Databases II
Semester 2025-III
Workshop No. 3 — Data System Architecture and Information Retrieval
ArtisanNexus

Eng. Jorge Andrés Quiceno S.[1], Eng. Laurent David Chaverra Córdoba[2] and Eng. Juan David Olmos Corredor[3]

November 22, 2025

Contents

1 Introduction

The ArtisanNexus project proposes the design and development of a high-performance, resilient *Multi-Vendor Marketplace Platform.* The strategic vision is to establish ArtisanNexus as the authoritative digital nexus for unique, customized, and artisanal goods, enabling seamless global commerce between independent creators (Vendors) and discerning consumers (Customers).

The platform manages the entire e-commerce process, from importing and standardizing vendor product data to processing high-volume payments and generating actionable business intelligence. The objective of this architecture is to eliminate the bottlenecks typically associated with monolithic e-commerce architectures by adopting a distributed, data-centric framework.

The core problem addressed by ArtisanNexus lies in the systemic constraints imposed by the multi-vendor paradigm, to handle two primary challenges simultaneously:

1. **Data Heterogeneity and Complexity:** Unlike single-brand e-commerce, the product data ingested from multiple independent vendors lacks a uniform schema. Efficiently storing, retrieving, and searching this dynamic, unstructured data demands flexibility that exceeds the capabilities of a pure Relational Database Management System (RDBMS).
2. **Distributed Transactional Integrity:** The capability for a Customer to execute an atomic checkout involving items from multiple distinct Vendors introduces significant *concurrency control* challenges. Ensuring that inventory reservation, payment authorization, and order finalization succeed or fail as a singular, consistent unit—thereby upholding *ACID* (Atomicity, Consistency, Isolation, Durability)—is critical to preventing organizational revenue leakage and preserving customer trust.

To overcome these constraints, the project is structured around three primary strategic goals:

- **Operational Excellence:** To achieve and maintain a high-availability (HA) operational posture with minimal latency for high-volume, customer-facing interactions.
- **Knowledge Maximization:** To transform raw user activity (clickstreams, purchase history) and sales data into actionable business intelligence (BI) for both platform administrators and Vendors.
- **System Resilience and Scalability:** To engineer a system capable of scaling elastically to accommodate exponential growth in both user count and transactional throughput via database partitioning and distributed architectures.

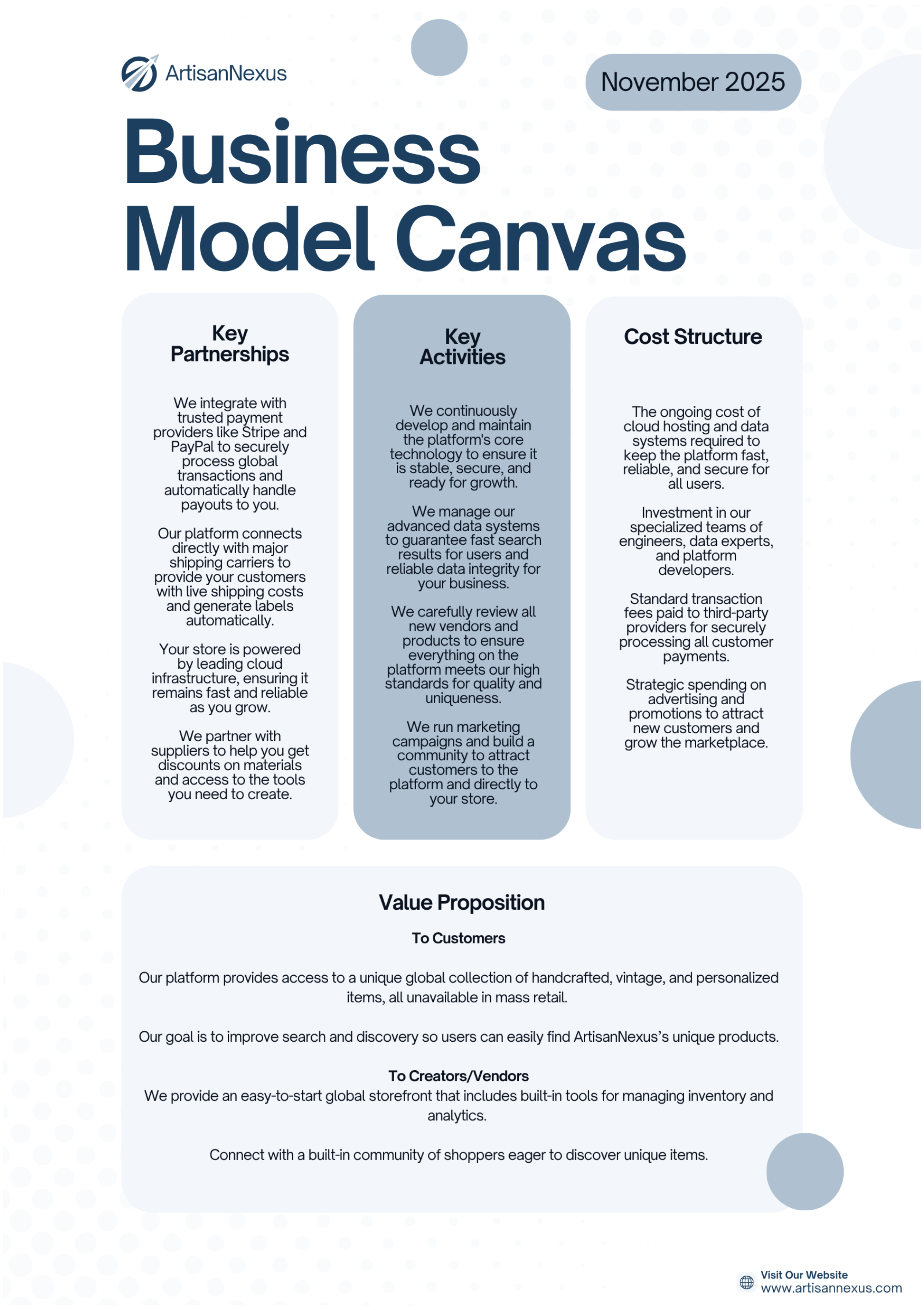
To meet the strategic goals, the system design will leverage a *Microservices Architecture* built on the robust *Java/Spring Boot* framework for the backend, supported by an *Angular* front-end. The core technological objective is the implementation of a *Polyglot Persistence* strategy, which dictates selecting the optimal database technology for each specific service's workload.

This approach directly satisfies the rigorous requirements for modern data-intensive applications:

- **Fast Query Execution** is achieved by dedicating an *Inverted Index NoSQL* solution for search operations, decoupling it from the transactional core.
- **Constant Data Ingestion** and inter-service communication are managed by a *Distributed Streaming Platform*, which processes event data in real-time.

2 Business Model Canvas

The Business Model Canvas describes the value proposition, customer segments, channels, revenue streams, and other key aspects of the application.



ArtisanNexus

November 2025

Business Model Canvas

Key Partnerships

We integrate with trusted payment providers like Stripe and PayPal to securely process global transactions and automatically handle payouts to you.

Our platform connects directly with major shipping carriers to provide your customers with live shipping costs and generate labels automatically.

Your store is powered by leading cloud infrastructure, ensuring it remains fast and reliable as you grow.

We partner with suppliers to help you get discounts on materials and access to the tools you need to create.

Key Activities

We continuously develop and maintain the platform's core technology to ensure it is stable, secure, and ready for growth.

We manage our advanced data systems to guarantee fast search results for users and reliable data integrity for your business.

We carefully review all new vendors and products to ensure everything on the platform meets our high standards for quality and uniqueness.

We run marketing campaigns and build a community to attract customers to the platform and directly to your store.

Cost Structure

The ongoing cost of cloud hosting and data systems required to keep the platform fast, reliable, and secure for all users.

Investment in our specialized teams of engineers, data experts, and platform developers.

Standard transaction fees paid to third-party providers for securely processing all customer payments.

Strategic spending on advertising and promotions to attract new customers and grow the marketplace.

Value Proposition

To Customers

Our platform provides access to a unique global collection of handcrafted, vintage, and personalized items, all unavailable in mass retail.

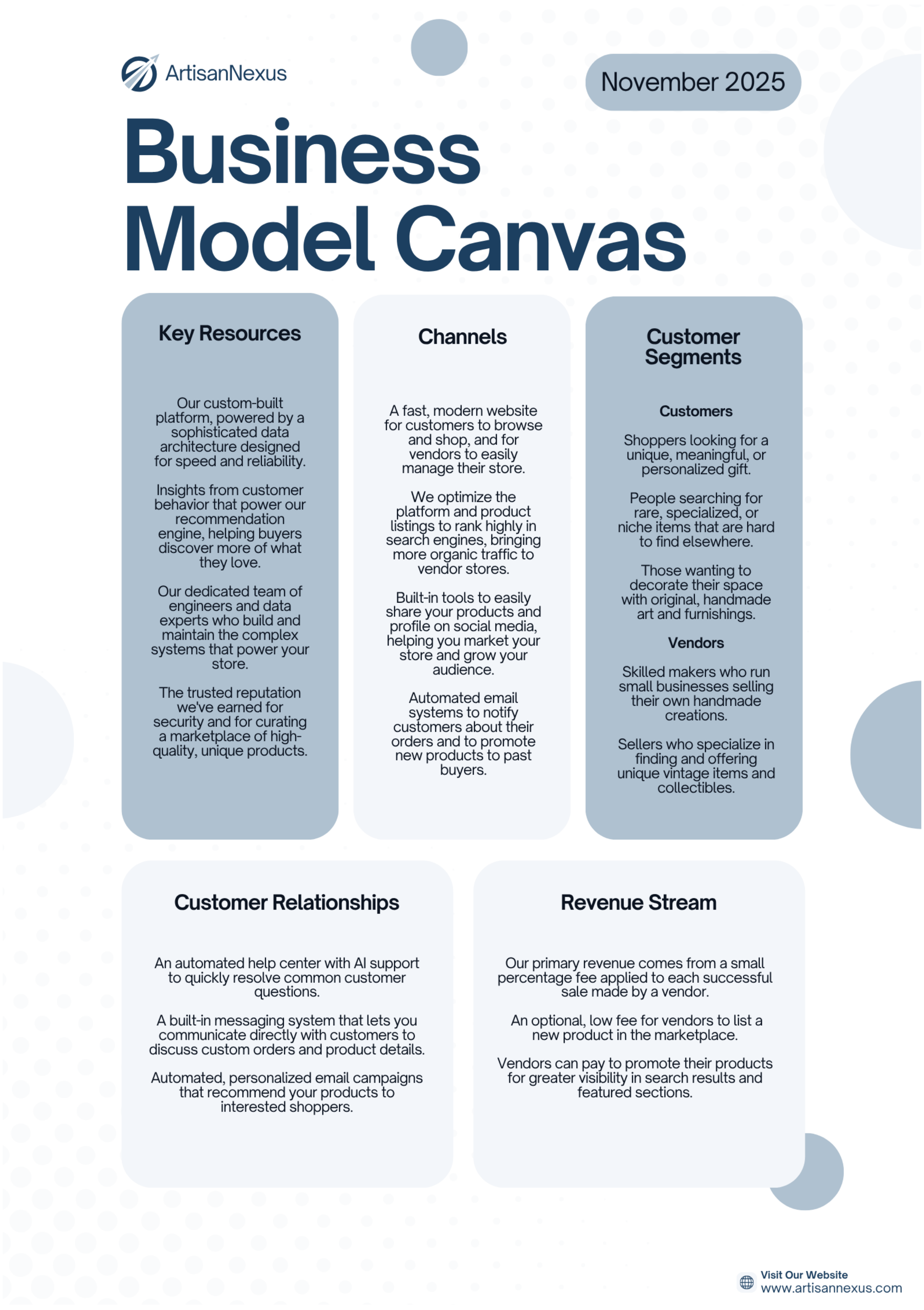
Our goal is to improve search and discovery so users can easily find ArtisanNexus's unique products.

To Creators/Vendors

We provide an easy-to-start global storefront that includes built-in tools for managing inventory and analytics.

Connect with a built-in community of shoppers eager to discover unique items.

Visit Our Website
www.artisannexus.com



ArtisanNexus

November 2025

Business Model Canvas

Key Resources

Our custom-built platform, powered by a sophisticated data architecture designed for speed and reliability.

Insights from customer behavior that power our recommendation engine, helping buyers discover more of what they love.

Our dedicated team of engineers and data experts who build and maintain the complex systems that power your store.

The trusted reputation we've earned for security and for curating a marketplace of high-quality, unique products.

Channels

A fast, modern website for customers to browse and shop, and for vendors to easily manage their store.

We optimize the platform and product listings to rank highly in search engines, bringing more organic traffic to vendor stores.

Built-in tools to easily share your products and profile on social media, helping you market your store and grow your audience.

Automated email systems to notify customers about their orders and to promote new products to past buyers.

Customer Segments

Customers

Shoppers looking for a unique, meaningful, or personalized gift.

People searching for rare, specialized, or niche items that are hard to find elsewhere.

Those wanting to decorate their space with original, handmade art and furnishings.

Vendors

Skilled makers who run small businesses selling their own handmade creations.

Sellers who specialize in finding and offering unique vintage items and collectibles.

Customer Relationships

An automated help center with AI support to quickly resolve common customer questions.

A built-in messaging system that lets you communicate directly with customers to discuss custom orders and product details.

Automated, personalized email campaigns that recommend your products to interested shoppers.

Revenue Stream

Our primary revenue comes from a small percentage fee applied to each successful sale made by a vendor.

An optional, low fee for vendors to list a new product in the marketplace.

Vendors can pay to promote their products for greater visibility in search results and featured sections.

Visit Our Website
www.artisannexus.com

3 Requirements

This section specifies the comprehensive requirements for the *ArtisanNexus* Multi-Vendor Marketplace, organized by functional domain and quality attributes, ensuring alignment with advanced database principles (ACID, Polyglot Persistence, Distributed Systems).

3.1 Functional Requirements (FR)

Functional requirements detail the specific behaviors and services the system must provide to its stakeholders (Customer, Vendor, Admin). Each requirement is designed to be independent and clarify system accountability.

- **FR 1.1: Secure User Authentication and Authorization.**
  Requirement: The system shall allow secure registration, login, and identity verification, enforcing role-based access for Customer, Vendor, and Admin roles.
  Role Association: All Users.

- **FR 2.1: Product Catalog Management (Catalog Service).**
  Requirement: The system shall allow a Vendor to create, update, and manage core product listings, vendor-defined attributes (e.g., size charts, material composition) within the Catalog Service.
  Role Association: Vendor.

- **FR 2.2: Real-Time Inventory Management and Concurrency Control.**
  Requirement: The system shall manage product inventory counts in a high-concurrency environment and utilize database mechanisms to prevent overselling of limited stock.
  Role Association: Vendor.

- **FR 3.1: High-Speed Faceted Search and Filtering.**
  Requirement: The system shall provide an advanced, high-speed **faceted search capability** across the entire product catalog, supporting full-text queries and filtering by category, price, vendor rating, and product attributes.
  Role Association: Customer.

- **FR 3.2: Atomic Multi-Vendor Transaction Execution.**
  Requirement: The system shall enable a Customer to perform a checkout for a cart containing items from multiple distinct vendors in a single transaction request.
  Role Association: Customer.

- **FR 3.3: Geospatial Product Discovery and Filtering.**
  Requirement: The system shall allow Customers to filter and sort products based on the Vendor's shipping origin (e.g., specific Spanish regions/Provinces) to optimize for shipping time and local commerce support.
  Role Association: Customer.

- **FR 4.1: Order Lifecycle Management.**
  Requirement: The system shall accurately manage the full lifecycle of a customer order, including cart persistence, checkout finalization, and displaying the order history.
  Role Association: Customer.

- **FR 5.1: Personalized Recommendation Generation.**
  Requirement: The system shall provide tailored product suggestions based on captured user activity and purchase history, supporting collaborative filtering logic.
  Role Association: System (Internal) / Customer.

- **FR 6.1: Granular Event Collection (Data Ingestion).**
  Requirement: The system shall securely and reliably capture and persist all raw user interactions (clickstreams, search terms, views), orders, and core transactions as events.
  Role Association: System (Internal).

- **FR 6.2: Asynchronous BI Data Pipeline.**
  Requirement: The system shall implement an asynchronous pipeline to transform operational data (Orders/Views) and load it into a dedicated structure (e.g., a MongoDB collection for analytics) to support the Admin Dashboard without impacting marketplace performance.
  Role Association: System (Internal).

- **FR 7.1: Real-Time Vendor Operational Dashboard.**
  Requirement: The system shall provide the Vendor with a real-time dashboard displaying key operational metrics, including inventory alerts and current month's sales transactions.
  Role Association: Vendor.

- **FR 7.2: Analytical Reporting for Business Intelligence (BI).**
  Requirement: The system shall enable an Administrator to generate comprehensive sales and performance reports, aggregated by product category, vendor region, and time period, based on historical data.
  Role Association: Administrator.

- **FR 8.1: Multi-Location Shipping Calculation.**
  Requirement: The system shall automatically calculate shipping costs based on the distance or region difference between the Vendor's registered location and the Customer's shipping address.
  Role Association: System (Internal).

3.2 Non-Functional Requirements (NFR)

Non-functional requirements specify quality attributes essential for the system's operational viability, scalability, and compliance, with a focus on distributed system characteristics.

- **NFR 1.0: High Availability and Fault Tolerance.**
  Requirement: The core transactional services must be highly available and resilient to failures using database replication. *Validation/Metrics:*
  - **Metric:** The application must recover from a simulated service crash (Docker container restart) within 30 seconds.
  - **Validation:** During the demo, kill the Product Service container and verify that the Front End displays a "Service Unavailable" friendly message rather than crashing, and recovers automatically upon container restart.

- **NFR 2.0: Horizontal Scalability and Partitioning.**

Requirement: The architecture must support scaling horizontally to accommodate growth in user base, products, and transactional volume. *Validation/Metrics:*

  - **Metric:** The backend architecture must support running 2+ instances of the Order Service simultaneously behind a load balancer.
  - **Validation:** Demonstrate via docker-compose up –scale order-service=2 that requests are distributed (logging the instance ID handling the request).

- **NFR 3.0: Data Consistency and Isolation (Concurrency Control).**
  Requirement: Strict transactional correctness must be maintained in the financial and inventory core to prevent anomalies like Lost Updates and Dirty Reads. *Validation/Metrics:*
  - **Metric:** Zero "Lost Updates" on inventory during concurrent purchases.
  - **Validation:** Use a JMeter script to simulate 10 concurrent users trying to buy the last 1 unit of a product. Exactly 1 transaction must succeed, and 9 must fail with an "Out of Stock" error.

- **NFR 4.0: Multi-Location Data Support.**
  Requirement: The system must logically partition data or support queries optimized for different regions. *Validation/Metrics:*
  - **Metric:** Database queries filtering by "Region" must use indexed fields to ensure execution time remains under 100ms.
  - **Validation:** Run EXPLAIN ANALYZE on a PostgreSQL query filtering products by vendor_region to prove index usage.

- **NFR 5.0: System Performance and Query Execution.**
  Requirement: Fast query execution for the product catalog and search results. *Validation/Metrics:*
  - **Metric:** 95% of read-only API calls (e.g., Product Search) must respond in under 300ms.
  - **Validation:** Measured using Postman Runner or JMeter with a seeded database of at least 1,000 products.

- **NFR 6.0: Security, Encryption, and Compliance.**
  Requirement: All sensitive data, including customer PII and payment references, must be encrypted both in transit and at rest. *Validation/Metrics:*
  - **Validation:** Inspect the database to verify that passwords are hashed (e.g., BCrypt) and not stored in plain text. Verify HTTPS (or simulated TLS) is required for login endpoints.

- **NFR 7.0: Maintainability and Observability.**
  Requirement: The system must provide comprehensive logging, metrics tracking, and alerting to enable clear diagnosis and support for operations teams. *Validation/Metrics:*
  - **Validation:** Integrate Spring Boot Actuator. Demonstrate a /health endpoint returning system status and show application logs appearing in a centralized console output.

- **NFR 8.0: Constant Data Ingestion Capacity.**
  Requirement: The system must handle a continuous flow of interaction events. *Validation/Metrics:*
  - **Metric:** The ingestion endpoint must accept 50 events/second without blocking the user interface.
  - **Validation:** Fire a batch of dummy "Page View" events to the ingestion API and verify that the HTTP response returns immediately (202 Accepted) while the database processes them in the background.

4 Improved User Stories

The user stories will describe the main use cases from the user's perspective, including the related actions, the necessary conditions to do these actions, and the expected results of the actions.

To improve the latest release, we rewrite the acceptance requirements in third-person, according to the review recommendations. Also, we aligned some user stories with the reviewed functional requirements, to not lose the link between both.

Title: **Register**	Priority: **High**	Estimation: **5 h**
User Story: As an End-User or Admin, the user wants to register in the system so that they can authenticate to access application resources and perform operations according to their permission level.		
Acceptance Criteria: • Given the user is on the registration page, when they submit a unique email, strong password, and full name, then the system creates the account in the PostgreSQL Identity tables and redirects them to the login page. • Given the user attempts to register with an existing email, when the form is submitted, then the system prevents creation and displays a "User already exists" error. • Given the user enters a password that violates the policy (minimum 8 characters, mixed case), when the form is submitted, then the system rejects the request with a specific validation error. • Given a successful registration, when the account is created, then the system assigns the default role (Customer or Vendor) to allow immediate distinct access control.		

Table 1: End User and Admin user story for register

Title: **Authentication**	Priority: **High**	Estimation: **5 h**
User Story: As an End-User or Admin, the user wants to authenticate against the system so that they can access resources corresponding to their role.		
Acceptance Criteria: • Given the user attempts login with valid credentials, when the system verifies the hash against the database, then it issues a secure session token (e.g., JWT) and redirects to the role-specific landing page. • Given the user fails authentication 3 consecutive times, when the threshold is reached, then the system temporarily locks the account or triggers a security alert. • Given the user requests a password reset, when the email is verified, then the system sends a recovery link. • **Technical Constraint**: Passwords must never be returned to API responses and must be encrypted at rest.		

Table 2: End User and Admin user story for authentication

Title: **Create Product**	Priority: **High**	Estimation: **4 h**
User Story: As a Supplier, the user wants to create new products with specific details so that they can add them to the catalog for customers to view.		
Acceptance Criteria: • Given the supplier is on the product creation page, when they submit mandatory fields (Name, Price, Stock) and variable attributes (Size, Material), then the system saves the structured data into the MongoDB Product Catalog. • Given the supplier enters Price or Stock, when the form is validated, then the system ensures values are non-negative integers before persistence. • Given the product is successfully saved, when the transaction completes, then the system triggers an event to the Search Index to ensure the product appears in search results within 2 seconds.		

Table 3: Supplier user story for creating a product

Title: **Edit Product**	Priority: **High**	Estimation: **3 h**
User Story: As a Supplier, the user wants to edit existing product details to correct errors or update pricing.		
Acceptance Criteria: • Given the supplier modifies a product description or price, when the form is saved, then the changes are immediately reflected in the Product Catalog (MongoDB). • Given the supplier changes the status to "Inactive", when the update is processed, then the system immediately removes the item from Customer-facing search results. • Given the supplier attempts to save invalid data, when validation fails, then the system returns an error highlighting the specific fields.		

Table 4: Supplier user story for editing a product

Title: **Supplier Profile & Location**	Priority: **High**	Estimation: **3 h**
User Story: As a Supplier, the user wants to personalize their profile and set their shipping origin so that customers know where products are shipping from.		
Acceptance Criteria: • Given the supplier is editing their profile, when they enter business details, then they must select a valid Region/Province (e.g., "Andalusia"). • Given the location is selected, when the profile is saved, then the system persists this location data in the Supplier's Relational record (PostgreSQL) to enable future geospatial filtering. • Given the supplier submits invalid location inputs, when the form is processed, then the backend validation logic rejects the update.		

Table 5: Supplier user story for profile and location management

Title: **Advanced Search & Filtering**	Priority: **High**	Estimation: **8 h**
User Story: As a Customer, the user wants to search for products using keywords and filters so that they can find specific items, including those from local artisans.		
Acceptance Criteria: • Given the user types a keyword, when the search is executed, then the system queries the Inverted Index (MongoDB) and returns results in under 300ms. • Given the user selects a "Region" filter (e.g., "Madrid"), when the results update, then the system filters the list to show only products linked to Vendors in that region. • Given the user applies multiple filters, when the query runs, then the system performs a faceted search by Category, Price Range, and Vendor Location simultaneously.		

Table 6: Customer user story for advanced and geospatial search

Title: **Personalized Recommendations**	Priority: **Medium**	Estimation: **6 h**
User Story: As a Customer, the user wants to see product suggestions based on their browsing history so that they can discover relevant items.		
Acceptance Criteria: • Given the user views the homepage, when the page loads, then the system calls the Python Recommendation Service to retrieve a list of "Recommended Products". • Given the user has a history of views and purchases, when recommendations are generated, then the results are based on collaborative filtering logic. • Given the user has no history (cold start), when the page loads, then the system defaults to displaying the top 5 "Best Selling" items globally.		

Table 7: Customer user story for personalized recommendations

Title: **Manage Shopping Cart**	Priority: **High**	Estimation: **3 h**
User Story: As a Customer, the user wants to add, view, and remove items in their shopping cart to review selections before purchase.		
Acceptance Criteria: • Given the user adds an item, when the action is triggered, then the system persists the cart state (MongoDB or Session) so it is retained if the page is refreshed. • Given the user attempts to add an item, when the request is received, then the system performs a preliminary check to ensure the requested quantity does not exceed current visible stock. • Given the user removes an item, when the update occurs, then the cart total price is immediately recalculated.		

Table 8: Customer user story for managing the shopping cart

Title: **Process Payment & Atomic Order**	Priority: **High**	Estimation: **6 h**
User Story: As a Customer, the user wants to securely pay for their cart items in a single transaction so that they can finalize the purchase.		
Acceptance Criteria: • Given the user confirms payment, when the system processes the request, then it must execute the Inventory Decrement, Order Creation, and Payment Recording as a single atomic unit in PostgreSQL. • Given the stock is insufficient due to a concurrent purchase, when the transaction attempts to commit, then the system must roll back the changes and display an "Out of Stock" error. • Given a successful order, when the transaction commits, then the system must asynchronously publish a "Sale Event" to the Data Ingestion Pipeline for BI processing.		

Table 9: Customer user story for atomic payment processing

Title: **BI Dashboard**	Priority: **Medium**	Estimation: **10 h**
User Story: As an Administrator, the user wants to view a dashboard of key business metrics to understand marketplace performance.		
Acceptance Criteria: • Given the Dashboard loads, when data is requested, then the system fetches pre-aggregated data from the Analytics Collection, ensuring no load is placed on the live transactional database. • Given the user views the "Sales by Region" chart, when it renders, then it must visualize which Spanish provinces are generating the most revenue. • Given the user views the "Top Products" section, when it updates, then it displays a list derived from historical sales data.		

Table 10: Admin user story for business intelligence dashboard

Title: **Real-Time Vendor Dashboard**	Priority: **Medium**	Estimation: **5 h**
User Story: As a Vendor, the user wants to see real-time alerts about low inventory and recent sales to manage their shop efficiently.		
Acceptance Criteria: • Given a product's stock drops below 5 units, when the dashboard refreshes, then the system flags the item with a low-inventory alert. • Given the user views the sales section, when the data loads, then the dashboard displays a list of the current month's transactions queried directly from the Order Service.		

Table 11: Vendor user story for operational dashboard

5 Database Architecture

Presentation of the initial database architecture for the project

5.1 High-Level Architecture

(This section has been improved with respect to the last version in Workshop 2, updating the diagram to clarify the information flow)

Our high level architecture is based in the three basic layers frontend, backend and data. the frontend will be the visual layer, the backend will work in an api structure, with some services exposed.

The data layer will be divided in 2 main parts, the operational data and the analytical data.

Operational data:

The Operational Data Layer serves as the core transactional storage of the e-commerce platform, supporting all real-time operations required by buyers and SME sellers. This layer is composed of two complementary data stores: a relational SQL database that manages structured and transactional information such as user profiles and orders, and a NoSQL document database, which is going to manage the semi-structured product catalog data, including product descriptions, images, attributes, and category hierarchies.

The reason to use these 2 databases is to allow fast reads and writes for day-to-day platform functionality such as searching products, placing orders, updating inventory, and managing seller catalogs. The operational layer is directly consumed by backend services, which are responsible for ensuring consistency, validation, and security of the data. Additionally, this layer acts as the source of truth for the ETL component, which periodically feeds the analytical storage.

Analytical data:

The Analytical Data Layer centralizes the information required for reporting, monitoring, and recommendation features. It is fed by an ETL component that periodically extracts data from the operational SQL and NoSQL stores, transforms it into clean, structured analytical formats. This layer is optimized for historical queries allowing the platform to generate metrics such as product popularity, sales performance, and user preferences without impacting real-time operations. By separating analytical processing from transactional activity, the layer provides the data foundation for features like dashboards, trend analysis, and the recommendation engine.

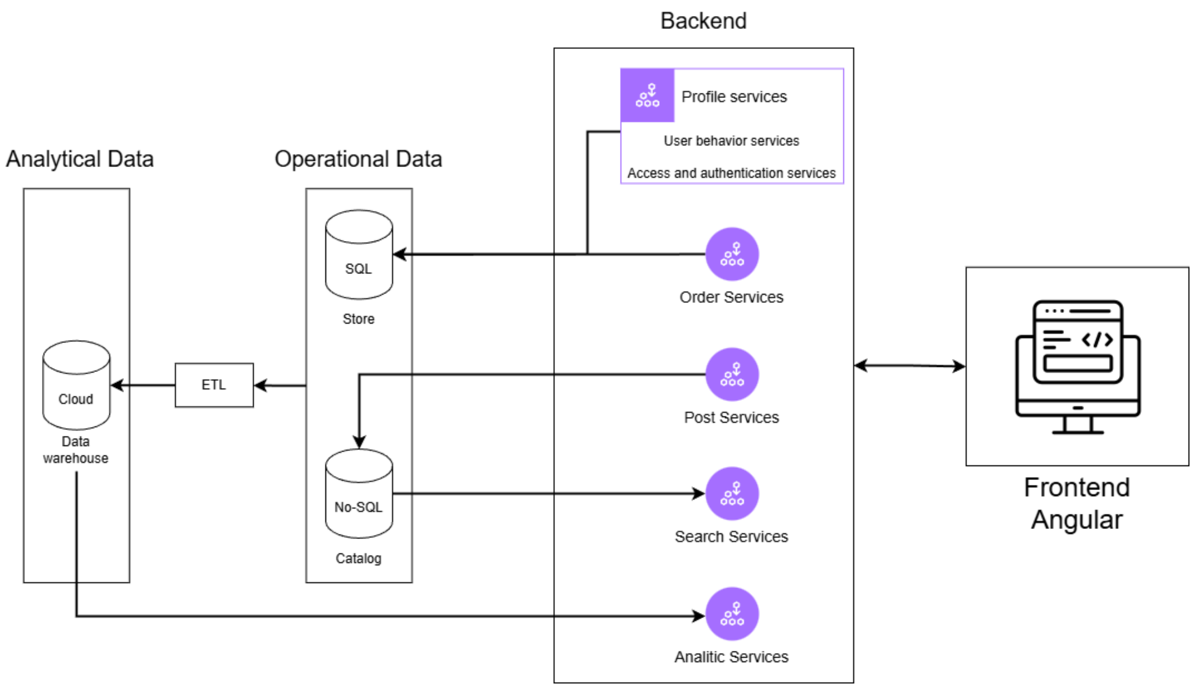


Figure 1: High Level DB Diagram

5.2 Data Flow

There are mainly 3 data flows between the layers presented

- Operational Dataflow: Describes how data moves when a buyer or SME uses the platform.

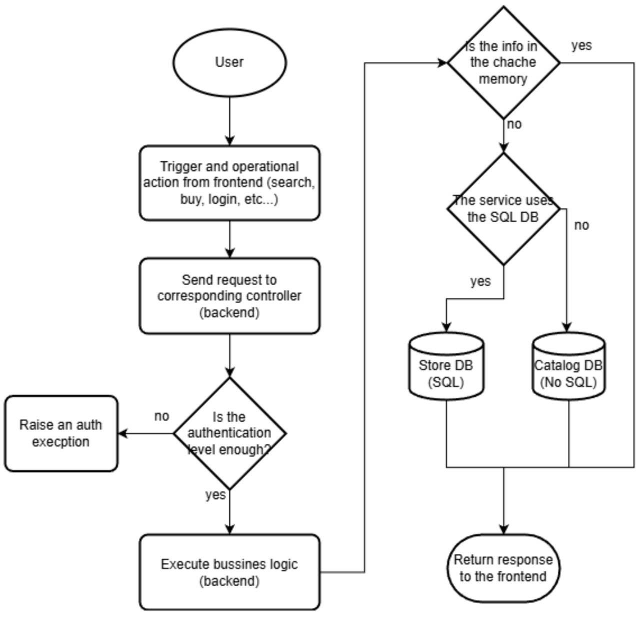

User
Trigger and operational action from frontend (search, buy, login, etc...)
Send request to corresponding controller (backend)
Is the authentication level enough?
no
Raise an auth execption
yes
Execute bussines logic (backend)
Is the info in the chache memory
yes
no
The service uses the SQL DB
yes
no
Store DB (SQL)
Catalog DB (No SQL)
Return response to the frontend

Figure 2: High Level DB Diagram

In the figure we can see the expected dataflow for the operational process which are: order, login, buy cart administration, profile settings, etc...

- ETL Dataflow: Shows how operational data is transformed for analytics.

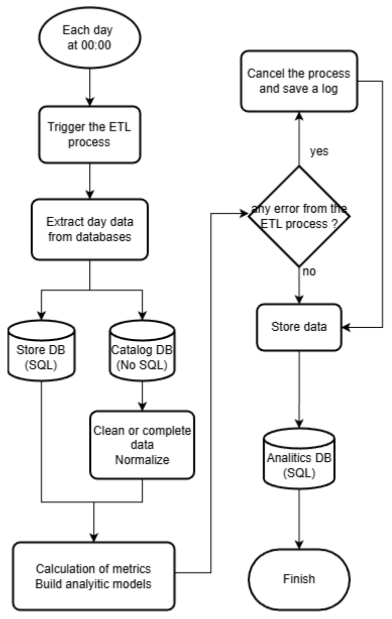

Each day at 00:00
Trigger the ETL process
Extract day data from databases
Store DB (SQL)
Catalog DB (No SQL)
Clean or complete data Normalize
Calculation of metrics Build analytic models
any error from the ETL process ?
yes
no
Cancel the process and save a log
Store data
Analitics DB (SQL)
Finish

Figure 3: High Level DB Diagram

In the figure we can see the expected dataflow for the ETL process, that is the process of processing the information from the operational databases to the analytic database, this process is going to be triggered every day at 00:00 am (midnight), and it gonna try to process the data of the day, if there is any problem it will stop the process and save the log instead.

- Reports Dataflow: Show how recommendations and reports are produced and delivered.

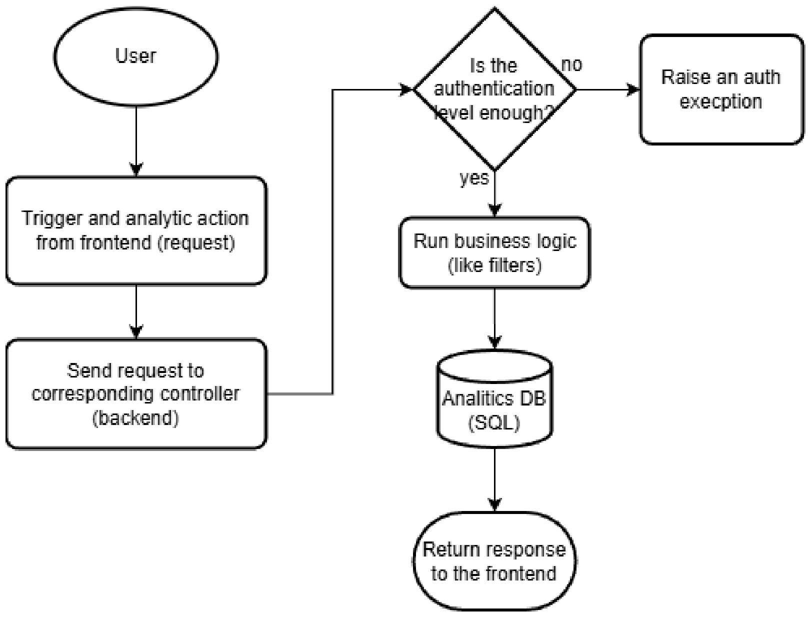

User
Trigger and analytic action from frontend (request)
Send request to corresponding controller (backend)
Is the authentication level enough?
no
yes
Raise an auth execption
Run business logic (like filters)
Analitics DB (SQL)
Return response to the frontend

Figure 4: High Level DB Diagram

In the figure we can see the expected data flow for report requests. note that the analytic database is not update during the request, it is designed to generate the report with the information from the previous day.

5.3 Updated version of ER Diagram

The next diagram addresses the relational part of the database corresponding to the sale's section. The core of the sales database consists of three entities: user, product, and sale. The sale entity has a master/detail structure, where each detail refers to a product. Additionally, the user entity has a seller relationship with the offered products and a buyer relationship with completed sales.

This section has been improved with respect to the last version in Workshop 1, updating the diagram format, and describing better the main relationships.

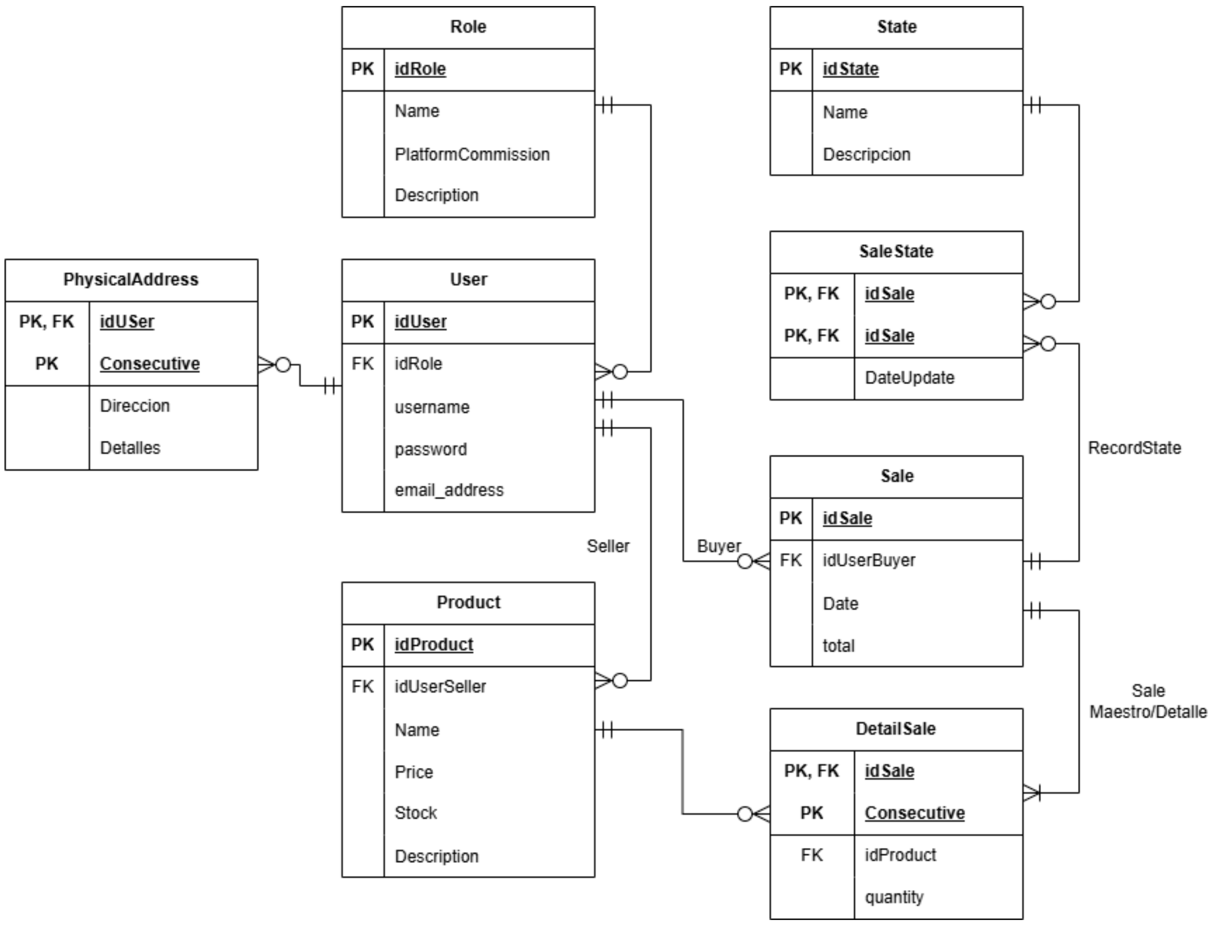


Figure 5: Entity-Relationship Diagram v.2.0

5.4 Storage Solutions

We propose a hybrid storage in order to reduce implementation's costs, potentially using a cloud storage solution for the data lake and an on-premise solution for a part of the data warehouse, due to our familiarity with these alternatives.

5.5 Stack technologies and Justification Matrix

The selection of technologies is directly driven by the architectural components defined above.

Technology	Component	Design Reasoning & Justification
PostgreSQL	Operational DB (Store)	**ACID Compliance:** Essential for handling money and inventory. Prevents "dirty reads" during concurrent checkouts. Open-source and cost-effective.
MongoDB	Operational DB (Catalog)	**Schema Flexibility:** Allows vendors to add unique attributes (e.g., "Chain Length", "Paper Type") without altering table structures.
PostgreSQL	Data Warehouse	**Simplicity:** For an academic MVP, utilizing a separate Schema within Postgres acts as a cost-effective Warehouse, allowing complex SQL aggregations for BI.
Python	ETL & ML Service	**Library Ecosystem:** Provides superior libraries for data manipulation (Pandas) and Machine Learning (Scikit-Learn) required for the Recommendation System.
Spring Boot	Backend API	**Robustness:** Provides mature integration with both SQL (JPA) and NoSQL (Spring Data Mongo), simplifying the implementation of the Hybrid architecture.

Table 12: Technology Justification Matrix

6 Improved Information Requirements

The information requirements describe the main types of information our system must retrieve, and the data sources where this information comes from, based on the functional requirements and user stories of the system. The platform's polyglot persistence strategy dictates the optimal data store for each retrieval pattern, balancing query flexibility with transactional integrity. Each requirement specifies the data needed, its source, the frequency of access, and the operational purpose it serves.

The improvements primarily involved restructuring each information requirement to explicitly include the specific data elements being retrieved and the frequency of access, but preserving the core requirement descriptions and data sources.

6.1 IR1 – User and Authentication Data

IR1.1 **Retrieve user profile and credentials from the User Service.**
Requirement: To authenticate users and populate profile information during login and session management.
Data: User credentials (e.g., hashed password), core profile data (e.g., name, email, user ID), and assigned role.
Frequency: Per user session initiation and for subsequent authorized requests.
Data Source: PostgreSQL.

IR1.2 **Retrieve user permissions and role-based access controls from the User Service.**
Requirement: To authorize actions across the platform, ensuring users can only access resources and perform operations permitted by their role.
Data: User role (Customer, Vendor, Admin) and associated permissions for platform features and data access.

Frequency: Continuously during user interaction to govern access to functions and data.
Data Source: PostgreSQL.

6.2 IR2 – Product Catalog and Search Data

IR2.1 **Retrieve structured product inventory from the Catalog Service.**
Requirement: To manage stock levels, vendor assignments, and availability for the transactional core.
Data: Product identifier, current stock quantity, vendor identifier, and availability status.
Frequency: Continuously during product browsing and with high frequency during the checkout process.
Data Source: PostgreSQL.

IR2.2 **Retrieve structured product pricing from the Catalog Service.**
Requirement: To calculate an order's base pricing, discounts, and taxes.
Data: Product identifier, base price, applicable discount rules, and tax category.
Frequency: Continuously during product browsing and with high frequency during the checkout process.
Data Source: PostgreSQL.

IR2.3 **Retrieve unstructured product specifications and attributes from the Search Service.**
Requirement: To power the flexible product catalog where different categories (e.g., T-Shirts, Processors) have varying attributes (e.g., size, RAM).
Data: Product identifier and a collection of category-specific attributes and their values.
Frequency: On-demand when rendering product detail pages and when indexing products for search.
Data Source: MongoDB.

IR2.4 **Retrieve products for user searching and browsing from the Search Service.**
Requirement: To provide fast, flexible, and faceted search results based on product names, descriptions, and dynamic attributes.
Data: Product names, descriptions, categories, and the full set of indexed specifications and attributes for filtering.
Frequency: Continuously in response to user search queries and filter operations.
Data Source: MongoDB.

6.3 IR3 – Transaction and Order Data

IR3.1 **Retrieve order history and status from the Order Service.**
Requirement: To provide customers with their purchase history and vendors with their sales records.
Data: Order identifier, user identifier, order date, line items (product, quantity, price), and current order status.
Frequency: On-demand when a user accesses their order history or a vendor reviews their sales.
Data Source: PostgreSQL.

IR3.2 **Retrieve transaction and commission records from the Transaction Service.**
Requirement: Calculate commission per vendor and process payouts to vendors to facilitate financial reconciliation.
Data: Transaction identifier, vendor identifier, order total, commission rate, and calculated commission amount.

Frequency: Periodically for payout processing and on-demand for financial reporting.
Data Source: PostgreSQL.

6.4 IR4 – Business Intelligence and Analytics Data

IR4.1 **Retrieve user interaction events from the Analytics Event Stream.**
Requirement: To capture semi-structured data on user behavior, such as page views, product clicks, and search queries, for analysis.
Data: User session identifier, event type, product identifiers, timestamps, and any relevant event metadata.
Frequency: Continuously, as events are generated by user activity.
Data Source: MongoDB.

IR4.2 **Retrieve aggregated sales and performance metrics from the Data Warehouse.**
Requirement: To compile key business metrics for dashboard visualizations, such as top-selling products/categories and vendor performance rankings.
Data: Aggregated sales figures, product and category performance data, vendor sales rankings, and user engagement metrics.
Frequency: Periodically for scheduled reporting and on-demand for dashboard queries.
Data Source: PostgreSQL is necessary to obtain the sales data, MongoDB is necessary to obtain interaction data by products and products categories.

7 Query Proposals

Below is a query proposal for each information requirement, detailing how the information can be extracted.

7.1 IR1 – User and Authentication Data

Data provided: This query gets the basic information about a user and his role name as long as the credentials given (username and password) match the information stored.

```
SELECT U.username, U.emailAddress, R.name
FROM User U
    JOIN Role R ON U.idRole = R.idRole
WHERE U.username = <username> AND U.password = <password>;
```

7.2 IR2 – Product Catalog and Search Data

7.2.1 IR 2.1 – Retrieve structured product inventory from the Catalog Service

Data provided: This query gets the basic information about the products and their sellers.

```
SELECT P.name,
    P.description,
    P.price,
    P.stock,
    U.username AS seller
FROM Products P
```

```
    JOIN Users U ON P.idSeller = U.idUser;
```

7.2.2 IR 2.2 – Retrieve structured product pricing from the Catalog Service

Data provided: This query retrieve the products (and their prices) related with a sale, allowing to calculate the sale's total price.

```
SELECT P.name, P.price, DS.quantity
FROM Product P, DetailSale DS
WHERE DS.idProduct = P.idProduct
AND DS.idSale = <idSale>;
```

7.2.3 IR 2.3 – Retrieve unstructured product specifications and attributes from the Search Service

Data provided: This query retrieves all the relevant information for a product given.

```
db.products.find(
    { name: <Name>},
    {
        _id: 0,
        categories: 1,
        attributes: 1
    }
)
```

7.2.4 IR 2.4 – Retrieve products for user searching and browsing from the Search Service

Data provided: This query retrieves all the relevant information for a product given.

```
db.products.find(
    { name: <Name>},
    {
        _id: 0,
        categories: 1,
        attributes: 1,
    }
)
```

7.3 IR3 – Transaction and Order Data

7.3.1 IR 3.1 – Retrieve order history and status from the Order Service

Data provided: These queries retrieve all the necessary information to list a particular user's sales, including the status and products for each sale.

SQL Queries:

- Query to retrieve the sales from a user:

```
SELECT S.idSale
FROM Sale S
WHERE S.idBuyer = <idUser>;
```

- Query to retrieve the products of a sale:

```
SELECT P.name, P.price, DS.quantity
FROM Product P, DetailSale DS
WHERE DS.idProduct = P.idProduct
AND DS.idSale = <idSale>;
```

- Query to retrieve the state of a sale:

```
SELECT S.idSale, St.name as state, SST.dateUpdate
FROM Sale S, SaleStatus SS, State ST
WHERE S.idSale = SS.idSale
AND SS.idState = ST.idState
ORDER BY SST.dateUpdate desc
LIMIT 1;\\
```

7.3.2 IR 3.2 – Retrieve transaction and commission records from the Transaction Service

Data provided: This query retrieves the information necessary to calculate the amount sold and its percentage corresponding to the platform.

```
SELECT U.username as seller,
       R.plataformCommission,
       P.name as product,
       DS.quantity,
       P.price
FROM User U, Role R, Product P, DetailSale DS
WHERE U.idUser = <idUser>
       AND DS.idProduct = P.idProduct
       AND P.idSeller = U.idUser
       AND U.idRole = R.idRole;
```

7.4 IR4 – Business Intelligence and Analytics Data

7.4.1 IR 4.1 – Retrieve user interaction events from the Analytics Event Stream

Data provided: This query retrieves the recent activity stream for a specific user, including pages visited, products viewed, and searches performed, for detailed behavioral analysis.

```
db.user_activity.find(
    { user_id: <idUser> },
    {
        _id: 0,
        timestamp: 1,
        activity_type: 1,
        page_url: 1,
        product_viewed: 1,
        search_query: 1
    }
).sort( { timestamp: -1 } ).limit(50)
```

7.4.2 IR 4.2 – Retrieve aggregated sales and performance metrics from the Data Warehouse

Data provided: These queries retrieve aggregated data to power the admin dashboard, showing top-selling products, sales by category, and best-performing suppliers.

SQL Queries:

- Query to retrieve the top 5 best-selling products:

```
SELECT P.name, SUM(DS.quantity) as total_units_sold
FROM Product P
    JOIN DetailSale DS ON P.idProduct = DS.idProduct
    JOIN Sale S ON DS.idSale = S.idSale
WHERE S.date >= <start_date>
GROUP BY P.idProduct, P.name
ORDER BY total_units_sold DESC
LIMIT 5;
```

- Query to retrieve sales statistics by product category:

```
SELECT C.name as category,
       COUNT(DISTINCT S.idSale) as number_of_orders,
       SUM(DS.quantity * P.price) as total_revenue
FROM Category C
    JOIN Product P ON C.idCategory = P.idCategory
    JOIN DetailSale DS ON P.idProduct = DS.idProduct
    JOIN Sale S ON DS.idSale = S.idSale
WHERE S.date >= <start_date>
GROUP BY C.idCategory, C.name
ORDER BY total_revenue DESC;
```

- Query to retrieve a ranked list of suppliers by sales performance:

```
SELECT U.username as supplier_name,
       COUNT(DISTINCT S.idSale) as total_orders,
       SUM(DS.quantity * P.price) as total_sales_volume
FROM User U
    JOIN Product P ON U.idUser = P.idSeller
    JOIN DetailSale DS ON P.idProduct = DS.idProduct
    JOIN Sale S ON DS.idSale = S.idSale
WHERE U.idRole = <vendor_role_id>
  AND S.date >= <start_date>
GROUP BY U.idUser, U.username
ORDER BY total_sales_volume DESC;
```

8 Concurrency Analysis

This section analyzes the possible concurrency problems and proposes solutions for them.

8.1 Scenarios prone to Concurrency

The following operations in the system are prone to concurrent access:

1. **Checkout and Order Placement**: Multiple customers may attempt to purchase the same product simultaneously.
2. **Inventory Updates**: Sellers may update stock (add or reduce) while customers are buying.
3. **User Profile Updates**: A user may update their profile concurrently from different devices.

8.2 Potential Concurrency Problems

Concurrent access in the above scenarios may lead to:

- **Race Conditions**: Two or more processes read and update shared data concurrently, leading to incorrect results.
- **Lost Updates**: One update overwrites another due to non-atomic write operations.
- **Overselling / Double Spending**: Inventory can drop below zero when multiple orders occur simultaneously.
- **Deadlocks**: Two transactions hold locks in incompatible order.
- **Inconsistent updates**: Concurrent updates created an inconsistency across related rows.
- **Cache Inconsistency**: Cached data may not reflect the latest database state.

8.3 Proposed Solutions

Solutions are divided into database-level, application-level, and architectural approaches.

8.3.1 Database-Level Solutions

Conditional Updates for Inventory Control. Use a single SQL statement that performs a conditional update to prevent overselling:

```
UPDATE products
SET stock = stock - v_quantity
WHERE product_id = v_product_id AND stock >= v_quantity;
```

This ensures the check and update occur atomically.

Optimistic Concurrency Control. Corroborate if key values have change during the process before insert the data:

```
UPDATE product
SET stock = stock - v_quantity
WHERE id = :id AND stock = :expected_stock;
```

Database Constraints. Use unique constraints, foreign keys, and check constraints (e.g., stock cannot be negative).

8.3.2 Application-Level Solutions

Short Transactions and Retry Logic. Keep transactions short and implement exponential backoff when conflicts occur.

Incorparate data validation Incorporate data validation in the backend a way to improve the DB consistency, and divide the work between components

8.3.3 Architectural Solutions

Inventory Reservation System. Implement a reservation workflow:

1. Reserve stock during checkout.
2. Hold reservation for a defined period.
3. Confirm after payment or release on expiration.

Message Queues for Asynchronous Processing. Use a queueing system for sending emails, updating analytics, and cache-refresh events.

9 Parallel and Distributed Databases

9.1 Justification

It's important to mention that not all components of a system warrant the complexity to distribute it. In fact, in our case, the core transactional entities: User, Order, and Transaction, are best served by a single PostgreSQL instance, potentially scaled vertically. However, the system has two scaling challenges: the high-volume, read-intensive Product Search (IR2) and the massive, write-intensive Analytics Event Stream (IR4). These requirements can be better achieved with a distributed architecture.

MongoDB allows us to shard the product catalog across multiple nodes, distributing the read load and enabling sub-second query response times even as the catalog and events grows exponentially. Therefore, using it wisely can allow us to overcome the potential scaling problems of our product. This sharding strategy provides horizontal scalability, allowing the database capacity to grow linearly by adding more shards to the cluster.

In the other side, in MongoDB, each shard is a replica set. If the primary node of a shard fails, a secondary automatically takes over with minimal downtime, ensuring both product search and analytics ingestion remain available.

In conclusion, MongoDB allows distributing and parallelize the system, while provides high availability for the distributed data layer without requiring additional application logic.

9.2 High-Level Design Proposal

Operational Core (PostgreSQL): A primary-standby PostgreSQL setup for high availability, handling IR1 (User/Auth) and IR3 (Order/Transaction). This ensures ACID compliance for critical financial and user data.

Distributed Data Layer (MongoDB Sharded Cluster): A single, sharded MongoDB cluster to handle both IR2 (Product Catalog and Search) and IR4 (Analytics). This unified approach simplifies operations while providing scalability for both workloads.

Product Data Sharding: Data is partitioned by product categories to group similar products. This optimizes query performance for category-based browsing and filtering, as related data is stored together.

Analytics Data Sharding: User interaction events are partitioned by a temporal key like the date time to distribute the writing load evenly across shards. The time-based sharding is particularly effective for analytics workloads, as it naturally groups data by time periods for efficient querying and potential data retention policies.

9.3 Illustrative Diagrams

The sequence of actions is very simple, for the case of the search service, the client makes a request, and consequently the Services API will request the data to MongoDB. Then, the Mongo Router must propagate the request to each shard, which going to execute the query locally, and returns the partials to the Router. Then, the router performs a merge-sort operation on the results and applies any final aggregation pipeline stages to «complete» the query results, and return it to the services API.

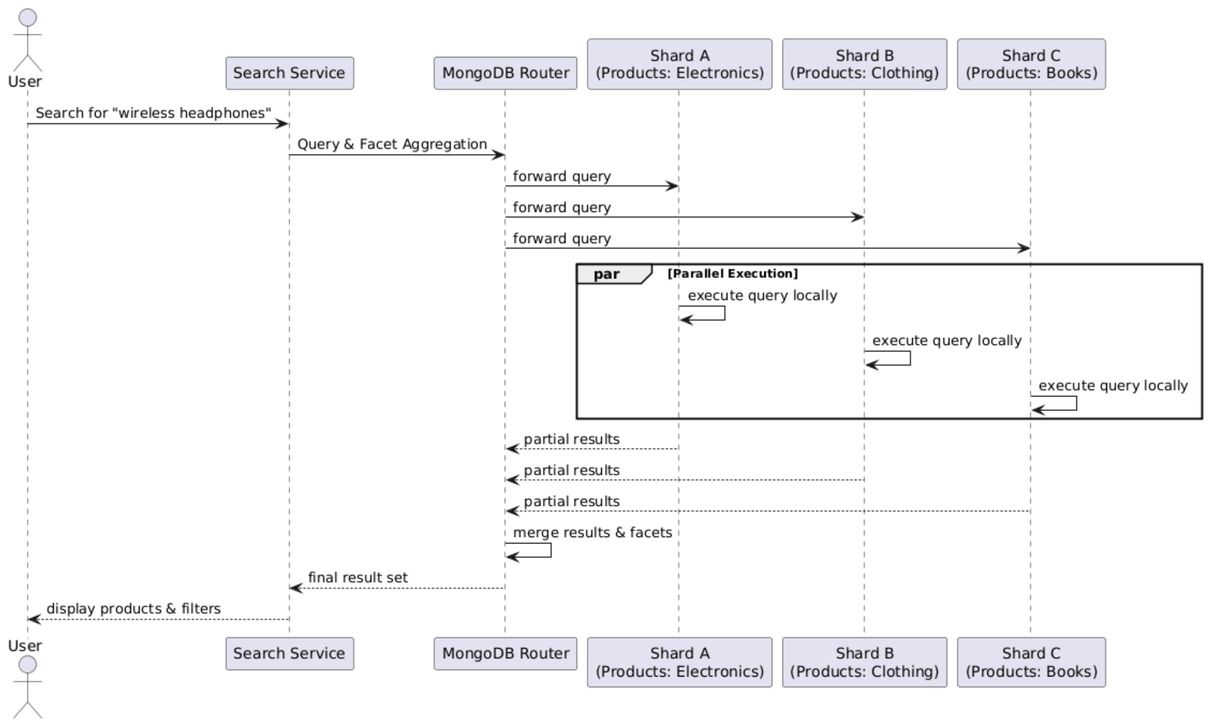


Figure 6: Products Search with MongoDB Shards

Something similar will happen in the case of the event recording, but instead of reading, the request will be to writing data. For writes, the MongoDB router uses the shard key to determine exactly which shard should receive and store each document, enabling parallel write operations across the cluster.

Finally, this database diagram illustrates the database distribution arch we have been talking about previously. The service layer connects to both database systems, with different services specializing in different data operations while maintaining clear boundaries between transactional and distributed data stores.

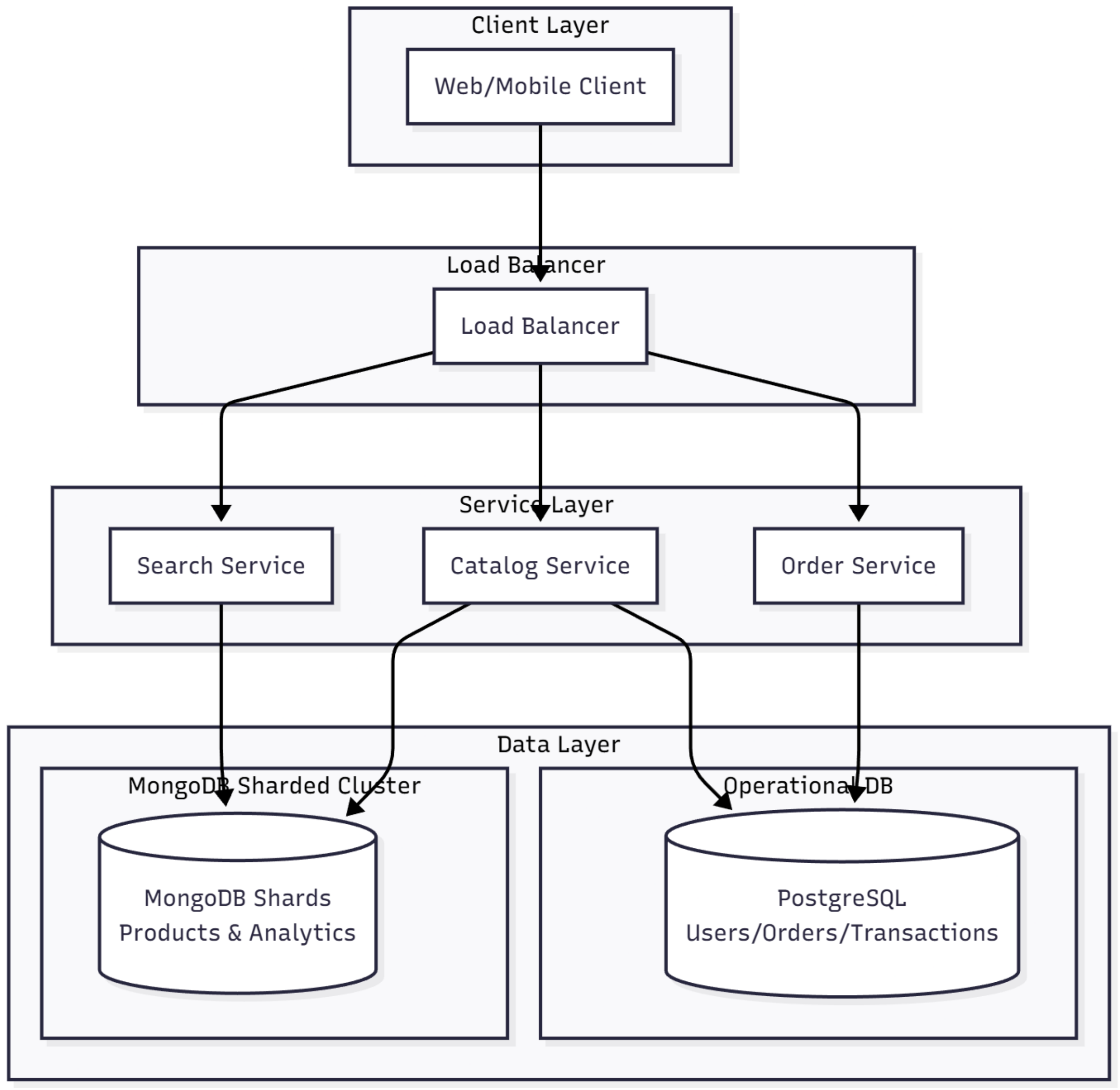


Figure 7: Database Distribution Architecture

The key advantage of this hybrid approach is maintaining strong consistency for core business transactions while achieving massive scalability for the most demanding data workloads, all within a manageable operational footprint.

10 Performance Strategies

10.1 Introduction

This section defines the concrete strategies ArtisanNexus can employ to meet the critical Non-Functional Requirements (NFRs) outlined in Section 3.2, specifically NFR 2.0 (Horizontal Scalability) and NFR 5.0 (System Performance). By leveraging the distributed nature of the polyglot architecture, these strategies ensure high throughput for the product catalog and strict ACID compliance for financial transactions.

10.2 Database Sharding (Category-Based)

- **Explanation & Implementation:** Horizontal Sharding is implemented within the MongoDB cluster utilizing a **Tag-Aware** or **Range-Based** strategy. The **Shard Key is the `Product Category`**, which groups similar products (e.g., "Electronics," "Clothing") onto specific shards. Additionally, the Analytics collection utilizes a **Time-Based Sharding** strategy (Shard Key: `timestamp`).
- **Justification & Benefit:** This strategy directly addresses **NFR 2.0 (Horizontal Scalability)**. By partitioning the catalog, the architecture prevents any single database node from becoming a throughput bottleneck. It provides **linear read scalability** and optimizes category-based browsing behavior by isolating queries to specific shards rather than broadcasting them to the entire cluster.
- **Target Component:** MongoDB Sharded Cluster (Data Layer).
- **Trade-offs & Challenges:** This introduces significant **operational complexity**. The system must monitor for "Shard Imbalance," where one category (e.g., "Clothing") becomes significantly larger or hotter than others, potentially requiring manual re-balancing.

10.3 Read Replicas & Replication

- **Explanation & Implementation:** We employ a **Primary-Standby** architecture for PostgreSQL, where the Primary handles writes and Standbys replicate data asynchronously. Similarly, every shard in the MongoDB cluster is configured as a **Replica Set**.
- **Justification & Benefit:** This is crucial for **NFR 1.0 (High Availability)**. It ensures the system can automatically recover from a node failure within the mandated 30-second window. It also allows us to offload read-heavy reporting queries to Read Replicas, preserving the Primary's capacity for transaction processing.
- **Target Component:** PostgreSQL Operational Core and MongoDB Shards.
- **Trade-offs & Challenges:** In the MongoDB context, relying on replication for high availability while allowing reads from secondaries introduces **Eventual Consistency**. A user might update a product description and momentarily see the old version if the read hits a lagging replica.

10.4 Table Partitioning

- **Explanation & Implementation:** Within the PostgreSQL database, we implement **Declarative Table Partitioning** on the `Orders` and `Sales` tables. These tables are partitioned by **Date** (e.g., monthly partitions).
- **Justification & Benefit:** This supports **FR 7.2 (Analytical Reporting)**. By partitioning historical sales data, the Admin Dashboard and ETL processes can scan smaller,

time-relevant subsets of data rather than executing full-table scans, significantly optimizing the retrieval of historical metrics.

- **Target Component:** PostgreSQL Operational DB (Transaction Service).
- **Trade-offs & Challenges:** Partitioning adds complexity to the database schema and maintenance. Queries that do not include the partition key (the date) in their `WHERE` clause may suffer from performance degradation as the planner must scan all partitions.

10.5 Multi-Layer Caching

- **Explanation & Implementation:** We implement a caching layer using **Redis**. The backend services check this cache before querying the database (Look-Aside pattern). We cache **Session Tokens** for authentication and **Product Details** for frequently accessed items.
- **Justification & Benefit:** This is the primary driver for **NFR 5.0 (System Performance)**. It ensures that 95% of read-only API calls respond in under 300ms by serving static or semi-static content from memory, drastically reducing latency and database load.
- **Target Component:** Backend Service Layer (Auth Service & Catalog Service).
- **Trade-offs & Challenges:** This introduces the challenge of **Cache Invalidation**. We must ensure that when a vendor updates a product price in the database, the corresponding cache entry is immediately invalidated to prevent customers from seeing incorrect pricing.

10.6 Parallel Processing

- **Explanation & Implementation:** The MongoDB Router (`mongos`) utilizes a scatter-gather approach to parallelize search queries across multiple shards when necessary. Additionally, the ingestion of user interaction events is decoupled and processed asynchronously.
- **Justification & Benefit:** This addresses **FR 3.1 (High-Speed Search)** and **NFR 8.0 (Constant Data Ingestion)**. It ensures that complex search queries return results quickly by leveraging the combined processing power of the cluster, and that high-volume clickstream writing does not block the user interface.
- **Target Component:** MongoDB Router and Search Service.
- **Trade-offs & Challenges:** Debugging performance issues becomes more difficult with parallel query execution, as a slow response could be caused by a single lagging shard slowing down the entire aggregated result.

10.7 Asynchronous ETL Pipeline

- **Explanation & Implementation:** We utilize a dedicated ETL process that triggers daily at midnight. It extracts raw data from the Operational DB and Data Lake, transforms it, and loads it into the Data Warehouse.
- **Justification & Benefit:** This strategy provides **Workload Isolation**. It ensures that complex, resource-intensive analytical queries required for the BI Dashboard (FR 7.2) are executed against the Data Warehouse, preventing them from locking rows or consuming resources in the Operational DB used by active customers.
- **Target Component:** ETL Component & Data Warehouse.

- **Trade-offs & Challenges:** This introduces **Data Latency**; the business intelligence reports will typically reflect data up to the previous day, rather than real-time status. Furthermore, joining data across the Polyglot boundary (PostgreSQL Users + MongoDB Events) during the Transformation phase adds significant logic complexity.

10.8 Horizontal Scaling of Stateless Services

- **Explanation & Implementation:** The Backend Service Layer is designed to be stateless and deployed behind a Load Balancer. This allows us to deploy multiple container instances of specific services, such as the Order Service, on demand.
- **Justification & Benefit:** This directly satisfies **NFR 2.0 (Horizontal Scalability)**. It allows the system to handle traffic spikes—such as during a flash sale—by simply spinning up additional instances to handle the increased concurrent checkout requests.
- **Target Component:** Service Layer (Order Service) & Load Balancer.
- **Trade-offs & Challenges:** While effective, this increases **Infrastructure Costs** due to the need for additional compute resources (VMs/Containers) and adds complexity to the Load Balancer configuration to ensure sticky sessions are not required or handled correctly.

10.9 System-Wide Integration

Combining these strategies creates a robust but complex ecosystem. The primary system-wide trade-offs include:

- **Consistency vs. Availability (CAP Theorem):** We have made a deliberate architectural split. The Transactional Core favors **Consistency** (CP) to protect financial data, while the Catalog and Search layers favor **Availability** (AP) to ensure a responsive user experience, accepting eventual consistency as a necessary trade-off.
- **Operational Complexity:** Adopting a Polyglot Persistence strategy with both Sharded NoSQL and Replicated SQL databases significantly increases the maintenance burden compared to a monolithic architecture. It requires advanced monitoring and distinct disaster recovery procedures for each data store.
- **Development Overhead:** The strict separation of Operational and Analytical data means that developers cannot simply "join" tables across the entire system. Application logic or ETL processes must bridge the gap between the User data in PostgreSQL and the Event data in MongoDB.

11 References

References

[1] Osterwalder, A. & Pigneur, Y. Business Model Generation. Wiley, 2010.

[2] Kimball, R. & Ross, M. The Data Warehouse Toolkit. Wiley, 2013.

[3] Ricci, F., Rokach, L., Shapira, B. Recommender Systems Handbook. Springer, 2015.