selected to align with the team's skill set and project needs. These tools facilitated productivity, streamlined collaboration, and supported quality software delivery.

1.4.2.1 System Design and Analysis Tools

- As part of the development process for the **ConnectX** platform, our team adopted a variety of tools to support the planning, analysis, design, and collaboration efforts. To gather, prepare, and organize system requirements and produce a comprehensive Software Requirements Specification (SRS), we utilized **Google Docs** and **Google Forms**. These tools facilitated efficient collaboration, documentation, and feedback collection from stakeholders.
- For the system analysis and design phase, we selected **PlantUML** as our UML modeling tool. PlantUML was chosen due to its support for object-oriented modeling and its ability to

generate clear, consistent, and maintainable diagrams, which significantly contributed to the preparation of our System Design Document (SDD).

- To manage our tasks, monitor progress, and ensure smooth collaboration throughout the development of ConnectX, we employed the **GitHub Kanban board**. This tool allowed us to break down work into manageable issues, assign responsibilities, and track progress across various milestones in a transparent and organized manner.
- Furthermore, for the UI/UX design of the ConnectX platform, we chose **Figma**. Figma provided a robust collaborative environment with real-time editing, component-based design systems, and interactive prototyping features that greatly enhanced the efficiency and quality of our design process.

Table 1. System Design and Analysis Tools

No.	Tool	Purpose
1	Google Docs	For documentation
2	Google Forms	To gather data and questionnaires
3	PlantUML	For system modeling
4	GitHub Kanban Board	Project Management and Collaboration
5	Figma	UI/UX design tool

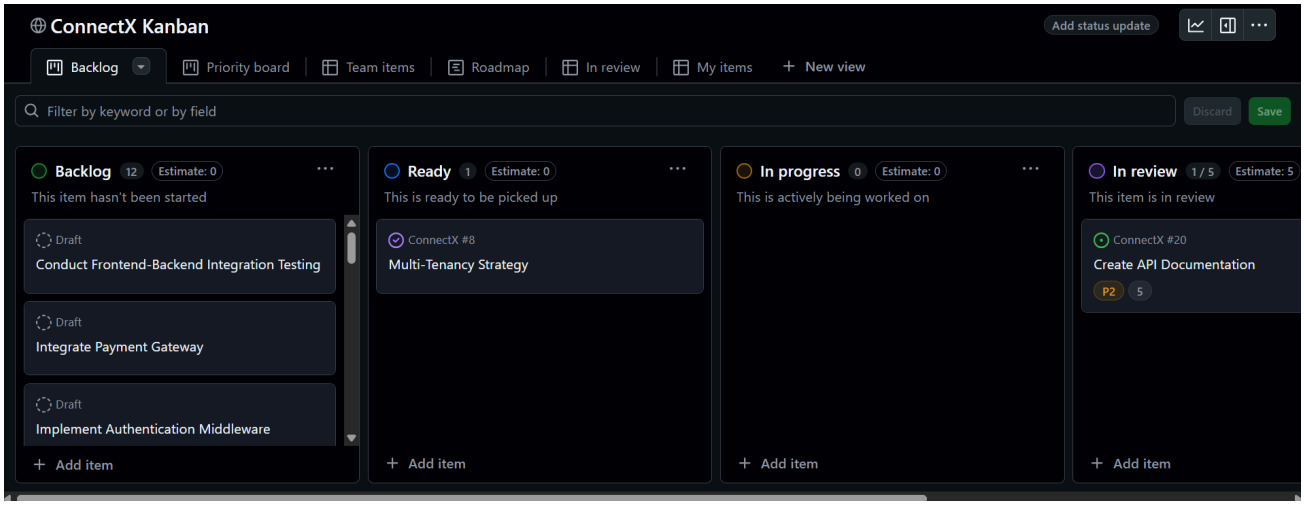


Figure 1. Kanban as our project management and collaboration tool

1.4.2.2 System Development Tools

We plan to use different tools throughout the project. The following is a list of tools that will be used during the project lifecycle. For frontend and backend development, we will use the following software development tools, as can be shown in the table below

Table 2. Basic System Development Tools

No.	System Development Tools	Type	Purpose
1	Next.js	Front-end Framework	Modern React Framework for building full-stack web application
2	Typescript	Programming Language	Type-safe JavaScript for better development experience
3	TailwindCSS	Front-end CSS Framework	Utility-first CSS framework for rapid UI development
4	Flutter	Mobile App Framework	Cross-platform UI toolkit for building natively compiled mobile applications from a single codebase
5	PostgreSQL	Database	Relational Database for structured database
6	Phyton	Programming Language	Backend programming language
7	Django	Backend Framework	High-level python web framework
8	Django Rest Framework (DRF)	API Framework	powerful toolkit for building Web APIs
9	Visual Studio Code	Code Editor/IDE	Support for debugging and version control
10	JWT	Authentication	Secure user authentication and authorization

Table 3. Other Helper Tools

No.	Helper Tool	Purpose

1	Git	Version Control System
2	GitHub	Collaboration Platform
3	pip	Python Package Manager
4	npm	Node Package Manager
5	Postman	API Testing
6	Windows	Operating System for Development
9	python-dotenv	Environment Variable Management
10	drf-yasg	API Documentation Generator

1.5 Plan of Activities

This section outlines the plan of activities undertaken to complete the **ConnectX** as our senior research project. The project was executed using an Agile methodology while ensuring that major phases of traditional Software Development Life Cycle (SDLC) were followed to meet academic requirements and ensure effective time and resource management. The activities are grouped into phases based on SDLC principles, with each phase containing detailed tasks, deliverables, and clearly defined outcomes.

1.5.1 Initiation and Requirement Analysis Phase

This phase focused on laying the foundation for the ConnectX system. Since the platform is aimed at solving real-world problems faced by students, entrepreneurs, and small-scale merchants who want to start an online store, it was crucial to gather input that would help shape practical, useful features.

Task 1: Requirement Gathering

We began by conducting informal interviews and surveys with our peers, instructors, and local business operators to understand their pain points around launching e-commerce platforms.

- **Deliverables:** Requirements Document with prioritized features and user stories

- **Tools Used:** Google Docs

Task 2: Define Project Scope & Objectives

Using the data gathered, we clearly defined the scope of ConnectX. The goal was to build a centralized backend platform for merchants to launch and manage stores without needing deep coding skills. We outlined the system boundary, two main user roles (admin, merchants—verified and unverified), and the shared storefront for customers.

- **Deliverables:** Project Scope Statement, User Role Matrix
- **Outcome:** Agreed project boundary that aligns with the available time and team size

Task 3: Competitor and Technical Research

To refine our feature set, we conducted a comparative study of platforms like Shopify, Dukaan, and snowflake. We analyzed their onboarding process, merchant tools, and admin control systems. This helped us validate the need for features like public/private product listings and profit-sharing for unverified merchants.

- **Deliverables:** Comparative Feature Matrix
- **Outcome:** Decision to implement a dual-merchant model and centralized product control

Task 4: Setup Tools and Team Structure

We divided roles among team members (frontend, backend, database, mobile). The repository was created in GitHub and Kanban board was used to track sprint tasks. Figma was used for design work, while Postman helped us test APIs during early development.

- **Deliverables:** GitHub Project Setup, Figma Design Board
- **Tools Used:** GitHub, Kanban board, Figma, Postman
- **Outcome:** Team prepared and aligned for iterative development

Milestone 1: Requirements, Scope, and Research Finalized

1.5.2 Planning Phase

Once our scope was defined, we moved on to planning. Since we were using Agile, we broke the project into sprints with a typical duration of 2–4 weeks depending on task complexity. However, to align with university expectations, we also created a waterfall-style activity timeline.

Task 5: Sprint Planning and Timeline Setup
We outlined 13 sprints between November and May. Each sprint focused on key deliverables such as database design, API implementation, storefront development, and admin dashboard features. Weekly review sessions with our advisor ensured accountability and guidance.

- **Deliverables: Sprint Roadmap, Timeline Gantt Chart**
- **Outcome:** Balanced plan combining Agile delivery with fixed academic expectations

Task 6: Risk Assessment and Contingency Planning
We identified potential risks such as API bugs, mobile app delays, and payment integration issues. Mitigation strategies were added (e.g., fallback testing plan, using sandbox environments).

- **Deliverables:** Risk Log with Owners and Contingencies
- **Tools Used:** Google Sheets
- **Outcome:** Risks accounted for early, ensuring smoother development later

Milestone 2: Full Sprint Plan and Risk Strategy Approved

1.5.3 System Design Phase

This phase focused on designing all technical and user-facing components before actual development started.

Task 7: Software Architecture Design
We used a modular approach: backend services (APIs), database, web frontend, mobile frontend, and role-based admin/merchant portal. All components connect to a shared centralized backend.

- **Deliverables:** System Architecture Diagram
- **Outcome:** Scalable, clean architecture for long-term use

Task 8: Database Schema Design
We modeled entities like products, merchants, orders, returns, stock, transactions, and users. Relations were established using foreign keys and normalization techniques were applied for performance.

- **Deliverables:** ER Diagram, Table Descriptions
- **Tools Used:** plantuml
- **Outcome:** Database ready for multi-tenant e-commerce structure

Task 9: UI/UX Prototyping
We designed low-fidelity wireframes for the admin panel, merchant dashboard, customer storefront (web and mobile). These helped us test flows like product listing, order checkout, and login/signup.

- **Deliverables:** Wireframes and UI Kit
- **Tools Used:** Figma
- **Outcome:** Approved mockups used during frontend sprints

Task 10: Security and Access Planning
We outlined JWT-based authentication, middleware for route protection, and user-role segregation (admin vs. merchant). This planning helped avoid security issues during API development.

- **Deliverables:** Security Flowchart, Auth Plan
- **Outcome:** API routes and role access clearly defined

Milestone 3: Design and Architecture Validated

1.5.4 Development Phase

The development phase was divided into 13 sprints, spanning from **November 15 to May 9**. Each sprint was 2–4 weeks long, depending on complexity.

Table 5: Sprint Planning

Sprint	Main Focus	Detailed Activities	Key Deliverables

Sprint 1	Centralized DB Schema	Design relational schema for users, products, inventory, orders, returns, and transactions	ERD, Migration Scripts
Sprint 2	Core API Implementation	Develop API routes for CRUD operations (users, products, orders, stock, returns)	RESTful APIs
Sprint 3	API Documentation	Document APIs using Swagger and test using Postman collections	Swagger UI, Postman Tests
Sprint 4	Auth System & Middleware	JWT login/signup, middleware for admin, merchant (verified/unverified)	Auth Module
Sprint 5	DB & API Refinement	Update models and endpoints based on test feedback	Improved Models, Updated Docs
Sprint 6	Storefront Web UI	Build Next.js frontend for customer product browsing, cart, and auth	Web UI with API Integration
Sprint 7	Storefront Mobile App	Develop Flutter app with same features as web	Mobile App UI (Flutter)
Sprint 8	Merchant Portal (Unverified)	Register merchants, link public products, customize storefront	Unverified Merchant Portal
Sprint 9	Admin Dashboard & Verification	Admin views merchant requests, approves/rejects verification	Admin Panel Features
Sprint 10	Verified Product Listing System	Enable verified merchants to list, edit, manage products	Product Dashboard
Sprint 11	Order & Payment Flow	Build cart, shipping, order summary, and integrate payment gateway	Checkout Flow

Sprint 12	Analytics & Admin Reports	Add charts for sales, revenue, merchant activity	Reports Module
Sprint 13	Testing & Deployment	Final testing, UI/UX polish, deploy to Vercel + Render	Final Build, Report, Slides

Milestone 4: Functional MVP Completed and Hosted

1.5.5 Testing & Optimization Phase

After all core features were developed, we conducted thorough testing and performance reviews.

Task 11: Unit and Integration Testing
We used manual testing along with Postman scripts and in-app navigation testing to ensure that all modules worked individually and when integrated, , jest for unit testing of frontend.

Task 12: Security Review
Authentication routes were tested for vulnerabilities. RBAC middleware was checked for bypass risks.

Task 13: Performance Optimization
We optimized DB queries, added lazy loading in frontend, and reduced bundle sizes.

- **Deliverables:** Test Cases, Fix Logs, Optimization Summary
- **Outcome:** Stable and secure application ready for deployment

Milestone 5: System Tested and Validated

1.5.6 Deployment Phase

Task 14: Prepare Deployment Environment
Configured Render (backend), Neon(postgresql db), Vercel (frontend), and mobile build system. Environment variables and DB access were secured.

- **Deliverable:** Deployment environment setup
- **Resources:** Deployment plan, credentials

Task 15: Deploy System Components
All components were deployed to their respective environments. Mobile APK was compiled for demo use.

- **Deliverable:** Deployed system

- **Resources:** GitHub, build tools

Task 16: Monitor and Fix Deployment Issues
Monitored system after deployment and addressed issues such as broken routes, loading delays, and mobile screen bugs.

- **Deliverable:** Fix logs, status reports
- **Resources:** Dev tools, Render logs

Task 17: User Acceptance Testing (UAT)
Selected users (classmates, advisor) tested the system. Their feedback was noted and used for final polish.

- **Deliverable:** UAT Summary
- **Resources:** Test checklist, observation logs

Task 18: Create User Manual
Wrote simple how-to guides for merchants and customers explaining key tasks like login, product listing, and order tracking.

- **Deliverable:** User Manual PDF
- **Resources:** UI screenshots, feature descriptions

Task 19: Developer Documentation
All technical files, API structure, and deployment instructions were documented for future maintainers.

- **Deliverable:** Tech Docs Archive
- **Resources:** Codebase, system diagrams

Milestone 6: Final Deployment & Documentation Complete

1.5.7 Maintenance Phase

Task 20: Monitor System and Gather Feedback
After deployment, we continuously monitored logs, response times, and user behavior.

- **Deliverable:** Monitoring Sheet
- **Resources:** Console logs, user feedback

Task 21: Bug Fixes and UI Polishing
We addressed UI bugs, fixed minor backend issues, and made improvements based on UAT.

- **Deliverable:** Patch Logs
- **Resources:** Issue reports

Task 22: Feature Enhancements (Optional)
Planned improvements (e.g., merchant analytics breakdown, better dashboard sorting) were listed for post-graduation release.

- **Deliverable:** Feature Roadmap
- **Resources:** User suggestions, development notes

Milestone 7: Project Closure and Handover

1.5.8 Task Dependencies Table

Table 6: Task dependencies

Task No.	Task	Dependencies	Duration (Weeks)
1	Requirement Gathering	-	2
2	Define Project Scope	1	1
3	Research Existing Platforms	1	2
4	Develop Sprint Plan	2,3	1

5	Risk & Contingency Planning	4	1
6	Timeline and Milestone Setup	5	1
7	Design System Architecture	2,4	2
8	Design Database Schema	7	1
9	UI/UX Prototyping	7	2
10	Security Design	7,8	2
11	Begin Development Sprints	7,8,9,10	20
12	Perform Testing	11	2
13	Final Deployment	12	1
14	Conduct UAT	13	1
15	Write Manuals & Documentation	14	1

1.6. Budget Required

The project will require a budget to cover the cost of resources, including hardware, software, and facilities. The budget will be required to cover the costs of software deployment, and database management. Some costs include server fees, domain name, deployment related expenses.

1.7. Significance of the Study

The development of ConnectX holds significant value in addressing technical and financial barriers faced by aspiring e-commerce entrepreneurs in Ethiopia and beyond. By offering a centralized, scalable, and developer-friendly backend platform, ConnectX enables a wide range of users regardless of technical skill level or capital to participate in the digital commerce ecosystem. The platform's emphasis on collaboration, affordability, and flexibility supports sustainable business

development while promoting innovation and inclusivity in the market.

For Student Entrepreneurs and Early-Stage Startups

- Eliminates the need for upfront capital investment in inventory or backend infrastructure.
- Provides tools to build functional online stores without deep technical expertise.
- Enables faster go-to-market through ready-made frontend templates and pre-built APIs.
- Offers commission-based pricing, reducing financial risk during early growth stages.

For Small and Medium Businesses (SMBs)

- Supports business expansion through scalable backend services and customizable APIs.
- Facilitates public or private product listing for flexible inventory and revenue models.
- Encourages brand development by offloading technical responsibilities to a stable backend system.

For Developers and Technical Teams

- Offers developer-friendly, well-documented APIs for seamless frontend integration.
- Supports modular e-commerce development using reusable backend components.
- Enables collaboration through shared product ecosystems and team management features.

For the Local Tech and Business Ecosystem

- Fosters innovation by lowering barriers to e-commerce platform development.
- Encourages collaboration and resource sharing among merchants, developers, and startups.
- Promotes digital entrepreneurship as a pathway to economic participation and growth.

Chapter Two: Literature Review

The evolution of e-commerce backend systems has been significantly influenced by various computational advancements and theoretical foundations. This chapter provides an overview of existing literature to establish the context of the current study.

2.1 Study-Related Works

The exponential growth of digital commerce has created an urgent need for backend systems that are not only scalable and secure, but also modular, cost-efficient, and accessible to both developers and business users. To address this, modern e-commerce development has undergone a paradigm shift driven by innovations in backend architectures, API-first strategies, and developer tooling [1][6]. These advancements have laid the foundation for new platforms like ConnectX that aim to reduce complexity and democratize access to online business infrastructure. One of the most significant advancements in backend design is the adoption of microservices architecture. This approach breaks down a monolithic application into smaller, loosely coupled services that can be independently developed, deployed, and scaled [1]. Smith, Brown, and Davis (2021) argue that microservices have become critical for improving scalability

and maintainability in e-commerce systems, enabling organizations to respond quickly to changes in traffic, market conditions, and business needs [1]. Each microservice focuses on a specific domain such as product management, payment processing, or user authentication allowing greater flexibility and fault isolation. Complementing microservices is the emergence of serverless computing, which allows developers to execute code in response to events without provisioning or managing servers. As Zhao, Li, and Wang (2022) note, serverless infrastructure enables automatic scaling and cost optimization by charging only for the compute time used, making it ideal for startups and platforms with variable usage patterns [2]. Developers can thus focus entirely on business logic and application design without worrying about operational overhead.

Despite the rise of newer paradigms, traditional web frameworks like Django remain widely used due to their maturity, security, and ease of use. Green (2022) highlights Django's integrated components such as its ORM, admin interface, and built-in authentication as major advantages for developers building secure, data-driven applications [3]. While Django follows a monolithic model by default, it can be adapted into a modular monolith or restructured using Clean Architecture to promote separation of concerns and future extensibility.

To support scalable deployments, the backend infrastructure of many modern systems also incorporates containerization technologies like Docker and orchestration tools like Kubernetes. Nguyen, Lopez, and Chen (2022) demonstrate how containerized systems offer consistent development environments, isolate dependencies, and improve resource utilization, making them especially useful for multi-tenant architectures serving diverse customer bases [4]. These tools also enhance DevOps practices by enabling continuous integration and continuous delivery (CI/CD).

Security and privacy are central concerns for shared infrastructure platforms. Li, Zhang, and Xu (2021) emphasize the need for privacy-preserving mechanisms, including data encryption, access control layers, and secure multi-tenancy through schema-level or row-level isolation [5]. These features are essential to protect both business data and end-user information in platforms offering centralized backend services to multiple vendors.

Another key trend in backend development is the shift toward API-driven architecture. According to Knowl.io (2024), decoupling frontend and backend through well-defined RESTful or GraphQL APIs allows developers to build modular applications that are easier to maintain and scale [6]. APIs also enable cross-platform integration, allowing businesses to offer consistent services across web, mobile, and third-party platforms without rewriting core logic. This is especially relevant for ConnectX, which serves a diverse set of users ranging from developers building custom apps to merchants using no-code templates.

Financial considerations remain a significant constraint, particularly for student entrepreneurs and early-stage startups. Entrepreneur Magazine (2023) reports that backend development for an e-commerce platform can cost from $2,000 for small projects to over $100,000 for

enterprise-grade implementations [7]. These costs often deter innovation at grassroots levels, reinforcing the need for platforms like ConnectX that offer shared infrastructure and a performance-based pricing model, where users pay based on actual sales rather than upfront commitments.

Several real-world platforms have emerged with similar objectives to ConnectX. For instance, Medusa.js is an open-source headless commerce engine that provides modular APIs for order management, product catalogs, and customer profiles [12]. Its architecture allows developers to easily extend core services and integrate with custom frontends. Similarly, Vendure offers a GraphQL-based backend tailored for mid-market and niche commerce platforms. It provides extensible plugin support and a clean separation between backend services and presentation layers [13]. Another example is Commerce Layer, a commercial SaaS platform that enables headless commerce through globally scalable APIs, particularly for brands looking to integrate e-commerce into existing CMS or ERP systems [14].

Additionally, the influence of platforms like Stripe and Snowflake cannot be overstated. Stripe has become the gold standard for API-first payment processing, offering secure, developer-friendly integration methods that reduce implementation complexity [9]. Snowflake, on the other hand, demonstrates the power of cloud-native, multi-tenant architectures in managing large-scale data infrastructure while ensuring tenant-level security and access control [10]. These platforms have set the benchmark for how centralized services can serve a wide range of clients with minimal configuration.

Various studies and industry practices emphasize that developer and merchant experience is just as important as system functionality. Platforms must cater to both technical and non-technical users by offering intuitive management interfaces, clear onboarding flows, and operational visibility through dashboards. A critical factor in this experience is the availability of well-documented APIs, which help reduce friction in adoption, minimize integration errors, and empower third-party developers to build confidently. Comprehensive documentation including endpoint descriptions, example requests and responses, error handling, and version control enables faster onboarding and smoother integration across different environments.

Ease of integration is further enhanced by modular API design, token-based security, and support for common frontend technologies. Moreover, dashboards offering analytics, order tracking, and team management capabilities empower merchants to operate efficiently without requiring deep technical knowledge. Tools like Django REST Framework (DRF) facilitate this developer-first approach by providing automatic API scaffolding, authentication workflows, and advanced data manipulation features out of the box.

Altogether, the convergence of these trends modular architecture, scalable infrastructure, privacy and security, API-first development, documentation, and user-centric design has created fertile

ground for solutions like ConnectX. By leveraging these technologies and architectural practices, ConnectX aims to offer a robust, multi-tenant backend platform that empowers developers, students, and entrepreneurs to build and manage scalable e-commerce platforms with minimal overhead.

2.2 Identifying Milestones of the Related Literature and Finding the Gaps

2.2.1 Milestones

The evolution of e-commerce backend systems has followed a trajectory marked by advancements in architecture, scalability, security, and developer enablement. The literature outlines several key milestones that have shaped modern e-commerce infrastructure and informed the development of platforms like ConnectX:

- ➔ **Early E-Commerce Architectures (2005–2012):-** Initial online retail systems primarily adopted monolithic architectures focused on basic CRUD operations for products, orders, and users. These systems had limited scalability and minimal integration flexibility. During this period, backend solutions were tightly coupled to frontend interfaces, making them difficult to maintain or scale.

- ➔ **Rise of MVC Frameworks and REST APIs (2013–2016):-** Frameworks like Django and Ruby on Rails gained popularity for their MVC-based architecture and rapid development capabilities. RESTful APIs became the standard for enabling frontend-backend separation, allowing developers to build more flexible and responsive interfaces. These changes laid the foundation for multi-platform e-commerce development and facilitated the decoupling of business logic from presentation layers.

- ➔ **Microservices and Containerization (2016–2020):-** This phase saw the widespread adoption of microservices architecture to address the limitations of monolithic systems. Researchers like Smith et al. (2021) highlighted that microservices enabled independent scaling, improved fault tolerance, and reduced deployment friction [1]. Tools like Docker and Kubernetes became essential for managing services in isolation and deploying scalable backend systems. Nguyen et al. (2022) emphasized how container orchestration enabled high availability, resource efficiency, and CI/CD pipelines in complex e-commerce platforms [4].

- ➔ **API-First and Headless E-Commerce (2020–2023):-** A growing number of systems embraced headless architecture with a strong focus on developer experience. Platforms such as Medusa.js and Vendure adopted API-first approaches to provide reusable backend services and customization for niche e-commerce platforms [12][13]. Knowl.io (2024) also emphasized the architectural benefits of API-driven development, which include parallel frontend-backend development, better modularity, and reduced coupling [6].

- ➔ **Focus on Data Privacy and Regulatory Compliance (Ongoing):-** With the enforcement of

GDPR, CCPA, and other global data privacy regulations, backend systems evolved to integrate encryption, access controls, and data isolation mechanisms. Li et al. (2021) discussed advancements in privacy-preserving techniques, including homomorphic encryption and role-based access control [5]. Django, with its ORM-level query protection and secure session management, has been cited as a practical framework for achieving data security and compliance.

- ➔ **Developer Experience and Documentation-Centered Development (2022–present):-** Recent research and platform evolution (e.g., Stripe, Medusa.js, and Django REST Framework) stress the significance of clean documentation, self-service APIs, and merchant dashboards in adoption rates and user satisfaction. Tools that simplify backend integration such as DRF's auto-generated API views, token-based authentication, and pagination contribute significantly to developer productivity and are now considered core design considerations.

2.2.2 Identified Gaps

While significant advancements have been made in the development of backend systems for e-commerce platforms, a review of the literature reveals several key areas that remain underexplored and present opportunities for further research and system innovation. One such area is the limited integration of real-time processing capabilities in multi-tenant e-commerce environments. Although microservices and containerization technologies have improved scalability and deployment efficiency, many systems still rely heavily on batch processing for updates related to inventory, order status, and customer interactions. The lack of real-time data flow introduces latency and may compromise user experience during high-traffic or high-frequency transactional periods.

Another notable gap in the literature concerns the concept of shared or collaborative product ecosystems. Most studies and commercial implementations focus on isolated store models, where each merchant manages a distinct product catalog and backend instance. This structure limits accessibility for entrepreneurs who do not have the resources to source or manage their own inventory. There is a need for more research into systems that enable shared product listings, where products can be made publicly available for resale across multiple storefronts, enhancing efficiency and inclusivity within the ecosystem.

Additionally, while API-first architectures are now widely adopted, the importance of developer-facing elements, particularly comprehensive and standardized API documentation, is often underemphasized in the academic literature. Poor or inconsistent documentation continues to be a barrier for developers, especially those working independently or without formal training. There is a growing recognition that strong documentation, developer support tools, and clear integration workflows are crucial to maximizing the usability and scalability of backend systems.

In the context of multi-tenant systems, the literature supports the theoretical value of tenant isolation for data security and regulatory compliance. However, there is a lack of practical guidance or case studies on how this isolation is implemented particularly within commonly used frameworks such as Django. Questions remain regarding best practices for schema-based versus row-level isolation, especially in environments where scalability, security, and maintainability must be balanced.

Although cost is a frequently cited barrier to entry in e-commerce development, pricing models for backend infrastructure have received limited academic attention. Most available platforms adopt traditional subscription or usage-based models, which may not align with the needs of early-stage ventures or entrepreneurs operating with minimal capital. There is a noticeable absence of research exploring alternative pricing strategies, such as performance-based or revenue-share models, which could enable more equitable access to backend services.

These gaps suggest that future research could focus not only on technical scalability and security, but also on collaborative design, inclusive architecture, documentation quality, and economic accessibility factors that are increasingly important in supporting a broader spectrum of e-commerce participants.

2.2.3 Important Insights

The body of literature on e-commerce backend systems often concentrates on specific technologies such as microservices, APIs, and containerization but rarely presents a holistic view of how these components can work together to support scalable, accessible platforms. Much of the research remains compartmentalized, focusing on optimizing isolated elements rather than examining how full backend ecosystems function when implemented in real-world, multi-tenant environments. This leaves a gap in understanding the challenges and design considerations involved in delivering backend infrastructure as a service for diverse user groups.

There's also a noticeable lack of attention to the developer and business-user experience. While performance and scalability receive considerable focus, aspects like ease of onboarding, documentation quality, and interface usability tend to be underemphasized. These are critical factors for independent developers, students, and small merchants who often operate without dedicated technical teams. Without intuitive tools and clear documentation, even well-architected systems can become difficult to adopt or maintain.

The literature gives limited attention to the broader social and economic context in which backend systems are used. Most studies take a technical perspective, without fully addressing how infrastructure constraints, financial limitations, or access disparities affect small businesses or entrepreneurs in resource-limited regions. These real-world constraints directly influence adoption but are rarely explored in depth. There's room for more research that bridges technology with accessibility and economic inclusion.

Another point often overlooked is the operational side of deployment. While many papers describe ideal system designs or highlight successful case studies, few discuss the day-to-day implementation challenges teams face—such as integration with existing platforms, maintaining performance under scale, or evolving the system as business needs change. This lack of practical deployment insight makes it difficult for new projects to anticipate and plan for long-term sustainability.

There's growing curiosity around the role of automation, low-code development, and even decentralized technologies in backend system design, especially as these tools become more accessible. However, in most current literature, these concepts are still emerging, with limited real-world data to evaluate their effectiveness. Exploring how these innovations could simplify

system maintenance or reduce technical barriers for new businesses is an area that could benefit from deeper research.

These insights suggest that future work on backend systems for e-commerce should not only push for technical innovation but also focus on real-world usability, inclusivity, and resilience. Understanding how systems perform beyond the code in the hands of everyday users is essential to shaping backend infrastructure that truly supports digital entrepreneurship at every level.

2.3 Lessons Learned from the Literature

The review of recent research and systems in the domain of e-commerce backend development reveals several key lessons that inform the design and implementation of scalable, secure, and developer-friendly platforms. These insights span architectural choices, infrastructure management, security practices, and user experience considerations.

1. **Smith, Brown, & Davis (2021):-** ***Modular Architecture Enables Agile Scaling and Maintenance***

The adoption of microservices architecture allows different components such as product management, user authentication, and order processing to function independently. This structure enables horizontal scaling, fault isolation, and easier updates, which are essential for platforms handling dynamic e-commerce workflows and user growth [1].

2. **Zhao, Li, & Wang (2022):-** ***Serverless Architecture Reduces Cost and Operational Overhead***

Serverless computing facilitates automatic scaling based on demand and eliminates the need for manual infrastructure management. This model supports event-driven architectures and is particularly advantageous for applications with fluctuating workloads, as it minimizes operational costs and improves deployment speed [2].

3. **Green (2022):-** ***Django as a Secure, Extensible Framework***

Django's built-in components, such as its Object-Relational Mapping (ORM), admin panel, and authentication system, streamline backend development and ensure robust security. The framework's modularity also makes it adaptable for projects requiring a balance between rapid development and maintainability [3].

4. **Nguyen, Lopez, & Chen (2022):-** ***Containerization for Multi-Tenant Isolation and CI/CD***

Containerization technologies like Docker, along with orchestration tools like Kubernetes, enhance deployment efficiency and enforce system consistency across environments. These tools support continuous integration and delivery (CI/CD) workflows and simplify tenant isolation in multi-tenant architectures [4].

5. **Li, Zhang, & Xu (2021):-** ***Security and Data Privacy as Foundational Principles***

Ensuring secure access through mechanisms such as token-based authentication, role-based

authorization, and data encryption is critical. Multi-tenant systems benefit from row-level or schema-level isolation to maintain data confidentiality and regulatory compliance in shared environments [5].

6. **Knowl.io (2024) — *API-First Design Enhances Flexibility and Developer Adoption***

API-driven systems promote decoupling between frontend and backend, enabling seamless integration across different platforms and devices. This architecture supports flexible frontend customization and simplifies collaboration between teams, improving development velocity and platform extensibility [6].

7. **Entrepreneur Magazine (2023):- *Reducing Barriers Through Shared Infrastructure and Tiered Pricing***

Traditional backend development can be prohibitively expensive, particularly for early-stage businesses or student entrepreneurs. Shared infrastructure platforms and performance-based pricing models have been identified as effective strategies to reduce upfront costs and encourage wider adoption [7].

8. **Medusa.js, Vendure, & Commerce Layer:- *Modular, Headless E-Commerce as a Trend***

Modern headless platforms emphasize modularity and reusability through pluggable APIs and extensible services. These systems support tailored commerce experiences and allow developers to build custom storefronts while reusing core backend functionalities [12][13][14].

9. **Stripe & Snowflake:- *Developer-Centric APIs and Secure Multi-Tenant Design***

Stripe demonstrates the value of well-documented, developer-first APIs that accelerate integration and reduce development overhead. Snowflake, in parallel, exemplifies scalable and secure multi-tenant architectures capable of serving diverse customer bases with centralized management and tenant-level isolation [9][10].

Chapter Three: Problem Analysis and Modeling

3.1 Existing System and Its Problems

Today's e-commerce backend systems handle important tasks like order processing, user accounts, product management, and payment integration. These systems are well-developed and used by many businesses, but they are often built for companies that already have enough money, staff, and technical skills. This makes it hard for small businesses, students, or new entrepreneurs to use them easily or affordably. Many platforms either offer full packages with limited flexibility or tools that require strong backend knowledge to set up and maintain. For example, services like Shopify make it easy to launch a store, but they come with fixed features and pricing that can be expensive over time. On the other hand, platforms like WooCommerce or Magento give more control but need experience with servers, coding, databases, and system security. For someone just getting started

especially without a technical background this can be overwhelming.

Building a custom backend system is also expensive. Even basic setups cost thousands of dollars, and ongoing costs like hosting, updates, and security add up quickly. This stops many people from turning their ideas into working businesses. Without affordable, ready-to-use backend tools, a lot of great ideas never make it to market. As businesses grow, they also need backend systems that can grow with them. But most systems don't offer pricing models that match how a small business earns money. Instead of paying based on success or usage, businesses are often charged a fixed amount every month even if they have no sales. For startups or seasonal businesses, this can lead to spending more than they can afford. Another issue is that current platforms are mostly built for one user or store at a time. They don't make it easy for people to share resources or work together. For example, someone with marketing skills but no products to sell can't easily connect with suppliers or use shared product listings. Developers have to build their own backend features from scratch, and businesses can't easily plug into a shared system. This makes the whole process slower, more expensive, and more complicated than it needs to be.

In recent years, a few open-source and commercial projects have emerged to address some of these issues by offering more flexible and modular architectures. Platforms like Medusa.js, Vendure, and Commerce Layer represent a shift toward headless and API-first e-commerce systems. These platforms focus on providing reusable backend services through well-structured APIs, allowing developers to build custom storefronts for different devices and audiences without rebuilding core logic. Their modular architecture also supports plugin-based extensions, which improves maintainability and enables easy integration with external services. While these platforms have made backend development more accessible for developers, they are still largely targeted at technically skilled users who can manage infrastructure and code-level customization. In addition, most of these systems still lack built-in collaboration models or shared product listing features that would support resource-sharing among less-resourced users. For many small teams or entrepreneurs without strong technical backgrounds, setting up, customizing, and scaling these systems remains a challenge, both financially and technically. While current e-commerce backends are powerful, they often don't meet the needs of small teams, independent developers, or people starting with little money or experience. The tools that exist work well for bigger companies or experienced developers, but there's a clear need for simpler, more flexible backend systems that support smaller players who want to build and grow online businesses without huge upfront costs or deep technical skills.

3.2 Requirements of the Proposed Solution

To solve the challenges identified in current e-commerce backend systems such as high development costs, lack of technical accessibility, limited scalability, and the absence of collaborative infrastructure the proposed solution, ConnectX, implemented a set of well-defined functional and non-functional requirements. These requirements are designed to support a flexible, developer-friendly, and multi-tenant backend that enables users to launch and manage e-commerce platforms efficiently, even with limited resources.

The solution we provide support both technical users (e.g., developers building custom storefronts) and non-technical users (e.g., small merchants using provided templates), while also ensuring scalability, security, and ease of use. By combining customizable APIs, shared product listing systems, and plug-and-play tools, ConnectX aims to lower the barrier to entry for digital entrepreneurship and offer a collaborative environment for growing online businesses.

3.3 System Modeling

3.3.1 Functional Requirements

The functional requirements outline the essential features and capabilities that the system must provide to meet the needs of its users. These requirements are based on the project goals and the gaps identified in existing systems.

1. Multi-Tenancy Support
    - Host multiple e-commerce businesses under one backend system.
    - Ensure strict data separation for each tenant using row-level or schema-level isolation.
2. User Registration and Authentication
    - Allow users to register, log in, and recover passwords securely.
    - Implement token-based authentication and role-based access control.
3. Team Management
    - Enable store owners to add team members with specific roles.
    - Allow shared access to dashboards, product management, and order handling.
4. Developer API Access
    - Provide RESTful APIs for operations like product listing, order creation, and account handling.
    - Secure API access with keys and authentication tokens.
5. API Key Management
    - Allow users to generate, view, and revoke API keys.
    - Enforce permissions and access scopes for each key.
6. Product Management System
    - Support adding, editing, and deleting products.
    - Include attributes such as name, price, description, images, categories, and status.

7. Dual Product Listing System (Public/Private)
    - Let merchants list products privately (for personal use) or publicly (for others to sell).
    - Support commission settings for public product contributors.
8. Order Management
    - Handle order placement, updates, cancellation, and fulfillment.
    - Allow both merchants and customers to track order status.
9. Payment Gateway Integration
    - Integrate with gateways like Chapa for secure online payments.
    - Support real-time payment status updates and transaction logging.
10. Merchant Verification System
    - Enable merchants to submit business details for verification.
    - Give admins tools to review, approve, or reject applications.
11. Dashboard for Merchants
    - Provide an interface to manage products, orders, earnings, and team settings.
    - Display basic business analytics and API management options.
12. Admin Panel
    - Allow platform admins to manage users, oversee business verifications, and monitor activity.
    - Include tools for banning, flagging, or assisting businesses when needed.
13. Analytics and Sales Reporting
    - Generate reports for sales performance, order trends, and top-selling products.
    - Present data through charts or tables in the merchant dashboard.
14. Frontend Template Provision
    - Offer ready-to-use web and mobile storefront templates.
    - Allow template customization and integration with backend via API.
15. Custom Frontend Integration
    - Enable developers to connect their own frontend apps using API and API keys.
    - Support flexible design and branding options.

16. Commission Configuration for Public Listings
    - Let merchants define earnings percentage for products they list publicly.
    - Automatically apply this during order processing and revenue distribution.
17. Business Settings Management
    - Allow businesses to update store name, contact info, logos, and branding.
    - Support customization of basic store policies and preferences.
18. Secure Media Uploads
    - Enable secure image and file uploads for product listings.
    - Validate file types, sizes, and storage access rights.
19. Product Search and Filtering
    - Provide tools to search and filter products by keyword, category, price, and type.
    - Improve discoverability in both merchant and customer dashboards.

3.3.2 Non-Functional Requirements

1. **Scalability**
    - Supports multi-tenancy with clean architecture.
    - Utilizes PostgreSQL with schema or row-level data isolation.
    - Implements containerization (Docker) and optimized APIs (pagination, filtering).

2. **Security**
    - Applies token-based authentication and role-based access control.
    - Enforces HTTPS, input validation, and Django's built-in protections.
    - Secures API usage through key management and merchant verification.

3. **Reliability**
    - Built with a modular monolith for better fault isolation.
    - Includes error handling, logging, and monitoring tools.
    - Leverages Django's stable core for continuous operation.

4. **Usability**
    - Features user-friendly dashboards for merchants and admins.
    - Provides well-documented and versioned APIs with practical examples.
    - Includes API key control and team access roles for easier collaboration.

5. **Performance**

- Uses optimized queries and efficient response structures.
- Employs caching and async tasks for background processing.
- Ensures responsive performance through proper database indexing.

3.3.3 Use Case

3.3.1.1 Use Case Identification

Table 7: Use Case Identifications

Use Case ID	Use Case Name	Actor(s)	Description
UC01	Register Account	Customer,Merchant	Users sign up for the platform by providing personal details and verifying their identity.
UC02	Approve Merchant	Admin	Admin reviews and approves merchant registrations to ensure credibility.
UC03	Browse Product	Customer	Customers explore and search for products available on the platform.
UC04	Manage Products	Merchant	Merchants add, edit, or remove products from their store listings.
UC05	Place Order	Customer	Customers select products, add them to their cart, and complete the purchase.
UC06	Process Orders	Merchant	Merchants view and fulfill customer orders, updating their status.
UC07	Monitor System	Admin	Admin tracks platform performance and ensures smooth operations.

UC08	Process Payment	Customer,Payment Gateway	Customers securely pay for orders through an integrated payment system.

3.3.1.2 Use Case Mapping

- **Use Case Name: Register Account (UC01)**

Table 8: Use Case Name: Register Account (UC01)

Use Case ID	UC01
Primary Actor	Customer,Merchant
Description	Users join the ConnectX platform by creating an account with their personal details, such as name, email, and password. Customers may optionally verify their identity for added security.
Precondition	None
Postcondition	User account is created and ready for use.
Flow of Events	1. User selects the "Sign Up" option. 2. User enters personal details (e.g., name, email, password). 3. User optionally verifies identity (e.g., via email or ID). 4. System validates the information. 5. System creates the account and sends a confirmation email.
Alternate Flows	➢ If details are incomplete, the system prompts, "Please fill all required fields," and returns to step 2. ➢ If the email is already registered, the system shows, "Account already exists," and suggests logging in.

- **Use Case Name: Approve Merchant (UC02)**

Table 9: Use Case Name: Approve Merchant (UC02)

Use Case ID	UC02
Primary Actor	Admin

Description	Admin reviews merchant registration requests, checks their credentials, and approves them to sell on the platform, ensuring only trusted merchants join.
Precondition	Merchant has registered and Admin is logged in.
Postcondition	Merchant account is approved and activated.
Flow of Event	1. Admin selects the merchant approval option. 2. Admin reviews the merchant's submitted details. 3. Admin approves or rejects the registration. 4. System updates the merchant's status and notifies them.
Alternate Flows	➢ If Admin rejects the merchant, the system sends a rejection notice and stops the process. ➢ If details are incomplete, the system prompts the merchant to provide more information.

❖ **Use Case Name: Browse Products (UC03)**

Table 10: Use Case Name: Browse Products (UC03)

Use Case ID	UC03
Primary Actor	Customer
Description	Customers explore the platform, searching for products by category, keywords, or filters to find items they want to buy.
Precondition	None
Postcondition	Customer views a list of relevant products
Flow of Events	1. Customer selects the search or browse option. 2. Customer enters keywords or applies filters (e.g., category, price). 3. System displays matching products. 4. Customer views product details.
Alternate Flows	➢ If no products match, the system shows, "No results found," and suggests related items. ➢ Customers can cancel the search and return to the homepage.

❖ **Use Case Name: Manage Products (UC04)**

Table 11: Use Case Name: Manage Products (UC04)

Use Case ID	UC04
Primary Actor	Merchant
Description	Merchants update their store by adding new products, editing existing ones, or removing items no longer available.
Precondition	Merchant is logged in and approved.
Postconditions	Product catalog is updated
Flow of Events	1. Merchant selects the product management option. 2. Merchant chooses to add, edit, or remove a product. 3. Merchant enters or updates product details (e.g., name, price, description). 4. System saves the changes.
Alternate Flows	➢ If details are invalid, the system displays, "Invalid input," and returns to step 3. ➢ Merchants can save a draft and complete the product listing later.

❖ **Use Case Name: Place Order (UC05)**

Table 12: Use Case Name: Place Order (UC05)

Use Case ID	UC05
Primary Actor	Customer
Description	Customers choose products, add them to their cart, and finish the purchase process to place an order.
Preconditions	Customers are logged in and products are in the cart.
Postconditions	Order is created and delivered to the merchant.
Flow of Events	1. Customer reviews items in the cart. 2. Customer selects "Checkout." 3. Customer enters shipping and payment details.

	4. System confirms the order and notifies the merchant.
Alternate Flows	➢ If payment fails, the system shows, "Payment error," and returns to step 3. ➢ Customers can cancel the order and clear the cart.

❖ **Use Case Name: Process Orders (UC06)**

Table 13 Use Case Name: Process Order (UC06)

Use Case ID	UC06
Primary Actor	Merchant
Description	Merchants review customer orders, prepare them for shipping, and update the order status to keep customers informed.
Preconditions	Merchant is logged in and orders are placed.
Postconditions	Order status is updated.
Flow of Events	1. Merchant selects the order management choice. 2. Merchant views new orders. 3. Merchant updates the order status (e.g. shipped or delivered). 4. System notifies the customer of the update.
Alternate Flows	➢ If an order issue arises, the merchant can mark it as pending and resolve it later. ➢ Merchant can cancel an order if it cannot be fulfilled.

❖ **Use Case Name: Monitor System (UC07)**

Table 14. Use Case Name: Monitor System (UC07)

Use Case ID	UC07

Primary Actor	Admin
Description	Admin keeps an eye on the platform's performance, checking metrics like uptime and user activity to ensure everything runs smoothly.
Preconditions	Admin is logged in.
Postconditions	Admin looks at systems performance data.
Flow of Events	1. Admin selects the monitoring option. 2. System displays performance metrics (e.g., uptime, traffic). 3. Admin reviews the data.
Alternate Flows	➢ If issues are detected, Admin can trigger alerts for further investigation. ➢ Admin can exit monitoring without taking action.

❖ **Use Case Name: Process Payment (UC08)**

Table 15: Use Case Name: Process Payment (UC08)

Use Case ID	UC08
Primary Actor	Customer,Payment Gateway
Description	Customers securely pay for their orders using a trusted payment gateway integrated with the platform.
Preconditions	Customer is at the checkout and the order is set.
Postconditions	Payment is processed and the order is accepted.

Flow of Events	1. Customer selects a payment method. 2. Customer enters payment details. 3. System connects to the payment gateway. 4. Payment gateway processes the transaction. 5. System confirms the payment.
Alternate Flows	➢ If the payment fails, the system shows, "Payment declined," and returns to step 1. ➢ Customers can cancel the payment and return to the cart.

3.3.1.4 Use Case Diagram

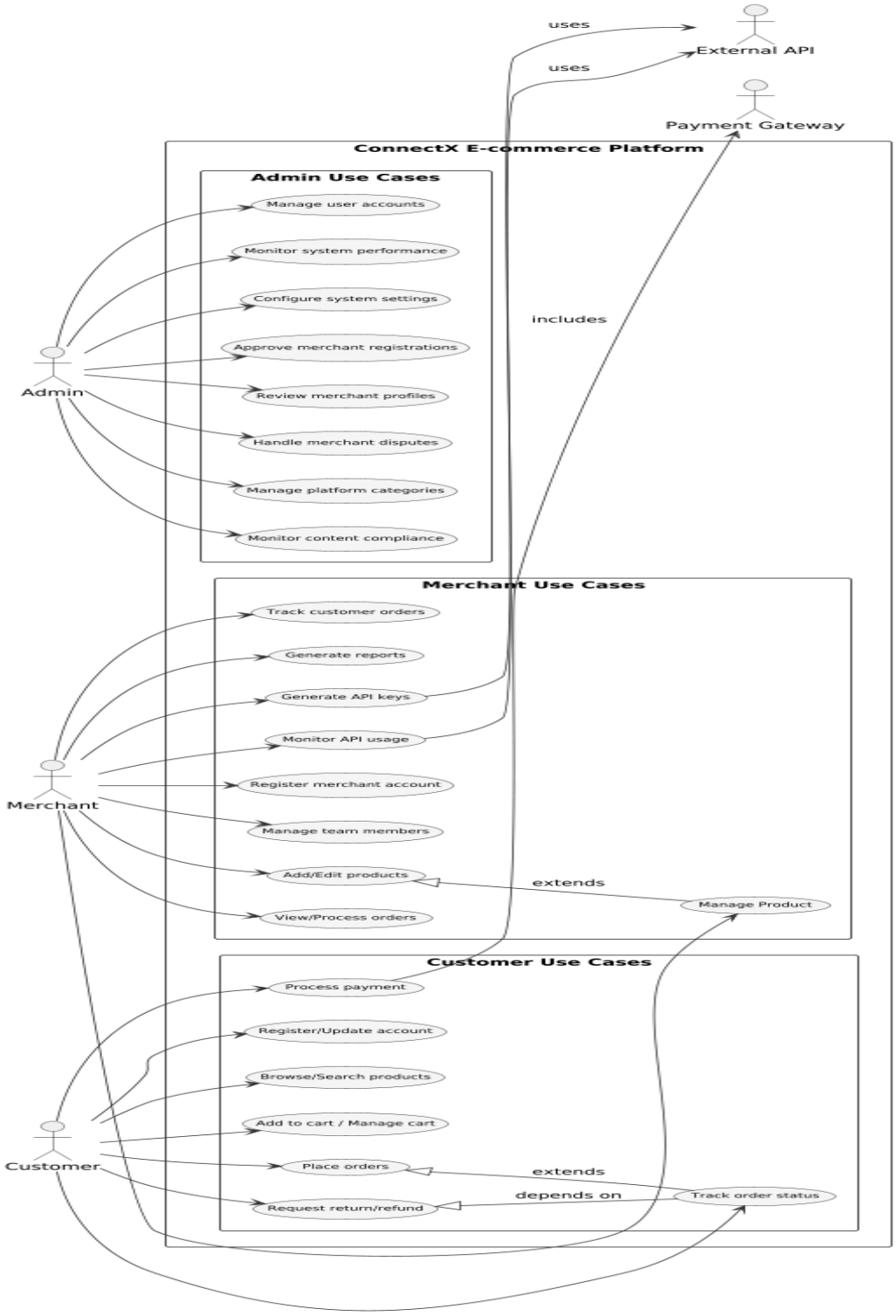


Figure 2: Use Case diagram

3.3.4 Dynamic Models of a System

3.3.4.1 Sequence Diagrams

A sequence diagram, in the context of our proposal, illustrates the interactions between objects in a system over time. It shows the sequence of messages exchanged between objects to complete a specific task or process. This helps in understanding the order of events, the flow of information, and the collaboration between objects, ensuring a clear representation of how different components interact to achieve a desired outcome.

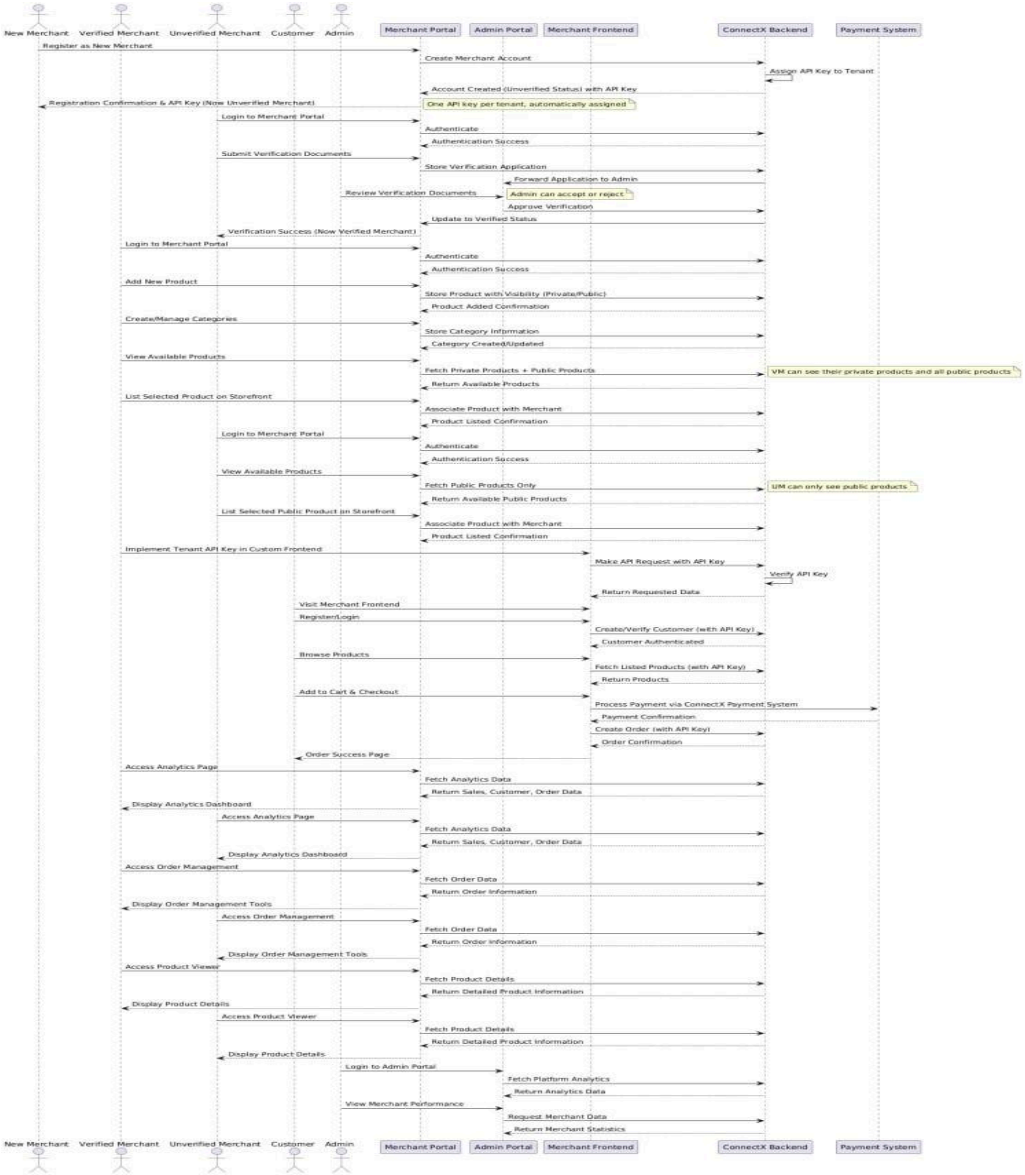


Figure 3: Sequence Diagram

3.3.4.2 State Machine Diagrams

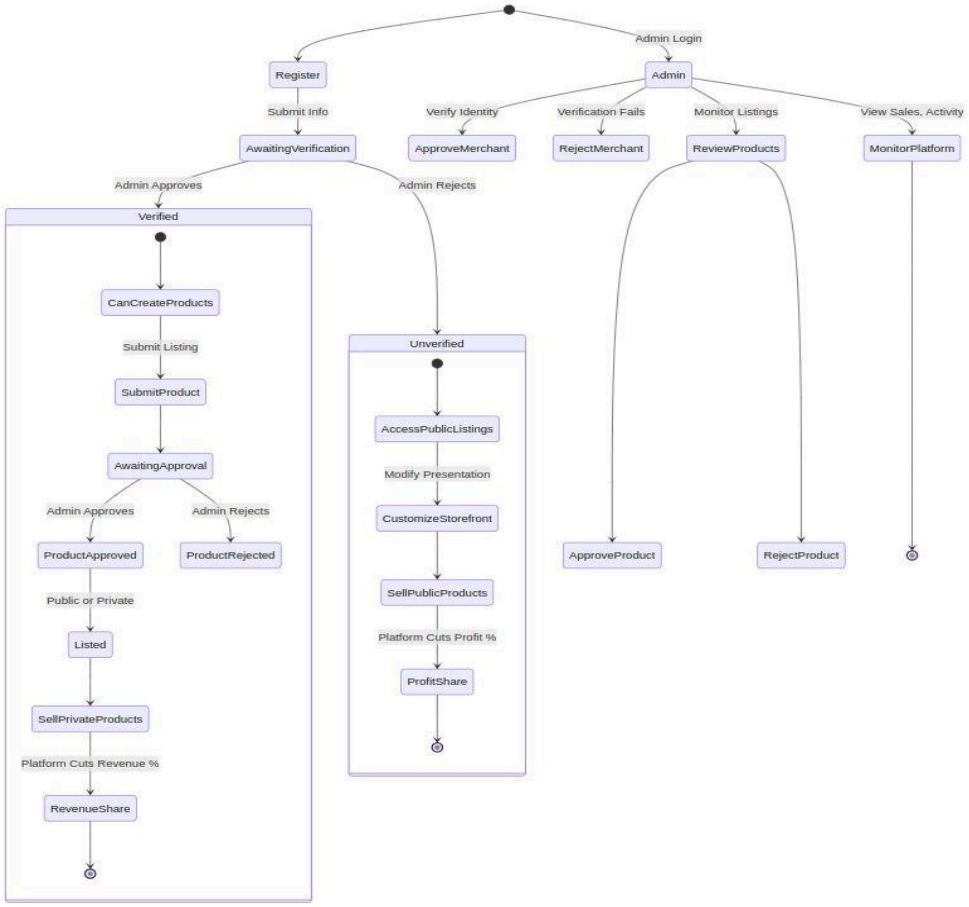


Figure 4:State Machine Diagrams

3.3.4.3 Activity Diagrams

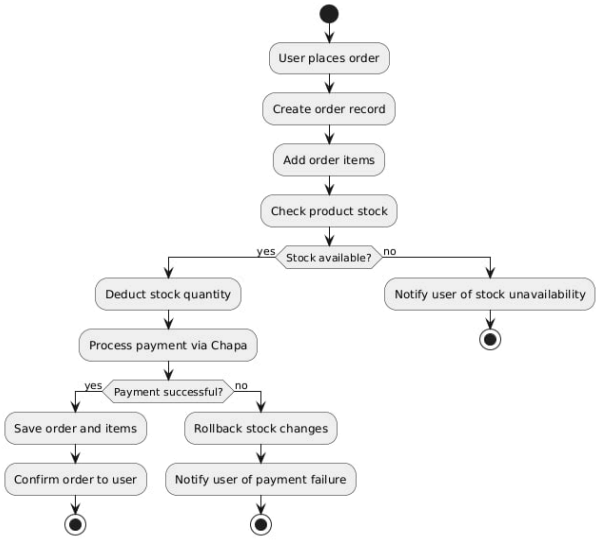


Figure 5: Activity Diagram

3.3.4.4 Collaboration Diagrams

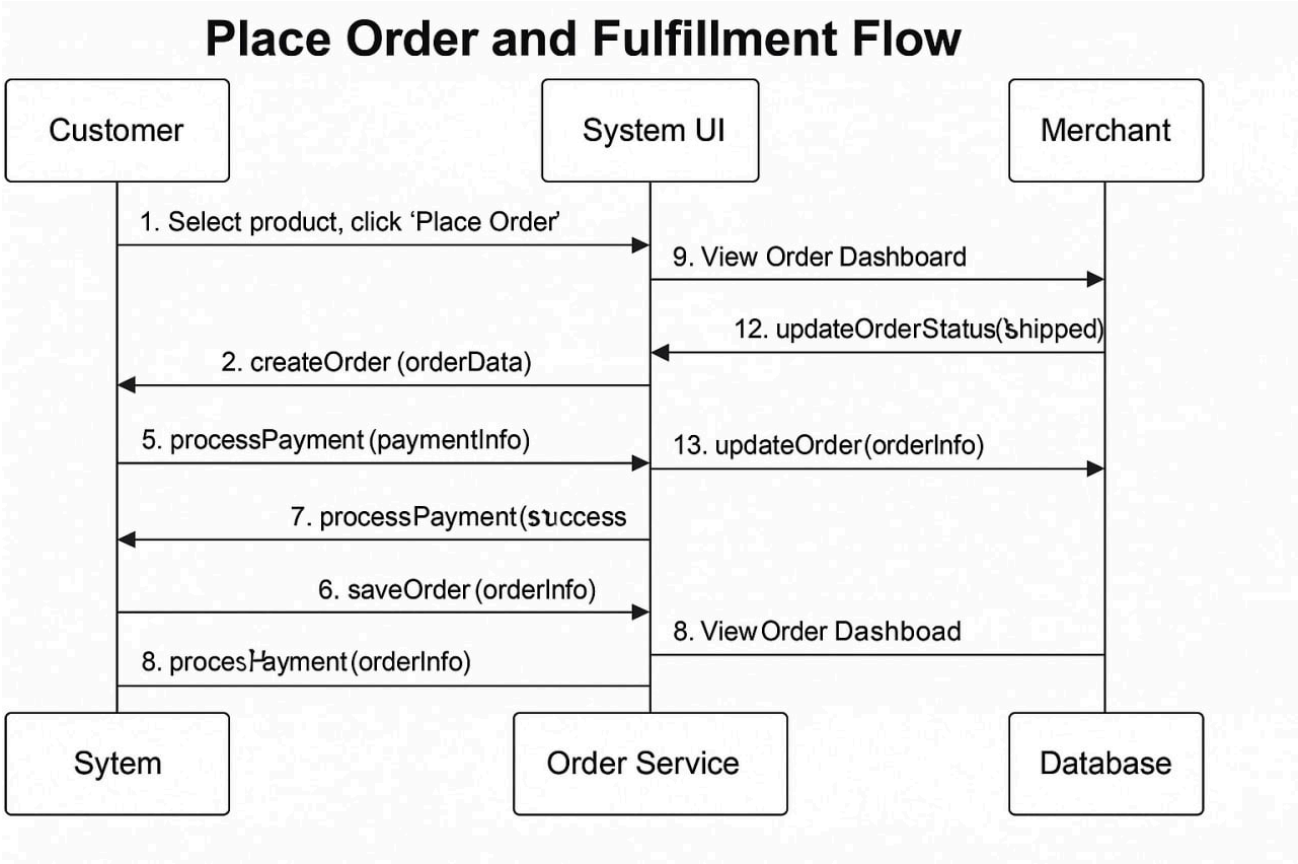


Figure 6: Collaboration Diagram

3.4 User Stories

The following user stories outline the essential functionalities required for ConnectX, reflecting the needs of different user roles within the e-commerce ecosystem.

3.4.1 User Account Management

As a platform user, I want my account details to be securely stored so that I can log in and access my personalized data safely.

- **Acceptance Criteria**:
    - Creation of a Users table with fields: id (Primary Key), name, email (Unique), password (hashed), created_at, updated_at.
    - Implementation of unique email validation to prevent duplicate accounts.
    - Successful schema implementation and testing with sample data.

3.4.2 Product Categorization

As a shopper, I want products to be grouped into categories so that I can easily browse items based on my preferences.

- **Acceptance Criteria**:

- Creation of a Products table with fields: id (Primary Key), name, description, price, stock_quantity, category_id (Foreign Key → Categories.id), created_at, updated_at.
- Creation of a Categories table with fields: id (Primary Key), name, created_at, updated_at.
- Each product is associated with a category, and multiple products can share the same category.
- Successful schema migrations and testing with sample data.

3.4.3 Order Tracking

As a customer, I want my order and its items to be tracked so that I can review my purchases and their details.

- **Acceptance Criteria**:
  - Creation of an Orders table with fields: id (Primary Key), user_id (Foreign Key → Users.id), order_status, total_price, created_at, updated_at.
  - Creation of an Order_Items table with fields: id (Primary Key), order_id (Foreign Key → Orders.id), product_id (Foreign Key → Products.id), quantity, price_per_unit, total_price.
  - An order can have multiple associated items via the Order_Items table.
  - Successful schema migrations and testing with sample data.

3.4.4 Product Restocking

As an entrepreneur, I want to request product restocks so that I can keep my inventory available to customers.

- **Acceptance Criteria**:
  - Creation of a Stock_Requests table with fields: id (Primary Key), user_id (Foreign Key → Users.id), product_id (Foreign Key → Products.id), requested_quantity, status (e.g., pending, approved, declined), created_at, updated_at.
  - A single user can make multiple requests for different products.
  - The table tracks the status of each request.
  - Successful schema migrations and testing with sample data.

3.4.5 Transaction Tracking

As an administrator, I want to track all product-related transactions so that I can monitor sales, restocks, and returns for the platform.

- **Acceptance Criteria**:
    - Creation of a Transactions table with fields: id (Primary Key), product_id (Foreign Key
      → Products.id), transaction_type (e.g., sale, restock, return), quantity, transaction_date, created_at.
    - Each transaction is associated with a specific product.
    - Accurate logging of transaction types.
    - Successful schema migrations and testing with sample data.

3.4.6 API Product Management

As a developer, I want to manage my product listings via APIs so that I can add, edit, or delete products easily.

- **Acceptance Criteria**:
    - Development of API endpoints:
        - POST /api/products: Add a new product (parameters: name, description, price, category, stock quantity).
        - PUT /api/products/{id}: Update product details.
        - DELETE /api/products/{id}: Delete a product by ID.
    - Validation to ensure required fields are provided and have valid data types.
    - APIs return appropriate success and error messages.

3.4.7 Product Search and Filtering

As a shopper, I want to search and filter products based on categories, price, and availability so that I can quickly find what I need.

- **Acceptance Criteria**:
    - Enhancement of GET /api/products endpoint to support parameters: category, min_price, max_price, and in_stock.
    - Products can be filtered and sorted based on provided parameters.
    - Paginated responses return 10 products per page by default.
    - API returns metadata such as total pages and current page.

3.4.8 Paginated Product Listings

As a developer (student and startup), I want paginated product listings so that I can navigate large datasets without performance issues.

- **Acceptance Criteria**:
    - Addition of pagination support to GET /api/products with query parameters:
        - page (default: 1)

        - limit (default: 10)
    - API response includes metadata:
        - total_items, total_pages, current_page, per_page.

3.4.9 API Endpoint Testing

As a developer, I want to test created API endpoints to ensure their reliability and correctness.

- **Acceptance Criteria**:
    - Writing and executing test cases for each API endpoint.
    - Validating responses for edge cases and error handling.

3.5 Design Pattern

3.5.1 Overview of Chosen Design Pattern

The **Model-View-Controller (MVC)** design pattern has been selected for the ConnectX backend platform. MVC is a widely adopted architectural pattern that promotes separation of concerns, enhancing scalability, maintainability, and testability of the application.

3.5.2 Components of MVC in ConnectX

Model

- **Responsibility**: Represents the data and business logic of the application.
- **Implementation**: Utilizes Django's Object-Relational Mapping (ORM) to define data models corresponding to database tables such as Users, Products, Orders, etc.
- Benefits:
    - Simplifies database interactions.
    - Ensures data integrity through Django’s built-in validation and migration tools.

View

- **Responsibility**: Handles the presentation layer and user interface.
- **Implementation**: In the context of a backend platform, views are implemented as API endpoints using Django REST Framework (DRF). These endpoints process HTTP requests, interact with models, and return appropriate responses.
- Benefits:
    - Facilitates the creation of RESTful APIs.
    - Enhances the modularity of the application by decoupling the API logic from the data models.

Controller

- **Responsibility**: Manages the communication between models and views.
- **Implementation**: Django's views (in DRF) act as controllers that process incoming requests, invoke model operations, and determine the appropriate responses.
- Benefits:
  - Streamlines request handling and response generation.
  - Promotes a clear and organized flow of data within the application.

3.5.3 Advantages of MVC for ConnectX

- **Separation of Concerns**: Clearly distinguishes data management, business logic, and user interface, making the system easier to develop and maintain.
- **Scalability**: Facilitates the addition of new features and modules without disrupting existing functionalities.
- **Reusability**: Encourages the reuse of components across different parts of the application, reducing redundancy and development time.
- **Testability**: Enhances the ability to perform unit testing and integration testing by isolating different components.

3.5.4 Implementation of MVC in ConnectX

- **Models**: Defined using Django's models.Model classes, representing entities like User, Product, Order, etc.
- **Views**: Implemented as DRF viewsets and API views, handling CRUD operations and business
  logic.
- **Controllers**: Managed by DRF's routers and serializers, ensuring seamless communication between models and views.

3.5.5 Architectural Diagram

To provide a holistic view of the ConnectX system, both the **Entity-Relationship Diagram (ERD)** and the **Architectural Diagram** are included. These diagrams complement each other by illustrating different aspects of the system's design and functionality.

Entity-Relationship Diagram (ERD)

The ERD visualizes the data structure and relationships within the ConnectX backend system. It highlights how different entities interact, ensuring a clear understanding of data flow and storage mechanisms.

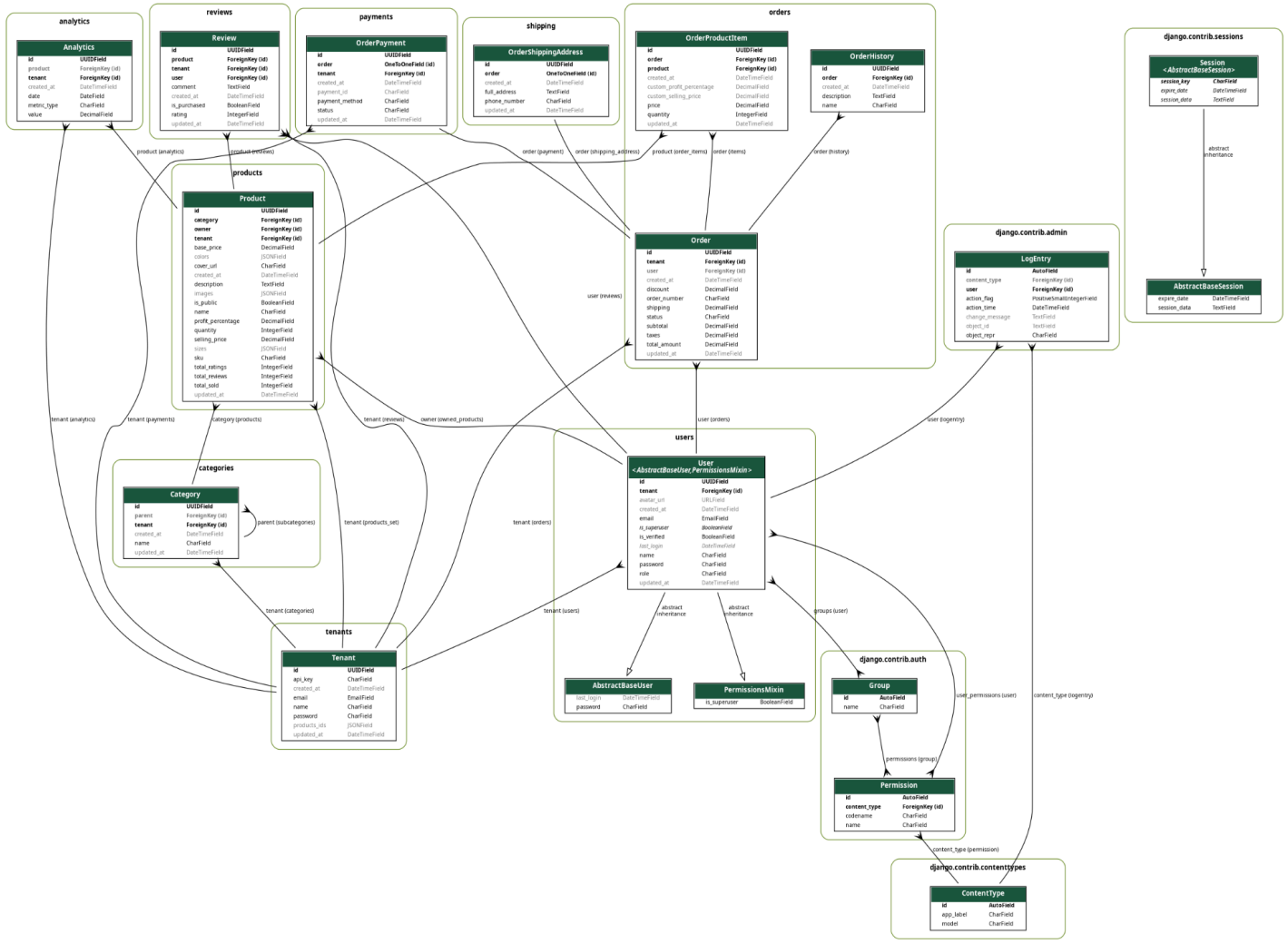


Figure 7: Entity-Relationship Diagram for ConnectX

Key Components:

- **Users**: Stores user information with unique email constraints.
- **Products**: Contains product details linked to categories.
- **Categories**: Defines product categories for better organization.
- **Orders**: Manages customer orders and their statuses.
- **Order_Items**: Details individual items within an order.
- **Transactions**: Logs all product-related transactions.

Architectural Diagram

The Architectural Diagram provides a high-level overview of the ConnectX system, showcasing the interaction between various components and services. It outlines how the frontend, backend, database, and third-party services like payment gateways collaborate to deliver a seamless e-commerce experience.

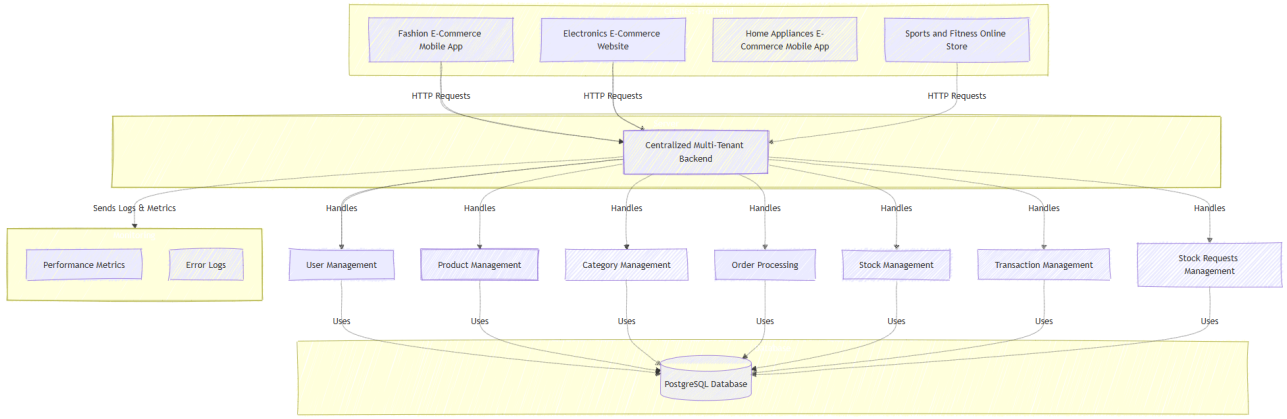


Figure 8: Architectural Diagram for ConnectX

Key Components:

- **Frontend (Next.js)**: Handles user interactions and displays data fetched from the backend APIs.
- **Backend (Django & DRF)**: Manages business logic, API endpoints, and data processing.
- **Database (PostgreSQL)**: Stores all persistent data, including user information, product details, orders, and transactions.
- **Payment Gateway (Chapa)**: Facilitates secure payment processing.
- **Authentication Service (JWT)**: Manages user authentication and authorization.
- **Hosting (Vercel & Render)**: Deploys the frontend and backend services, ensuring scalability and reliability.
- **Containerization (Docker)**: Ensures consistent environments across development and production.
- **CI/CD Pipeline (GitHub Actions)**: Automates testing and deployment processes, enhancing development efficiency.

Integration of ERD and Architectural Diagram

Together, the ERD and Architectural Diagram offer a comprehensive view of both the data relationships and the system's structural layout. This dual representation ensures that stakeholders can understand both the underlying data model and the high-level system architecture, facilitating better planning, development, and maintenance.

3.6 Model Validation

This section presents the verification and validation activities conducted to ensure the correctness, consistency, and completeness of the problem analysis and system models developed throughout the

project.

3.6.1 Verification of Problem Analysis

To verify the problem analysis:

- The initial problem statement and objectives were reviewed against the identified stakeholder needs.
- Each problem identified during requirement gathering was mapped to one or more corresponding system functionalities or use cases.
- The use case identification (UC01–UC08) was cross-checked to ensure all essential user interactions were captured without redundancy or omission.

3.6.2 Validation of System Models

The system models were validated through the following:

- **Use Case Models**: Ensured that each actor (Customer, Merchant, Admin, Payment Gateway) is properly represented and their interactions align with real-world workflows. Use case flows (main and alternate) were traced back to functional requirements.
- **Data Models / ERDs**: Validated that all necessary entities (e.g., User, Product, Order, Payment) and relationships are defined with correct attributes and constraints, reflecting real business rules.
- **Sequence Diagrams**: Validated message exchanges against each use case's flow of events to ensure accurate system behavior representation.
- **Class Models**: Ensured that all entities, attributes, methods, and relationships align with the data and behavior requirements.

3.6.3 Peer and Supervisor Review

- Draft models were reviewed by project group members for internal consistency.
- Supervisor feedback was incorporated to correct logical errors and ensure models reflect actual system expectations.
- Review sessions were used to refine models iteratively based on evolving understanding and feedback.

3.6.4 Completeness and Consistency Check

- A traceability approach was applied to confirm that every requirement has been addressed by at least one use case or model element.
- Models were checked for internal consistency — no contradictory or redundant relationships, attributes, or flows were identified.
- Final validation confirmed that the models are sufficient to guide implementation without ambiguity.

Chapter Four: System Design

4.1. Overview

This chapter outlines the comprehensive system design for ConnectX, establishing a blueprint for our centralized, scalable backend platform for e-commerce. The design decisions documented here are strategically aligned with our objective to empower aspiring entrepreneurs, startups, and students by providing a robust, yet accessible e-commerce backend solution. Our design approach prioritizes scalability, security, and developer experience to ensure ConnectX meets the diverse needs of its target users while overcoming the technical and financial barriers identified in previous chapters.

4.2. Specifying the Design Goals

The design of ConnectX is guided by the following core objectives:

1. **Scalability**: Create a system architecture that efficiently handles growing numbers of users, products, and transactions without performance degradation.
2. **Multi-tenancy**: Implement secure data isolation while enabling resource sharing across multiple e-commerce platforms within a unified backend.
3. **Developer Experience**: Design intuitive APIs and comprehensive documentation that reduce the learning curve and accelerate integration for developers with varying skill levels.
4. **Security**: Ensure robust data protection and secure authentication mechanisms to safeguard sensitive user and transaction information.
5. **Flexibility**: Support diverse e-commerce scenarios through customizable product listings and order processing workflows.
6. **Performance**: Optimize response times and resource utilization to deliver a seamless experience for both developers and end-users.
7. **Maintainability**: Structure the codebase with clear separation of concerns to facilitate ongoing development and feature enhancements.

4.3. System Design

4.3.1. Proposed Software Architecture

ConnectX employs a modular monolithic architecture using Django's MVC pattern. This architectural choice balances rapid development with scalability, particularly suitable for the initial phase of our project.

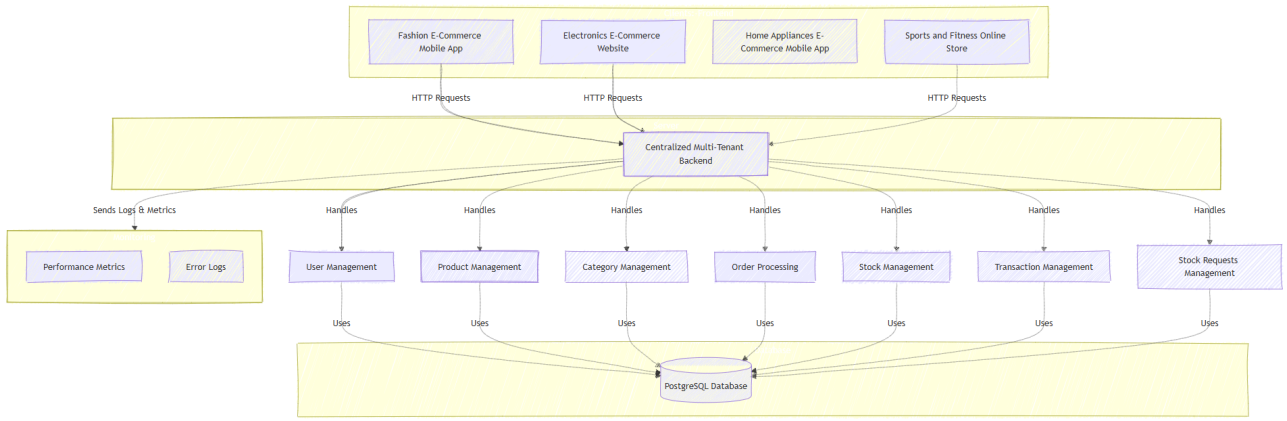


Figure 4.1: ConnectX Software Architecture

Key components of the architecture include:

1. **API Layer**: Implements RESTful endpoints using Django REST Framework for all core functionalities including product management, order processing, and user authentication.
2. **Service Layer**: Contains business logic and orchestrates interactions between different components. This layer houses services for order fulfillment, and payment processing.
3. **Data Access Layer**: Manages database operations through Django's ORM, providing an abstraction over direct database queries.
4. **Cross-Cutting Concerns**: Addresses aspects like security, logging, and caching that apply across multiple components of the system.

4.3.2. Subsystem Decomposition

ConnectX is divided into the following subsystems, each responsible for specific functionalities:

User Management Subsystem

- User registration and authentication
- Profile management
- Role-based access control
- JWT token issuance and validation

Product Management Subsystem

- Centralized product catalog management
- User-specific product listings
- Product categorization

Order Processing Subsystem

- Order creation and management
- Cart functionality
- Order status tracking
- Order fulfillment workflows

Payment Subsystem

- Integration with Chapa payment gateway
- Transaction processing
- Payment status tracking
- Refund handling

API Gateway Subsystem

- Request routing
- Rate limiting
- Authentication and authorization checks
- Response formatting

4.3.3. Database Design

ConnectX utilizes PostgreSQL as its primary database, leveraging its robustness for transactional data and advanced features for complex queries.

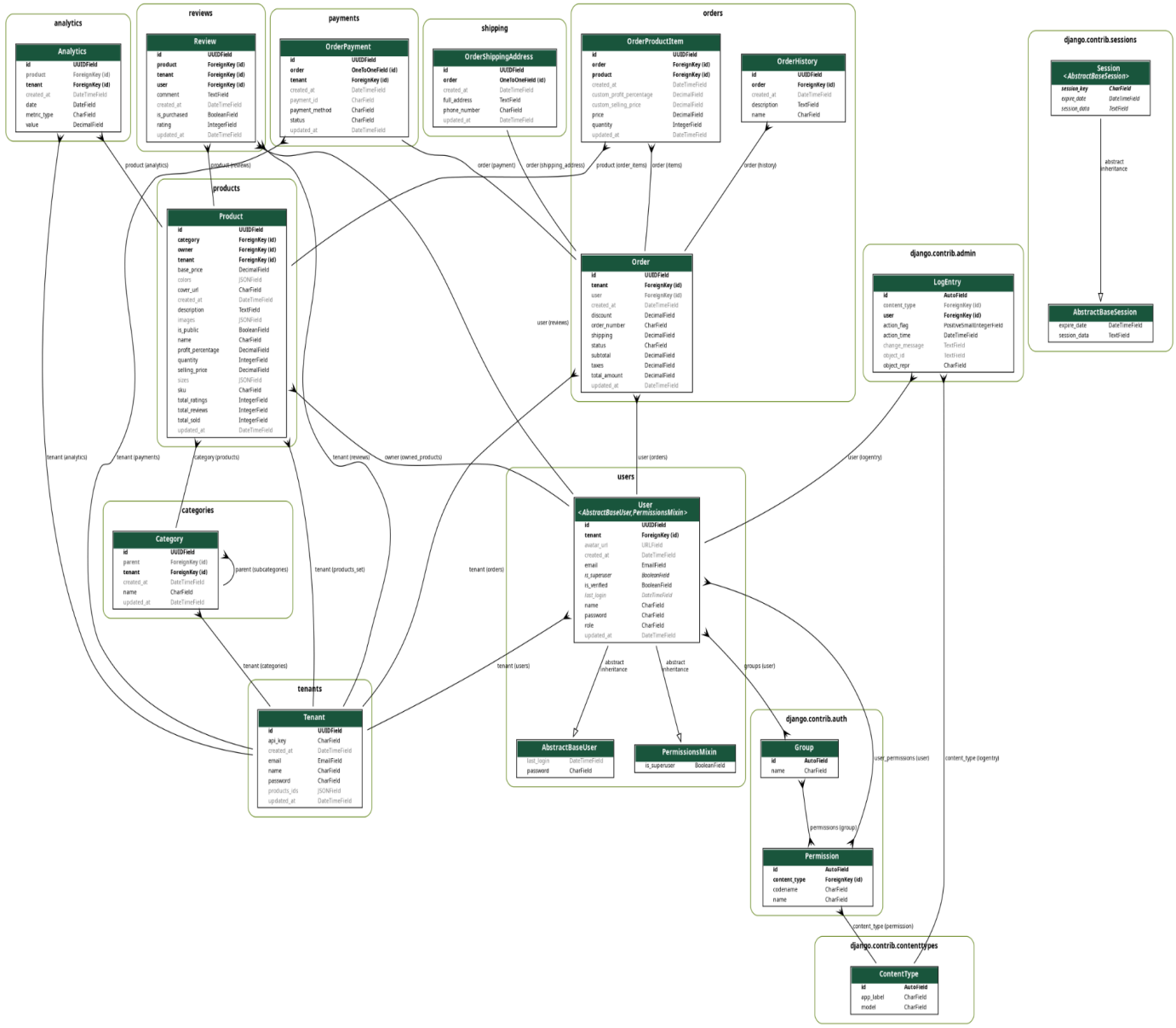


Figure 9: ConnectX Database Schema

Key Database Tables and Relationships:

1. Tenants Table

- Fields: id (PK), name, email, payment_status, created_at, updated_at
- Description: Each tenant represents a separate entity using the system.
- Relationships:
    - One-to-many relationship with Users, Products, Categories, Analytics, Reviews.

2. Users Table

- Fields: id (PK), tenant_id (FK), username, email, password, first_name, last_name, is_active, is_staff, is_superuser, created_at, updated_at
- Constraints: Unique email, username
- Relationships:
    - Many-to-one with Tenants
    - One-to-many with Orders
    - One-to-many with Reviews
    - Linked to Django authentication (Group, Permission)

3. Products Table

- Fields: id (PK), tenant_id (FK), category_id (FK), owner_id (FK, User), name, description, price, stock_quantity, is_public, discount_percentage, rating_avg, rating_count, latest_reviews, latest_rating, created_at, updated_at
- Relationships:
    - Many-to-one with Categories
    - Many-to-one with Users (owner)
    - Many-to-one with Tenants
    - One-to-many with OrderProductItems
    - One-to-many with Reviews
    - One-to-many with Analytics

4. Categories Table

- Fields: id (PK), tenant_id (FK), parent_id (self-referential FK), name, created_at, updated_at
- Relationships:
    - Many-to-one with Tenants
    - Self-referential for hierarchical categories
    - One-to-many with Products

5. Orders Table

- Fields: id (PK), user_id (FK), tenant_id (FK), status, total_amount, tax_amount, shipping_fee, discount_amount, created_at, updated_at

- Relationships:

    - Many-to-one with Users
    - Many-to-one with Tenant
    - One-to-many with OrderProductItems
    - One-to-one with OrderShippingAddress
    - One-to-many with OrderPayments
    - One-to-many with OrderHistories

6. Order Product Items Table

- Fields: id (PK), order_id (FK), product_id (FK), tenant_id (FK), quantity, price_at_purchase, created_at
- Relationships:
    - Many-to-one with Orders
    - Many-to-one with Products
    - Many-to-one with Tenants

7. OrderShippingAddress Table

- Fields: id (PK), order_id (FK), full_name, street_address, city, postal_code, country
- Relationships:
    - One-to-one with Orders

8. OrderPayment Table

- Fields: id (PK), order_id (FK), amount, payment_method, transaction_id, status, created_at
- Relationships:
    - Many-to-one with Orders

9. OrderHistory Table

- Fields: id (PK), order_id (FK), status, description, created_at
- Relationships:
    - Many-to-one with Orders

10. Reviews Table

- Fields: id (PK), user_id (FK), product_id (FK), rating, comment, created_at
- Relationships:

    - Many-to-one with Users
    - Many-to-one with Products

11. Analytics Table

- Fields: id (PK), tenant_id (FK), product_id (FK), view_count, purchase_count, created_at
- Relationships:
    - Many-to-one with Products
    - Many-to-one with Tenants

12. Django System Tables (Authentication & Sessions)

- Group / Permission / ContentType / Session
    - Standard Django tables for authentication and authorization management.
    - Sessions are tracked using the Session and AbstractBaseSession tables.

Multi-Tenancy Approach:

ConnectX implements row-level multi-tenancy, where data from different e-commerce stores is stored in the same tables but separated by user or organization identifiers. This approach:

- Maximizes resource sharing
- Simplifies backup and maintenance
- Enables efficient cross-tenant analytics
- Maintains data isolation through application-level access controls

4.3.4. Deployment Diagram

The deployment architecture of ConnectX is designed for reliability, scalability, and cost-effectiveness, utilizing cloud-based services.

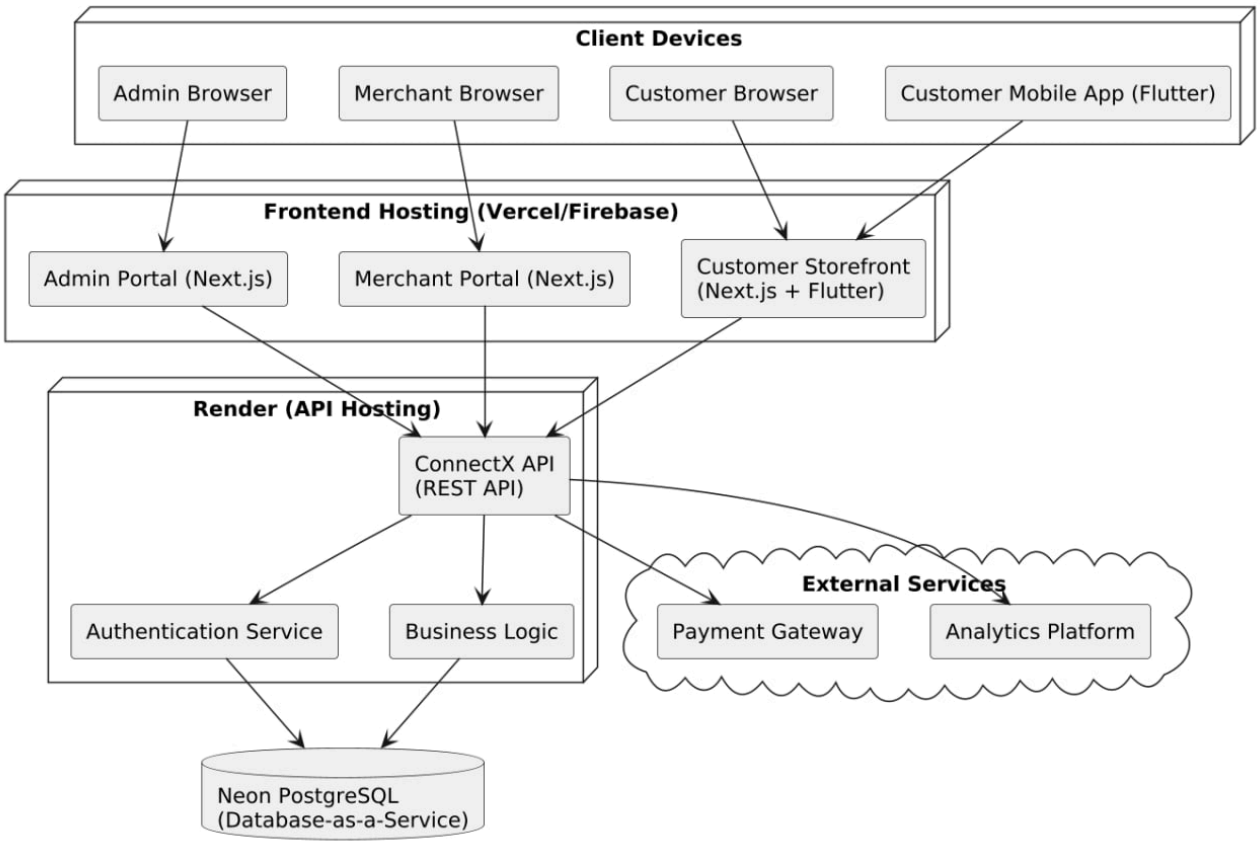


Figure 10: ConnectX Deployment Architecture

Key components of the deployment architecture:

1. **Frontend Deployment on Vercel**
    - Next.js application serving the sample e-commerce storefront
    - Static assets delivery via CDN
    - Server-side rendering for improved SEO and performance
2. **Backend Deployment on Render**
    - Django application containerized with Docker
    - Auto-scaling based on demand
    - Health monitoring and automatic recovery
3. **Database Hosting**
    - PostgreSQL database with automated backups
    - Read replicas for improved performance
    - Connection pooling for efficient resource utilization
4. **CI/CD Pipeline**
    - GitHub Actions for automated testing and deployment
    - Test, build, and deployment stages
    - Release management and versioning

4.3.5. User Interface Design

While ConnectX primarily focuses on backend functionality, a simple administrative interface and sample storefront are provided to demonstrate the system's capabilities.

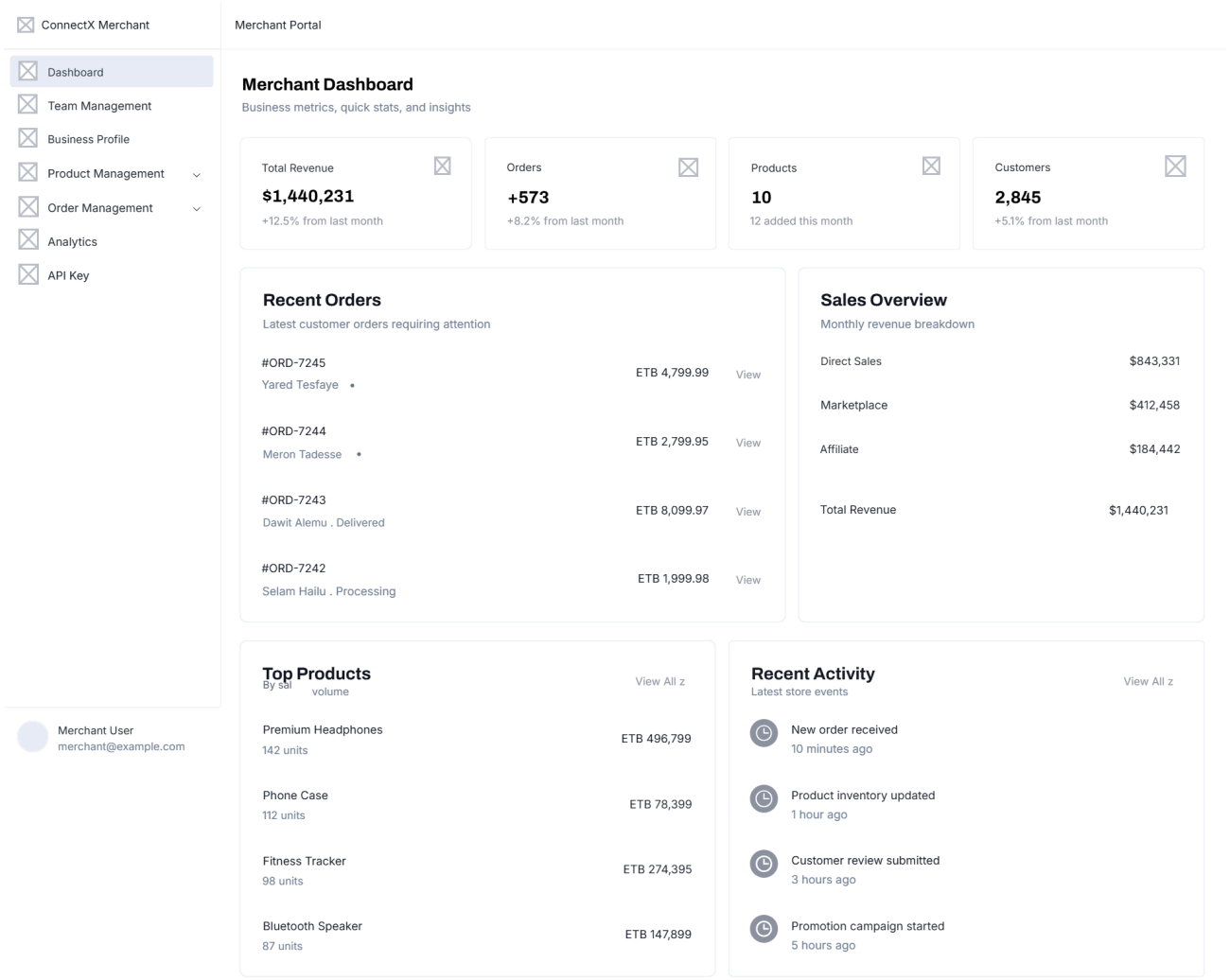


Figure 11: ConnectX Admin Dashboard Wireframe

The admin interface includes:

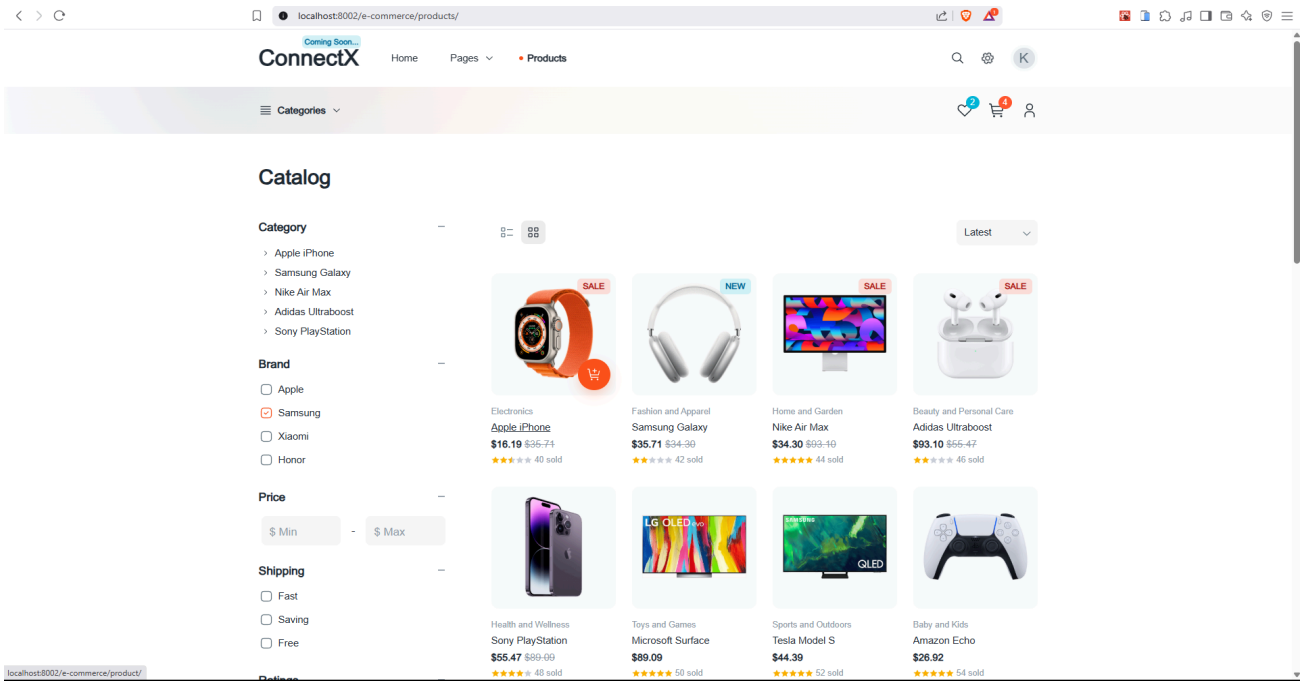
- Product management dashboard
- Order tracking and fulfillment
- User management
- Analytics and reporting
- System configuration

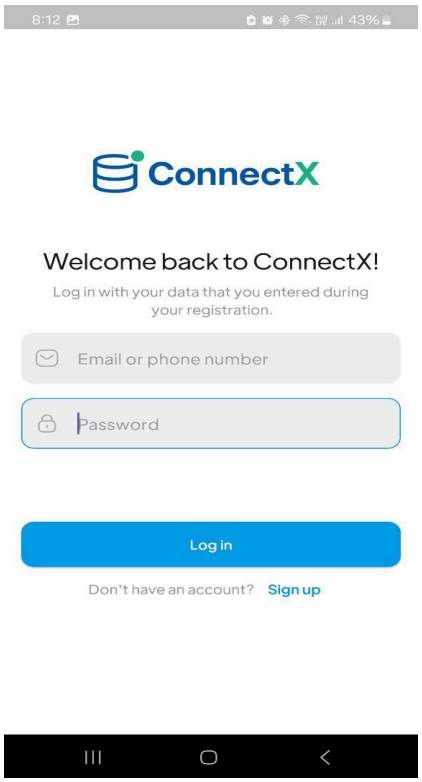
Sample storefront features:

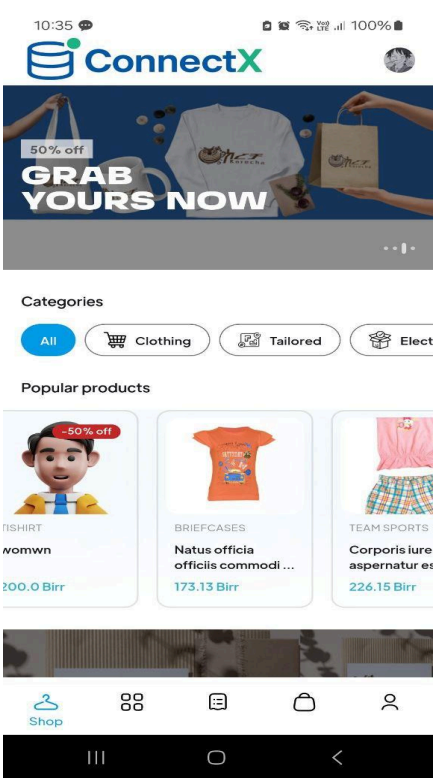
- Product browsing and searching

- Shopping cart functionality
- Checkout process
- Order history
- User account management

Template







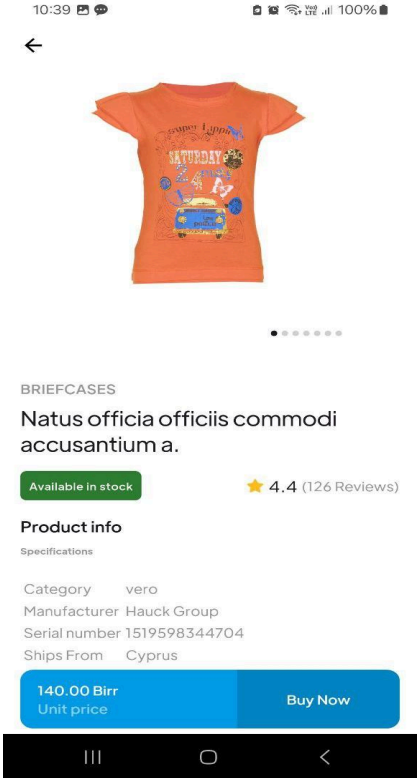


Figure 12: Ecommerce template for customer

4.3.6. System Integration

ConnectX integrates with several external systems and services to provide comprehensive functionality:

1. **Payment Gateway Integration**
    - Chapa payment processing for local payments
    - Secure API communication using encryption
    - Webhook handling for payment status updates
2. **Email Service Integration**
    - Order confirmations
    - Shipping notifications
    - Marketing communications
    - System alerts
3. **File Storage Integration**
    - Product image storage and delivery
    - Document management for invoices and receipts
    - Backup storage
4. **Analytics Integration**
    - User behavior tracking
    - Sales performance monitoring

4.3.7. Security Design

Security is a fundamental consideration in ConnectX's design, addressing various aspects:

1. **Authentication and Authorization**
    - JWT-based authentication for API access
    - Role-based access control
    - Session management and timeout policies
2. **Data Protection**
    - Encryption of sensitive data at rest and in transit
    - PCI DSS compliance for payment handling
    - Data anonymization for analytics
3. **API Security**
    - Rate limiting to prevent abuse
    - Input validation and sanitization
    - Protection against common attacks (SQL injection, XSS, CSRF)
4. **Infrastructure Security**

- Firewall configuration
- Regular security updates
- Network segmentation

5. **Audit and Compliance**
   - Comprehensive logging
   - Audit trails for sensitive operations
   - Compliance with relevant regulations

4.4. Verifying the Requirements in the Design

The ConnectX system design has been carefully evaluated against the functional and non-functional requirements specified in Chapter Three. The following table summarizes how the design addresses each key requirement:

Table 16: Requirement verifications

Requirement	Design Implementation	Verification Method
Multi-Tenancy Support	Row-level multi-tenancy in database design	Database schema review and tenant isolation testing
Order Processing	Order Processing Subsystem with comprehensive workflow	End-to-end order flow testing
User Authentication	JWT-based authentication with role-based access control	Security review and penetration testing
API Integration	Well-documented RESTful APIs with consistent patterns	API documentation review and integration testing
Payment Integration	Secure integration with Chapa payment gateway	Transaction flow testing and security audit
Scalability	Modular architecture with horizontal scaling capabilities	Load testing with simulated traffic
Security	Comprehensive security measures at all system levels	Security audit and vulnerability assessment
Reliability	Redundancy in deployment and error handling mechanisms	Failover testing and chaos engineering
Usability	Intuitive API design and comprehensive documentation	Developer experience review and usability testing

Requirement	Design Implementation	Verification Method
Performance	Optimized queries and caching strategies	Performance benchmarking and response time analysis

Through this verification process, we ensure that the ConnectX design meets all specified requirements while providing a solid foundation for future enhancements and extensions.

Chapter Five: System Implementation

5.1. Reviewing the Design Solution

The design solution for ConnectX was developed to address the challenges of high development costs, lack of technical skills, and operational complexities in e-commerce backend systems. The system adopts a multi-tenant, centralized backend platform using Django and Django REST Framework (DRF), with a PostgreSQL database to ensure scalability and data isolation. The Model-View-Controller (MVC) pattern, as described in Chapter 3, organizes the backend logic, separating data models, API endpoints, and business logic for maintainability.

The Entity-Relationship Diagram (ERD) guided the database design, defining relationships between entities like Users, Products, and Orders. The architectural diagram outlined the interaction between the backend (Django), frontend (Next.js), and third-party services like Chapa for payments. This review confirmed that the design aligns with the functional requirements (e.g., order processing, user authentication) and non-functional requirements (e.g., scalability, security) outlined in the previous chapter.

5.2. Deciding on the Development Tools

The development tools were selected to ensure rapid development, scalability, and ease of integration, as planned in the methodology :

- **Backend Framework**: Django and Django REST Framework were chosen for their robustness, security features, and rapid API development capabilities. Django's ORM simplifies database interactions, while DRF supports RESTful API creation with built-in authentication and throttling.
- **Frontend Framework**: Next.js was selected for its server-side rendering capabilities, which enhance SEO(Search Engine Optimization) and performance for the e-commerce frontend.
- **Database**: PostgreSQL was used for its reliability, support for complex queries, and scalability, crucial for handling multi-tenant data.
- **Authentication**: Django's built-in authentication system, augmented with JSON Web Tokens (JWT), ensures secure, stateless API access.
- **Payment Gateway**: Chapa was integrated for secure payment processing, supporting seamless transactions.

- **Deployment**: Vercel (frontend) and Render (backend) were chosen for cost-effective hosting, with Docker for containerization to ensure consistency across environments.
- **Version Control**: Git and GitHub facilitated collaborative development and version tracking.
- **Testing Tools**: Jest(JavaScript testing framework) used for unit testing and Playwright (end-to-end ui-testing).
- **Documentation**: Swagger was utilized to auto-generate interactive API documentation.

These tools align with the project's Agile methodology, enabling iterative development and continuous integration.

5.3. Developing the Solution

5.3.1. Backend Development

The backend was developed using Django and DRF, following the MVC pattern. Models were created for core entities like Users, Products, and Orders, as defined in the ERD. PostgreSQL schemas were implemented with multi-tenant support using row-based isolation, ensuring data separation between tenants. Django migrations were used to manage database schema changes.

Key backend modules included:

- **User Authentication(users)**: Implemented using Django's authentication system and JWT for API access. User registration, login, and role-based access control were developed.
- **Order Processing (orders)**: Contains the logic to create and manage customer orders, associate order items with products, and update order statuses (e.g., pending, processing, shipped, delivered).
- **Product Management (products):** This module is responsible for managing product information, including details such as name, description, price, and inventory. It interacts with the categories module.
- **Category Management (categories):** Manages the classification of products into different categories for organization and browsing.
- **Reviews and Ratings (reviews)**: Allows users to submit reviews and ratings for products.
- P**ayment Processing (payments)**: Integrates with payment gateways to handle financial transactions for orders.
- **Shipping Management (shipping):** Manages shipping addresses, methods, and integrates with shipping carriers to track shipments.
- **Analytics (analytics):** This module gathers and processes data to provide insights into sales, user behavior, and other key metrics.
- **Core Functionality (core):** This module contains shared utilities, base models, or configurations used across different applications within the backend.
- **Multi-Tenancy (tenants):** This module implements multi-tenancy, enabling the application to support multiple tenants with isolated data and configurations.

The backend was containerized using Docker to ensure consistency between development and production environments.

5.3.2. API Implementation

The ConnectX backend creates a centralized, scalable platform supporting many e-commerce platforms by means of RESTful APIs using Django REST Framework . By handling basic tasks including user management, product listings, order processing, payments, and analytics, this centralized backend helps retailers and developers to concentrate on frontend development and corporate expansion. The API implementation is broken out in great detail below:

Endpoints

The APIs are divided into modular Django apps, each of which handles a certain area of the application. This modular framework enables maintainability and scalability, allowing the centralized backend to easily manage various tenants. The key endpoints are:

Users & Authentication:

- Users: /api/users/
  - Handles user registration, and profile management.
  - POST /api/users/: Creates a new user account. Expected fields: email, password, name, etc. (as defined in UserSerializer). Validates unique email constraints and hashes passwords.
  - GET /api/users/: Retrieves a list of users (pagination applies).
  - GET /api/users/{id}/: Retrieves a specific user's details.
  - PUT /api/users/{id}/: Updates a specific user's information.
  - PATCH /api/users/{id}/: Partially updates a specific user's information.
  - DELETE /api/users/{id}/: Deletes a specific user.
  - GET /api/users/me/: Retrieves the authenticated user info.
  - PUT /api/users/{id}/update-profile/: Updates user profile information.
- JWT Authentication Endpoints:
  - POST /api/auth/login/: Authenticates users and returns access and refresh tokens (uses CustomTokenObtainPairView).
  - POST /api/auth/refresh/: Refreshes the access token using a valid refresh token (uses TokenRefreshView).

Products:

- Products: /api/products/
  - Manages product creation, updates, and retrieval.
  - POST /api/products/: Adds a new product. Expected fields: name, description, price, category, etc. (as defined in ProductSerializer).
  - GET /api/products/: Retrieves paginated product listings. Supports filtering (e.g., by category, price range) and sorting (e.g., by price, name) as implemented in the ProductViewSet.
  - GET /api/products/{id}/: Retrieves detailed information for a specific product.
  - PUT /api/products/{id}/: Updates an existing product’s details.
  - PATCH /api/products/{id}/: Partially updates an existing product’s details.

- DELETE /api/products/{id}/: Deletes a product.
- *GET /api/products/{id}/list-to-tenant :* List the product to their own
- *GET /api/products/{id}/unlist-to-tenant:*To remove the listed product product

Categories:

- Categories: /api/categories/ (Handled by CategoryViewSet)
  - Manages product categories.
  - POST /api/categories/: Creates a new category.
  - GET /api/categories/: Retrieves a list of categories.
  - GET /api/categories/{id}/: Retrieves a specific category.
  - PUT /api/categories/{id}/: Updates a specific category.
  - PATCH /api/categories/{id}/: Partially updates a specific category.
  - DELETE /api/categories/{id}/: Deletes a specific category.

Reviews:

- Reviews: /api/reviews/ (Handled by ReviewViewSet)
  - Allows users to submit and view reviews for products.
  - POST /api/reviews/: Submits a review. Expected fields: rating, comment, product ID
  - GET /api/reviews/: Retrieves reviews. Supports filtering (e.g., by user, product, rating) and sorting (e.g., by date, rating)
  - GET /api/reviews/{id}/: Retrieves a specific review.
  - PUT /api/reviews/{id}/: Updates a specific review.
  - PATCH /api/reviews/{id}/: Partially updates a specific review.
  - DELETE /api/reviews/{id}/: Deletes a specific review.

Orders:

- Orders: /api/orders/ (Handled by OrderViewSet)
  - Manages order creation, updates, and status tracking.
  - POST /api/orders/: Creates an order with associated items.
  - GET /api/orders/: Retrieves a list of orders for the authenticated user or merchant, with filtering by status.
  - GET /api/orders/{id}/: Retrieves a specific order.
  - PUT /api/orders/{id}/: Updates an order (e.g. status).
  - PATCH /api/orders/{id}/: Partially updates an order.
  - DELETE /api/orders/{id}/: Deletes an order (if applicable).

Payments:

- Payments: /api/payments/ (Handled by OrderPaymentViewSet)
  - Handles payment processing and transaction history.
  - POST /api/payments/: Initiates/records a payment for an order.
  - GET /api/payments/: Retrieves payment history. Supports filtering (e.g., by date, status) as implemented in OrderPaymentViewSet.
  - GET /api/payments/{id}/: Retrieves a specific payment record.
  - PUT /api/payments/{id}/: Updates a payment record.
  - PATCH /api/payments/{id}/: Partially updates a payment record.
  - DELETE /api/payments/{id}/: Deletes a payment record (if applicable).

Shipping:

- Shipping Addresses: /api/shipping-addresses/ (Handled by ShippingAddressViewSet)
    - Manages user shipping addresses.
    - POST /api/shipping-addresses/: Creates a new shipping address.
    - GET /api/shipping-addresses/: Retrieves a list of shipping addresses for the authenticated user.
    - GET /api/shipping-addresses/{id}/: Retrieves a specific shipping address.
    - PUT /api/shipping-addresses/{id}/: Updates a specific shipping address.
    - PATCH /api/shipping-addresses/{id}/: Partially updates a specific shipping address.
    - DELETE /api/shipping-addresses/{id}/: Deletes a specific shipping address.

Analytics:

- Analytics: /api/analytics/ (Handled by AnalyticsViewSet)
    - Provides insights into platform performance.
    - The router will provide:
        - GET /api/analytics/: Lists available analytic reports or a summary.
        - GET /api/analytics/{id}/: Retrieves a specific analytic report or data point.
        - (POST, PUT, PATCH, DELETE might not be applicable for analytics depending on your design).

Tenants:

- Tenants: /api/tenants/ (Handled by TenantViewSet)
    - Supports multi-tenancy by managing tenant-specific configurations and data isolation.
    - POST /api/tenants/: Registers a new tenant (merchant).
    - GET /api/tenants/: Retrieves a list of tenants.
    - GET /api/tenants/{id}/: Retrieves a specific tenant.
    - PUT /api/tenants/{id}/: Updates tenant settings.
    - PATCH /api/tenants/{id}/: Partially updates tenant settings.
    - DELETE /api/tenants/{id}/: Deletes a tenant.

Features

- **ModelViewSet**: Most APIs leverage DRF's ModelViewSet to provide out-of-the-box CRUD operations. For example, the ProductViewSet handles all product-related endpoints, reducing boilerplate code and ensuring consistency.
- **Custom Permissions**: Custom permission classes like is_verified and role restrict access to certain endpoints (e.g., submitting reviews) to users who have purchased the product. Role-based permissions ensure admins, merchants, and customers access only their authorized resources.
- **Search and Filtering**: APIs facilitate advanced filtering and search through DRF's SearchFilter and DjangoFilterBackend. For example, the GET /api/products/ endpoint supports filtering by categoryname , product visibility, or price range, and searching by keywords in the products.
- **Pagination**: DRF's PageNumberPagination class is utilized to paginate large data sets with configurable page size (default 10 items per page). Responses include metadata like total data of items, total pages, and current page so that frontend applications can manage the data effectively.

- **Multi-Tenant Data Isolation**: Tenant-specific data is isolated by row-based filtering in database queries so that merchants can view and manage their own listed products , orders, and analytics.

Interactive Documentation

For internal use, we utilized Swagger UI, accessible at /swagger/, to provide an interactive interface for developers to explore and test API endpoints, complete with detailed request/response schemas, authentication requirements, and sample payloads.

- **Swagger UI**: Available at /swagger/, providing an interactive interface for developers to explore and test API endpoints. It includes detailed request/response schemas, authentication requirements, and sample payloads.
- **ReDoc**: Available at /redoc/, offering a clean, structured view of the API documentation with a focus on readability and navigation.
- **Static Assets**: Static files for Swagger and ReDoc are served from the static/drf-yasg directory, ensuring seamless integration with the API documentation and fast load times.

5.3.3. Template Development

The frontend was developed using Next.js, focusing on a user-friendly, multi-tenant e-commerce platform with tailored interfaces for different user roles. A landing page was created to provide an entry point for users, directing them to role-specific dashboards after authentication. The system implemented role-based access control, unified authentication with single sign-on, and customizable interfaces for our system admins, and merchants. A notification system enabled cross-role communication, such as notifying merchants of new orders.

Customer-Facing Interface

- **Product Management**: Dynamic pages displaying products with filtering and search capabilities, fetching data via the GET /api/products endpoint.
- **User Profile**: A section for customers to manage their account details and view order history, integrated with the backend's authentication system.
- **Shopping Cart and Checkout**: A cart system allowing customers to add products and proceed to checkout, interfacing with the POST /api/orders endpoint for order creation and Chapa for payment processing.

System Administrator Interface

The admin interface provides tools for platform oversight and merchant management:

- **System Dashboard**: Displays real-time monitoring of services, including API uptime and database performance, using data fetched from backend metrics endpoints.
- **Tenant Management**: Allows admins to view and manage all merchants, with options to search, filter, and view merchant details.

- **Service Configuration**: Enables global settings adjustments, such as enabling/disabling features like payment gateways.
- **SystemAnalytics**: Shows system-wide analytics, including API response times and resource utilization, visualized with charts.
- **Merchant Approval Workflow**: Admins can review and approve new merchant registrations, with a workflow to track pending and approved merchants.
- **Merchant Directory**: A searchable list of merchants, allowing admins to filter by status (e.g., active, suspended) and view account details.
- **Account Intervention Tools**: Features to suspend, restore, or modify merchant privileges, with audit logs for tracking actions.
- **Bulk Operations**: Admins can apply changes (e.g., suspend accounts) to multiple merchants at once
- **Dark and Light Mode Support:** A toggleable dark and light themes are available across the admin interface, reducing eye strain and enhancing visual comfort, especially during extended use or nighttime work sessions.

.

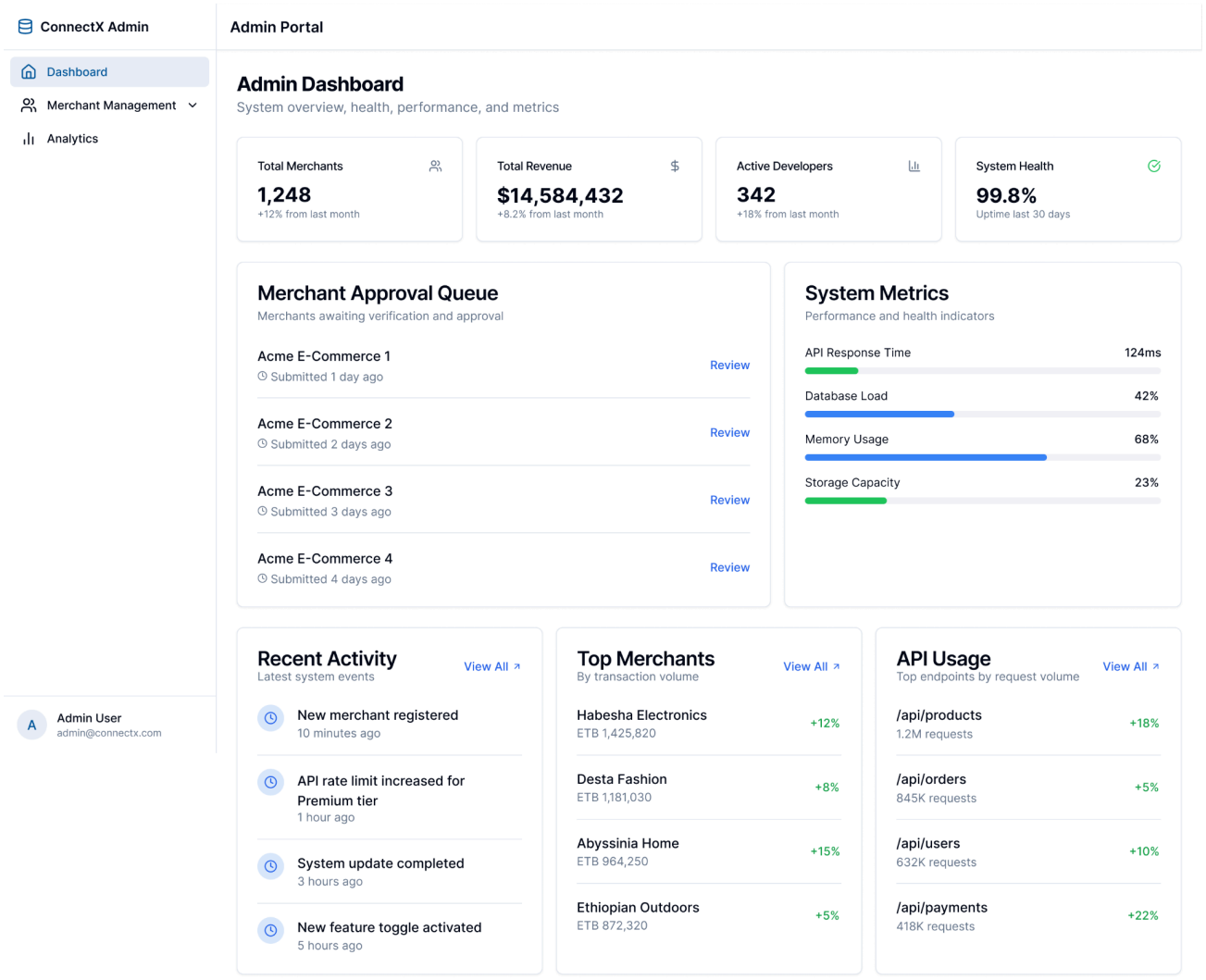


Figure 13: Admin Dashboard snapshot

Merchant Interface

The merchant interface empowers merchants to manage their business operations:

- **Merchant Dashboard**: Displays business metrics (e.g., total orders, revenue), service status, and quick actions like adding a product.
- **Team Management**: Merchants can manage team members with specific roles (e.g., editor, viewer) and permissions.
- **Business Profile**: Allows merchants to update and manage company details, branding (e.g., logo), and payment information for receiving payouts.
- **Product Catalog**: Verified merchants can add, edit, and organize products with details like name, description, price, category, and images. And verified merchants can toggle products between public (accessible to developers) and private, controlling which products are displayed in each context. In addition to this All merchants have a permission to list added

products from backend that are available public(available for everyone) or private(only available for verified merchants).

- **Order Processing**: Merchants can view, manage, and fulfill orders, with status updates (e.g., shipped, delivered).
- **Customer Database**: A comprehensive view of customer information, including order history and contact details.
- **Returns & Refunds**: Merchants can process return requests and issue refunds, with a workflow to track return status.
- **Sales Performance**: Analytics to track revenue, product performance, and sales trends, visualized with graphs.
- **Channel Analytics**: Compares performance across sales channels (e.g., direct sales vs. developer-integrated sales).
- **Customer Insights**: Provides behavior patterns (e.g., frequently purchased products) and demographic information.
- **Dark Mode Support:** A toggleable dark theme is available across the entire merchant interface.

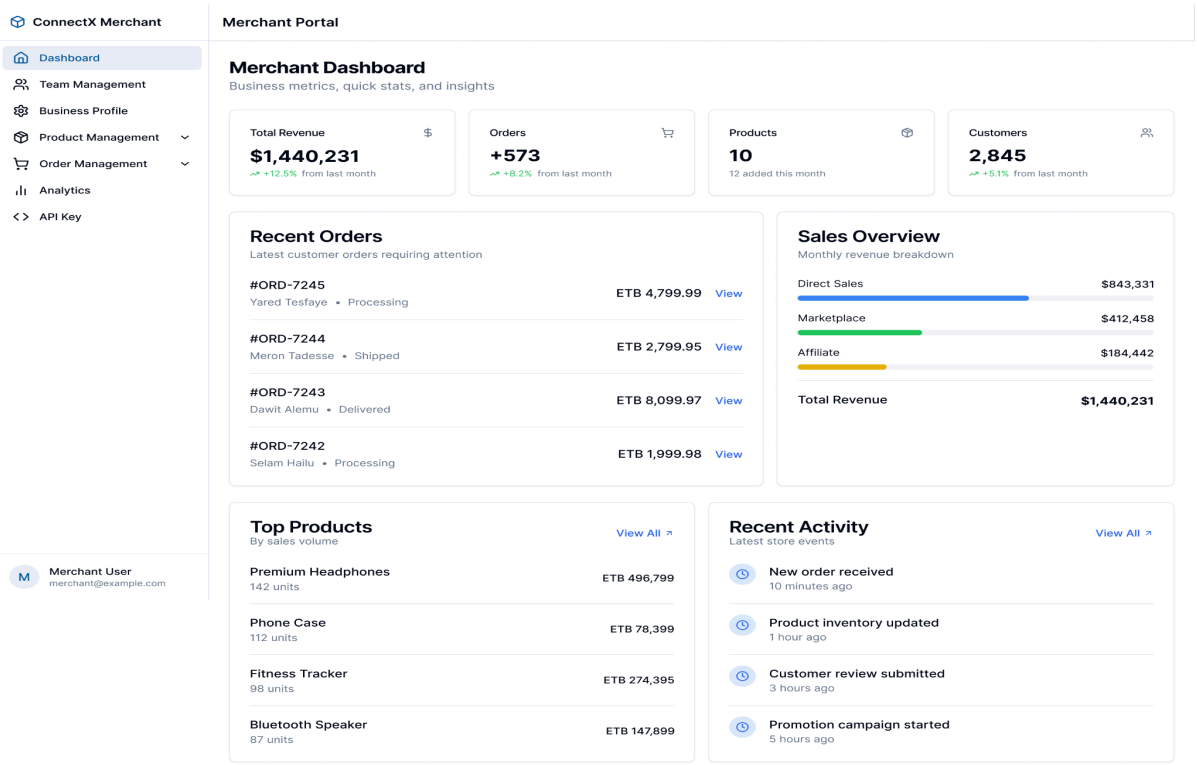


Figure 14: Merchant dashboard Snapshot

Landing page interface

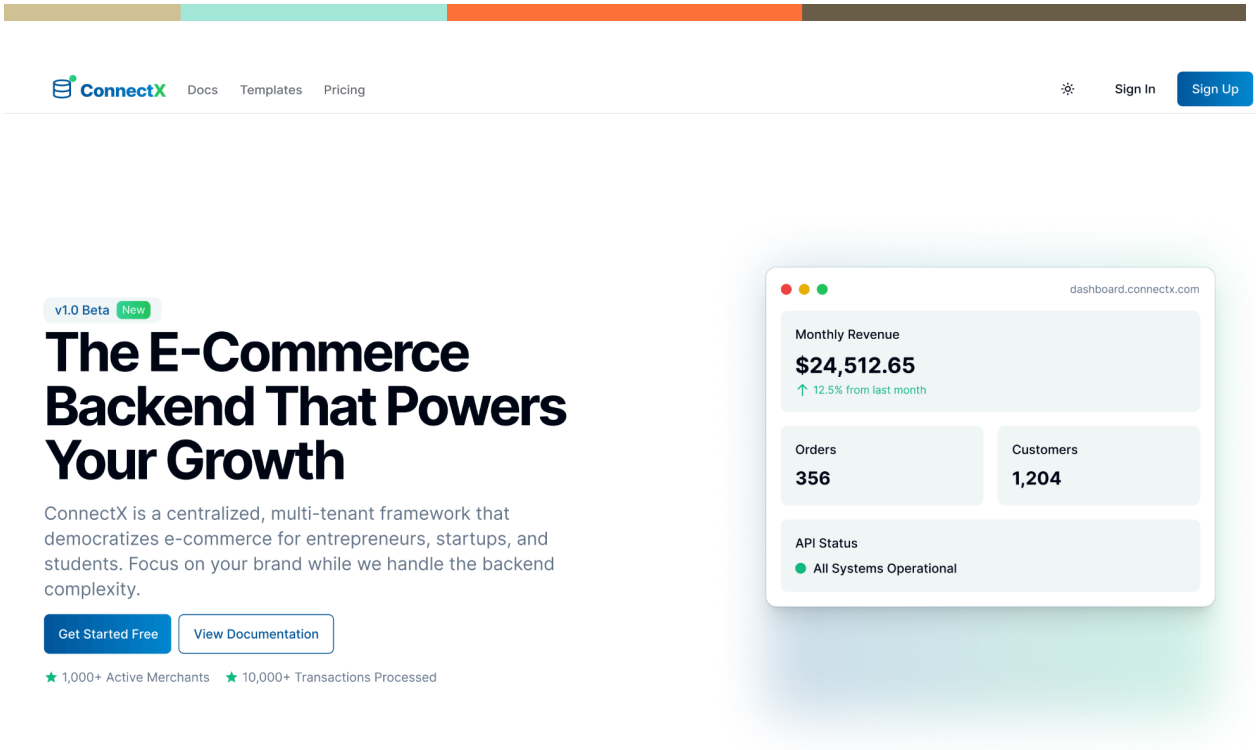


Figure 15: Landing page snapshot

Documentation Interface

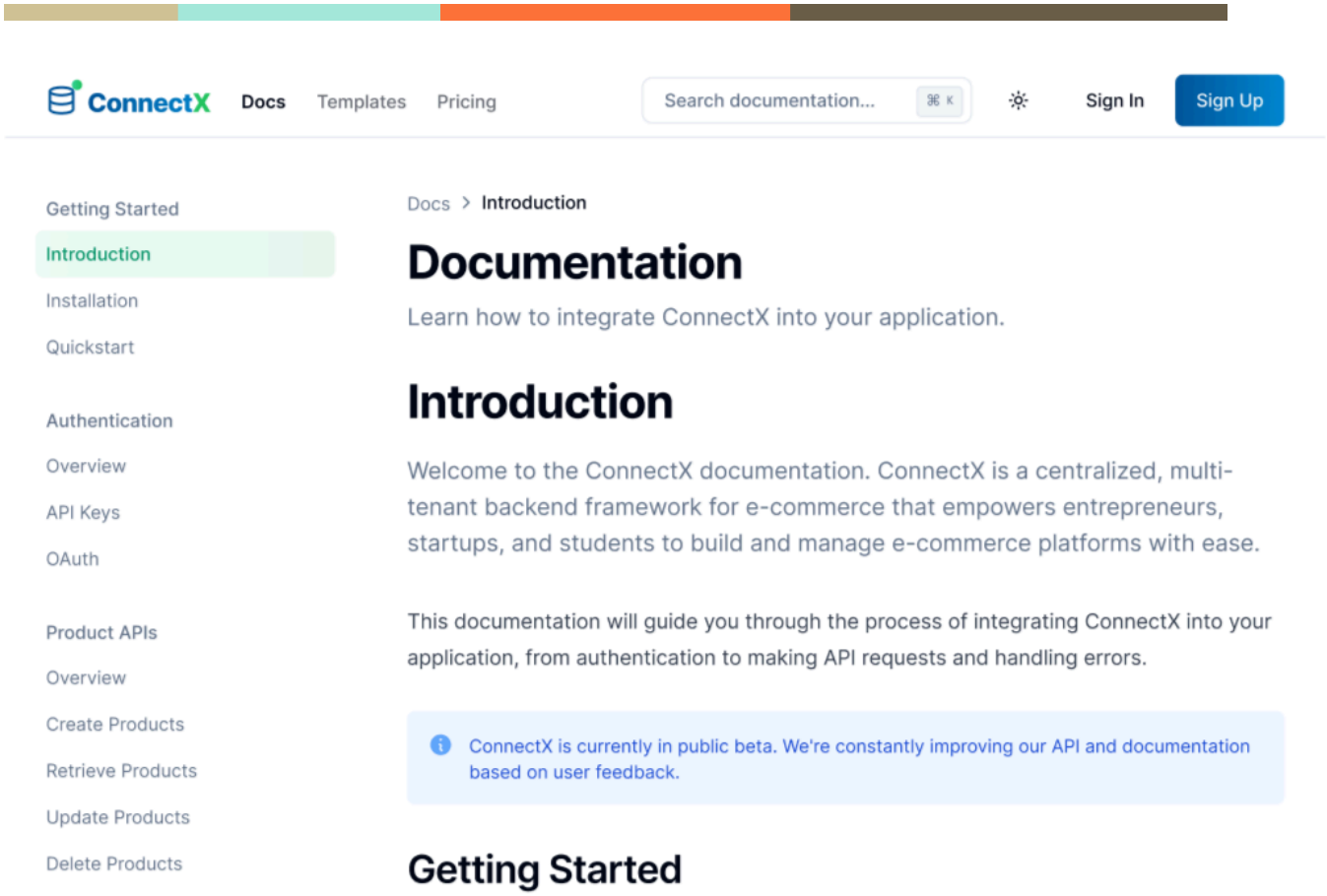


Figure 16: ConnectX Documentation page snapshot

★ The user interface design including all the sections of landing-page,admin interface and merchant portal with mockups, or prototypes to visualize the UI's layout and interaction design can be found with this Figma Link.

5.3.4. Integration

The integration phase of the ConnectX platform ensures a cohesive e-commerce experience by connecting key components. Below is a concise overview of the integration process.

1. Frontend-Backend Integration

- **API Integration**: Utilized Django REST Framework for RESTful endpoints, consumed by Next.js frontend via Fetch API.
- **Authentication**: Implemented JWT-based authentication with secure token storage in HTTP-only cookies, automatic token refresh, and role-based access control.
- **Endpoints**: Organized by domains including products (/api/products/), orders (/api/orders/), users (/api/users/), payments (/api/payments/), and analytics (/api/analytics/).

2. Database Integration

- **PostgreSQL**: Configured with Django ORM, managed schemas via migrations, and optimized queries with connection pooling.
- **Models**: Encompass products, categories, orders, payments, user management, and analytics.
- **Data Flow**: Employed WebSocket for real-time updates, caching for frequent data access, and background tasks for processing.

3. Payment Gateway Integration

- **Chapa Integration**: Implemented secure payment endpoints, webhook handling, error retries, and transaction logging.
- **Payment Flow**: Includes order creation, payment initialization, gateway redirection, confirmation, and order status updates.

4. Development and Deployment Integration

- **CI/CD Pipeline**: Used **GitHub Actions** for automated testing, code quality checks, and deployment.
- **Platforms**:
    - **Vercel**: Handled frontend deployment, environment variables, build optimization, and CDN integration.
    - **Render**: Managed backend deployment, database, SSL/TLS, and scaling.

5. Testing Integration

- **End-to-End Testing**: Conducted with Playwright for user journeys, API, payment flows, and cross-browser compatibility.
- **Test Scenarios**: Covered registration, authentication, product browsing, cart management, checkout, payment, order management, and admin operations.

6. Security Integration

- **API Security**: Implemented rate limiting, CORS, input validation, and SQL injection prevention.
- **Data Protection**: Used encryption at rest and secure transmission.

7. Monitoring and Logging Integration

- **Monitoring**: Tracked response times, error rates, resource utilization, and API health.
- **Logging**: Centralized error, access, audit, and performance logs.

8. Challenges and Solutions

- **Data Consistency**: Added transaction management, validation layers, and rollback

mechanisms.

- **Performance**: Optimized with caching, query improvements, and request batching.
- **Security**: Enhanced with comprehensive measures, input sanitization, and secure channels.

This integration ensures a secure, scalable, and user-friendly e-commerce platform meeting modern web standards.

Chapter Six: System Evaluation

6.1 Preparing Sample Test Plans

Software testing is an important step in making sure a software product works as expected and meets its original goals. It helps catch bugs and issues early by running different parts of the system and checking how they behave. Tools are often used to automate this process and check things like how the system performs or if all features work correctly. The main purpose is to spot any problems or differences between how the system works and what was planned. In this section, we explain how the test plans were prepared to check the ConnectX system, which brings together both frontend and backend parts to manage user actions and data.

Methodology

1. **Defining Test Cases:**
    - **Requirement Analysis:** System requirements were reviewed to pinpoint critical functionalities, such as user authentication, page rendering, and API operations.
    - **Test Case Design:** Specific test cases were crafted for each functionality, including:
        - **Input Conditions:** Diverse inputs, such as valid and invalid user credentials or API payloads.
        - **Execution Steps:** Detailed procedures to run each test case.
        - **Expected Results:** Predicted outcomes, like successful logins or error messages for invalid inputs.
        - **Edge Cases:** Scenarios where the system might fail, such as unauthorized access or large data inputs.
    - **Examples of Test Cases:**
        - **Frontend (JestJS):** Testing user authentication (login, signup), page loading (home, merchant, admin), and component behavior (protected routes).
        - **Frontend (Playwright):** Verifying end-to-end workflows, including login form display, invalid login handling, signup navigation, and home page functionality.

        - **Backend (Pytest):** Testing user API endpoints for listing, creating, updating, and deleting users, as well as authentication and profile management.
2. **Criteria for Success:**

    - **Functionality:** Each function must produce the correct output for given inputs.
    - **Performance:** The system should respond within acceptable time limits under normal and peak conditions.
    - **Reliability:** Consistent performance across multiple test runs is expected.
    - **Usability:** Clear, user-friendly error messages and guidance should be provided for invalid operations.
3. **Tools and Techniques:**

    - **JestJS:** Employed for unit testing of frontend React components.
        - **Assertions:** To verify conditions (e.g., component renders correctly).
        - **Mocking:** To simulate functions and modules.
        - **Snapshots:** To capture UI component states for consistency checks.
        - **Code Coverage:** To ensure all code paths are tested.
    - **Playwright:** Used for end-to-end testing of frontend workflows.
        - **Cross-Browser Testing:** Ensures compatibility across browsers like Chromium.
        - **Headless Mode:** Enables faster test execution.
        - **Automated UI Testing:** Simulates user interactions to validate UI functionality.
    - **Pytest:** Utilized for backend API testing.
        - Supports comprehensive test case execution for API endpoints, with verbose output for debugging.

6.2 Evaluating the Proposed Design and Solutions

This section details the evaluation process for the ConnectX system, focusing on executing the test plans, monitoring system behavior, and documenting results. The evaluation covered both frontend and backend components to ensure a cohesive system performance.

Execution of Test Plans

Procedure:

- **Setup:** The test environment was configured to mirror the production setup, using local development servers and virtual environments.
- **Execute Test Cases:** Tests were run using JestJS for frontend unit and integration tests, Playwright for end-to-end tests, and Pytest for backend API tests.

- **Monitor Behavior:** System behavior was observed during test execution, with attention to anomalies or unexpected outcomes.
- **Collect Data:** Performance metrics, such as execution times, were recorded for analysis.

Documentation:

The test results are presented in tables below, organized by the testing framework. Each table reflects the outcomes observed in the provided images, ensuring alignment with the actual test data.

Tests Executed with JestJS

The Jest test run, shown in Figure 15, evaluated frontend components and pages, including protected routes, merchant, home, admin, login, and sign up functionalities.

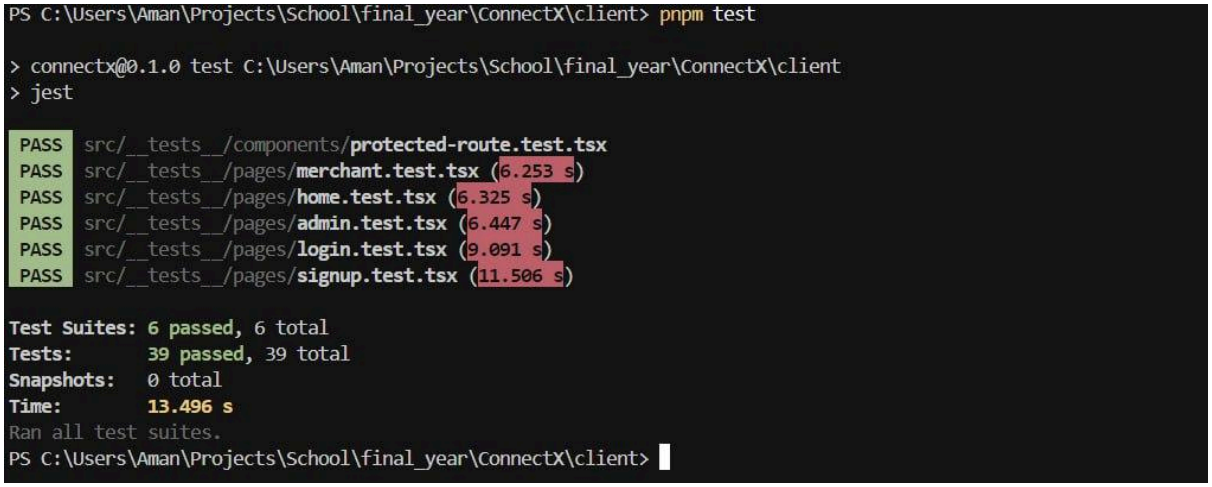


Figure 17: Jest Test Report

The screenshot indicates that all 39 tests across 6 test suites passed successfully, with a total execution time of 13.496 seconds. Due to the lack of individual test case details, we summarize the results as follows:

Total Tests with JestJS:

- **Test Suites:** 6 passed, 6 total
- **Tests:** 39 passed, 39 total
- **Snapshots:** 0 total
- **Time:** 13.496 s

The test suites covered:

- protected-route.test.tsx
- merchant.test.tsx (6.253 s)
- home.test.tsx (6.325 s)

- admin.test.tsx (6.447 s)
- login.test.tsx (9.091 s)
- signup.test.tsx (11.566 s)

Tests Executed with Playwright

Figure 16 illustrates the Playwright test run, focusing on end-to-end testing of authentication and home page workflows.

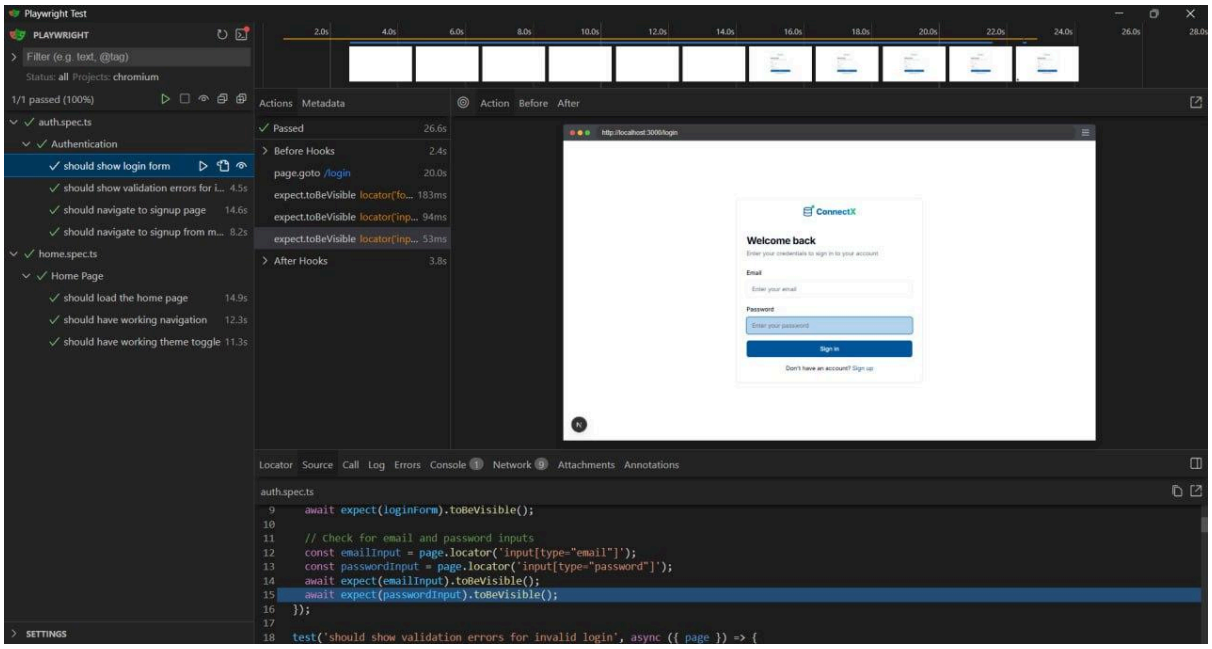


Figure 18: Playwright Test Report

The following table lists the test cases executed, derived from the image showing test cases under auth.spec.ts and home.spec.ts:

Table 17: Test Executed with PlayWright

Test Case ID	Test Case Description	Expected Result	Actual Result	Status
TC-P01	Verify that the login form is displayed	Login form is visible	Login form is visible	Pass
TC-P02	Verify that validation errors are shown for invalid login	Error message is displayed	Error message is displayed	Pass

TC-P03	Verify navigation to signup page	User is redirected to signup page	User is redirected to signup page	Pass
TC-P04	Verify navigation to sign up from menu	User is redirected to sign up from menu	User is redirected to sign up from menu	Pass
TC-P05	Verify that the home page loads correctly	Home page is loaded	Home page is loaded	Pass
TC-P06	Verify that navigation works on home page	Navigation links work	Navigation links work	Pass
TC-P07	Verify that theme toggle works	Theme changes when toggled	Theme changes when toggled	Pass

The test durations ranged from 1.4 seconds (validation errors) to 14.9 seconds (home page load), indicating acceptable performance for end-to-end workflows.

Tests Executed with Pytest

Figure 17 displays the Pytest test run for the backend user API, executed on a Linux system with Python 3.12.3.

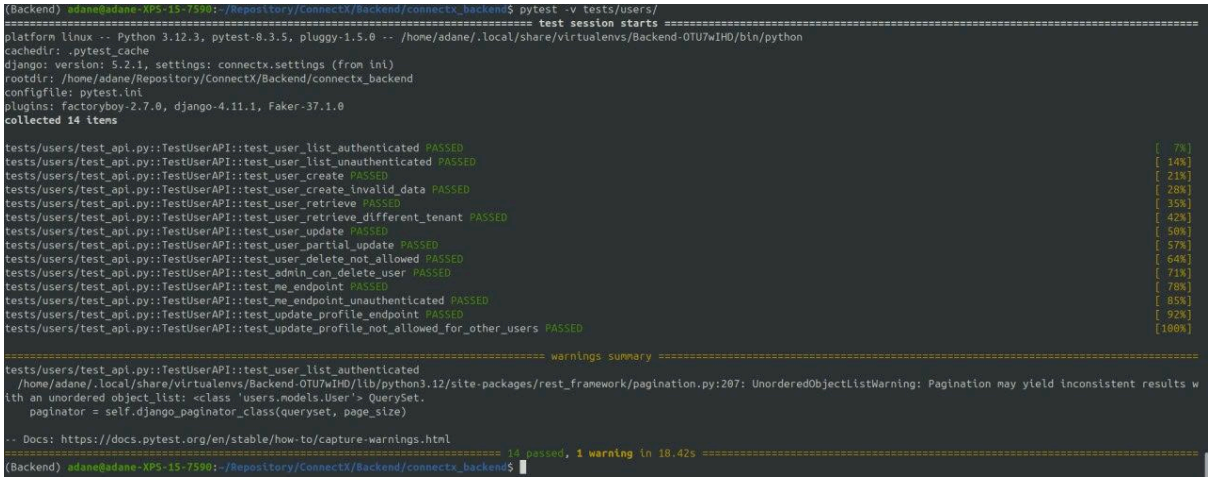


Figure 19: Pytest Test Report

The following table lists the 14 test cases executed, all of which passed, with a total execution time of 18.42 seconds:

Table 18: Test Executed with Pytest

Test Case ID	Test Case Description	Expected Outcome	Actual Outcome	Pass/Fail
TC-B01	Test user list authenticated	Authenticated user can list users	Users are listed	Pass
TC-B02	Test user list unauthenticated	Unauthenticated user cannot list users	Access denied	Pass
TC-B03	Test user create invalid data	Creating user with invalid data fails	Creation fails with error	Pass
TC-B04	Test user create	Creating user with valid data succeeds	User is created	Pass
TC-B05	Test user retrieve	Retrieving user details succeeds	User details are retrieved	Pass
TC-B06	Test user retrieve different tenant	Retrieving user from different tenant fails	Access denied	Pass
TC-B07	Test user update	Updating user succeeds	User is updated	Pass
TC-B08	Test user partial update	Partially updating user succeeds	User is partially updated	Pass
TC-B09	Test user delete not allowed	Non-admin cannot delete user	Deletion fails	Pass
TC-B10	Test admin can delete user	Admin can delete user	User is deleted	Pass
TC-B11	Test me endpoint	Authenticated user can access /me endpoint	/me endpoint returns user data	Pass
TC-B12	Test me endpoint unauthenticated	Unauthenticated user cannot access /me endpoint	Access denied	Pass
TC-B13	Test update profile not allowed for others	User cannot update another user's profile	Update fails	Pass

TC-B14	Test update profile	User can update own profile	Profile is updated	Pass

A warning was noted regarding potential pagination inconsistencies due to unordered object lists, which requires further investigation.

6.3 Discussing the Results

Analysis of Results

- **Criteria Met:**
    - **Functionality:** All tested functionalities, including frontend rendering, end-to-end workflows, and backend API operations, met the expected outcomes.
    - **Performance:** Execution times were within acceptable limits, with Jest tests completing in 13.496 seconds, Playwright tests ranging from 1.4 to 14.9 seconds, and Pytest tests totaling 18.42 seconds.
    - **Reliability:** The system performed consistently, with all 39 Jest tests, 7 Playwright tests, and 14 Pytest tests passing without failures.
    - **Usability:** Test cases like "verify validation errors for invalid login" suggest that the system provides clear error messages, though direct usability testing was not conducted.
- **Issues and Defects:**
    - No critical defects were identified, as all tests passed.
    - A single warning in the Pytest results highlighted potential inconsistencies in pagination due to unordered object lists, which could affect data retrieval reliability.
- **Implications:**
    - The system demonstrates strong functionality and reliability, making it suitable for its intended use.
    - The pagination warning indicates a need for backend optimization to ensure consistent data handling, particularly for large datasets.

Recommendations for Improvements

1. **Address Pagination Warning:**
    - Review the Django documentation on capturing warnings and implement ordered object lists to prevent pagination inconsistencies.

2. **Performance Optimization:**

    - Conduct load testing to assess system performance under high user traffic.
    - Optimize test execution times, particularly for slower Playwright tests (e.g., 14.9 seconds for home page load).
3. **Further Testing:**

    - Expand test coverage to include more edge cases, such as complex user interactions or extreme data inputs.
    - Implement automated regression testing to detect issues introduced by code changes.
    - Perform cross-browser testing beyond Chromium to ensure broader compatibility.
4. **Usability Enhancements:**

    - Collect user feedback to validate the clarity and helpfulness of error messages.
    - Standardize error message formats and provide actionable guidance to improve user experience.

Using JestJS, Playwright, and Pytest, we thoroughly tested the ConnectX system to make sure it's ready for deployment. While everything is working well, fixing the pagination warning and applying a few suggested improvements will make the platform even more reliable and user-friendly, helping us deliver a smoother and more enjoyable experience for users.

Chapter Seven: Conclusions and Recommendations

7.1 Conclusion of study

The ConnectX platform represents a significant step forward in democratizing access to scalable e-commerce infrastructure, offering a comprehensive backend solution that addresses the common limitations faced by small businesses, startups, and student developers. By utilizing a multi-tenant architecture and an API-first design approach, ConnectX simplifies the development and deployment of e-commerce platforms, enabling users to overcome high development costs, limited technical expertise, and scalability challenges.

Through its centralized backend services—including order processing, user management, and dual product listing—the system fosters collaboration between administrators and merchants while promoting resource sharing and operational efficiency. Its flexibility allows it to serve various market niches such as fashion, electronics, and more, without sacrificing security, reliability, or performance.

One of the core strengths of ConnectX lies in its ability to accelerate time-to-market and reduce financial and technical barriers for new entrants. By offloading backend complexities, the platform empowers users to focus on frontend innovation, branding, and customer experience. Additionally, ConnectX supports growing user and transaction volumes, ensuring long-term scalability and

business continuity.

In conclusion, ConnectX stands as a transformative solution for e-commerce development. It lowers entry barriers, encourages entrepreneurial innovation, and provides a solid foundation for building customized, secure, and scalable digital commerce solutions. Embracing such a platform will not only enhance current e-commerce capabilities but also pave the way for a more inclusive and collaborative digital marketplace.

7.2 Recommendations of the Study

As we move forward, it is essential to continue refining and evolving the ConnectX platform by incorporating emerging technologies and global best practices in e-commerce. This will ensure that the system remains competitive, user-friendly, and aligned with the growing demands of digital commerce.

Some of the recommendations we give forward are:

Performance Optimization and Security Enhancements:
Improve system performance by optimizing PostgreSQL queries and introducing caching mechanisms such as Redis or Memcached. This will reduce database load and ensure smooth operation under high traffic. Additionally, strengthen platform security through multi-factor authentication (MFA), end-to-end encryption, and compliance with global data protection standards like GDPR and PCI DSS.

Architectural Evolution:
Begin transitioning the system from a modular monolithic structure to a microservices-based architecture. This approach will support better scalability, resilience, and ease of deployment, especially as the platform scales to accommodate larger user bases and diverse business requirements.

Feature Expansion:
Integrate a real-time analytics dashboard that offers merchants insights into customer behavior, sales trends, and inventory levels. Leverage AI to deliver personalized product recommendations and dynamic pricing strategies, enhancing user engagement and conversion rates. Additionally, expand the platform's payment capabilities to include more gateways, enabling merchants to serve international markets.

Research and Development:

Allocate resources for ongoing research and development to explore new opportunities for innovation in e-commerce infrastructure. Encourage collaboration with academic researchers, cloud computing experts, and industry professionals to stay at the forefront of scalable multi-tenant systems and digital commerce trends. Invest in research initiatives that assess the impact of the

ConnectX platform on merchant growth, development speed, cost efficiency, and customer experience. Additionally, explore emerging technologies such as serverless computing, headless commerce, and AI-driven analytics to continuously evolve the platform and enhance its value to users.

By implementing these recommendations, we can ensure the successful deployment, adoption, and long-term sustainability of the ConnectX platform. Through continued collaboration with developers, merchants, and industry stakeholders, ConnectX has the potential to become a cornerstone of the modern e-commerce ecosystem—empowering innovation, expanding opportunities, and shaping the future of digital commerce.

Reference

[1] J. Smith, R. Brown, and L. Davis, "Advances in Microservices for E-Commerce Systems," Journal of Software Engineering, vol. 18, no. 3, pp. 45-60, 2021.
[2] M. Zhao, X. Li, and Y. Wang, "Serverless Computing in E-Commerce Applications," Computational Advances, vol. 27, no. 4, pp. 112-125, 2022.
[3] A. Green, "Leveraging Django for Scalable E-Commerce Projects," Python Development Quarterly, vol. 16, no. 2, pp. 78-89, 2022.
[4] T. Nguyen, R. Lopez, and H. Chen, "Enhancing Backend Scalability with Containerization," Journal of Systems Engineering, vol. 10, no. 3, pp. 98-109, 2022.
[5] P. Li, Q. Zhang, and Y. Xu, "Advances in Privacy-Preserving Techniques for E-Commerce," Data Security Journal, vol. 29, no. 3, pp. 89-105, 2021.
[6] Knowl.io. (2024, November 20). API Guides 2024: What is API-Driven Architecture? [Online]. Available: https://www.knowl.ai/blog/api-guides-2024-what-is-api-driven-architecture-cltvc40r0004bn2dicihdji3s (Accessed: Nov. 20, 2024).
[7] Reducing Costs in E-Commerce Startups, Entrepreneur Magazine, 2023. [Online]. Available: https://www.entrepreneur.com/growing-a-business/how-to-make-at-least-1000-your-first-month-of-ecom merce/287828. [Accessed: Nov. 16, 2024].
[8] OpenAI, "ChatGPT." [Online]. Available: https://chat.openai.com. [Accessed: Nov. 16, 2024].
[9] Stripe, Stripe Documentation: Payments Infrastructure for the Internet, 2024. [Online]. Available: https://stripe.com/docs. [Accessed: Nov. 16, 2024].
[10] Wikipedia, Snowflake: The Data Cloud, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Snowflake_Inc. [Accessed: Nov. 16, 2024].
[11] Jellyfish Technologies. (2024, November 20). The Cost of Developing an E-commerce Website in 2024 [Online]. Available: https://www.jellyfishtechnologies.com/ecommerce-website-cost/ [Accessed: November 20, 2024].

[12] Medusa.js – Open Source Headless Commerce. [Online]. Available: https://medusajs.com

[13] Vendure – Modern Headless GraphQL Commerce. [Online]. Available: https://www.vendure.io

[14] Commerce Layer – Headless Commerce APIs. [Online]. Available: https://commercelayer.io