[Date]

# “Multi-Tenant E-Commerce Marketplace with Search & Recommendations”

GROUP BY:
B BHARATH KUMAR
G THARUN
A RAMCHARAN
S SHAIK SYED
O SIVA SHANKAR

# SECTION 1 — Introduction & System Overview

## 1.1 Purpose of the Document

This document presents a comprehensive **System Design Specification** for a **Multi-Tenant E-Commerce Marketplace Platform**.
It serves as a technical blueprint for engineers, architects, product owners, and stakeholders involved in the design, development, deployment, and maintenance of the platform.

The document emphasizes:

- Scalability
- High availability
- Multi-tenant isolation
- Modern cloud-native architecture
- Search and recommendation engines
- Secure payments
- Resilient distributed communication
- Observability and DevOps maturity

## 1.2 System Overview

**1.2 System Overview**

The marketplace enables multiple independent sellers (tenants) to create and manage their online stores within a unified platform.
Buyers can search, browse, compare, and purchase products across all tenants using a unified catalog.

**The system supports:**

- Multi-tenant onboarding and tenant isolation
- Product catalog ingestion & management
- Distributed inventory control
- Shopping cart and order lifecycle management
- Secure payment processing
- Full-text search and product ranking
- Personalized recommendations
- Review & rating system
- Seller dashboards & analytics

- High-performance caching and async event-driven workflows
- Monitoring, logging, and metrics

## 1.3 Problem Statement

Most marketplaces struggle with:

- Handling thousands of tenants with isolated data
- Ensuring consistent search performance across large catalogs
- Preventing inventory mismatches during peak load
- Maintaining secure and compliant payment operations
- Scaling recommendations and search indexing
- Ensuring resilience during microservice failures
- Supporting near real-time analytics for sellers

This system design solves these problems through a **distributed microservices architecture**, **tenant-aware data models**, and **scalable cloud-native infrastructure**.

## 1.4 Key System Goals

**Primary Goals**

1. Support **unlimited tenants** with strict data separation.
2. Provide a **global searchable catalog** that updates in near real-time.
3. Deliver **fast and accurate recommendations** using behavioral data.
4. Ensure **high availability (99.9%)** across all services.
5. Enable **horizontal scaling** during peak sales events.
6. Provide **secure, PCI-compliant payment flows**.
7. Maintain end-to-end **observability** for operational health.

**Secondary Goals**

- Low-latency user experience (<150ms API response target).
- Distributed caching to offload database load.
- Event-driven communication for order, inventory, and search indexing.
- Modular services enabling independent deployment and scaling.

## 1.5 Scope of the System

**Included in Scope**

- Multi-tenant management
- Product ingestion, catalog, and inventory
- Search indexing pipeline
- Buyer experience (cart, checkout, payment, orders)
- Recommendation system
- Seller management dashboards
- Notification service
- Logging, metrics, tracing
- CI/CD pipelines

## 1.6 High-Level Features

**Buyers**

- Account creation, browsing, search
- Product filters and sorting
- Add to cart, wishlist
- Checkout, payment, order tracking
- Ratings & reviews
- Personalized recommendations

**Sellers**

- Seller onboarding, verification
- Product upload & catalogs
- Inventory management
- Order fulfillment workflows
- Sales analytics dashboard

**Platform**

- Multi-tenant data management
- Centralized search indexing
- Recommendation ML pipelines
- Fraud detection
- Observability stack
- Auto-scaling infrastructure

# SECTION 2 — Stakeholders & User Personas

## 2.1 Stakeholder Overview

**Stakeholders play a pivotal role in the development, operation, and continuous improvement of a marketplace platform. They are any individuals, teams, or systems that have a vested interest in how the platform functions, interacts, or delivers value. A successful marketplace platform requires an in-depth understanding of all stakeholder groups to align its features, functionality, and business strategies with the needs and expectations of these diverse groups.**

**By categorizing stakeholders, we can ensure that their needs are effectively met and that the platform's design prioritizes the most important features while maintaining flexibility for all relevant parties. The stakeholders can be grouped into four broad categories:**

---

**1. Primary Stakeholders**

**These are the key users and entities whose actions directly influence the success and viability of the marketplace. They are the backbone of the platform, and their satisfaction is paramount.**

**Examples of Primary Stakeholders:**

- **Buyers (Consumers): Individuals or organizations who purchase goods or services from the marketplace. Their primary needs include user-friendly interfaces, a variety of products or services, reliable payment methods, and timely deliveries.**
- **Sellers (Vendors): These are businesses or individuals who offer products or services on the platform. Their needs include an easy-to-use dashboard, effective inventory management tools, promotional features, and a streamlined payment process.**
- **Marketplace Owner/Platform Provider: The entity responsible for running the marketplace, ensuring it remains operational, compliant, and financially sustainable. They focus on platform performance, user experience, security, and customer support.**

**Why they're important:**
**These stakeholders have the most direct interaction with the platform and will drive its overall success. Buyers and sellers determine the economic viability of the platform, while the platform owner ensures everything functions as intended.**

---

**2. Secondary Stakeholders**

**Secondary stakeholders influence the marketplace indirectly but still play an important role in shaping the ecosystem. These groups may not engage directly with every aspect of the platform, but they provide support and services that enable the marketplace to function smoothly.**

**Examples of Secondary Stakeholders:**

- **Payment Processors & Gateways: Services that handle the transactions between buyers and sellers. They are crucial for enabling secure and efficient financial exchanges.**

- **Shipping & Logistics Partners: Companies responsible for delivering products to consumers. Their efficiency and reliability directly impact the buyer's experience and the platform's reputation.**
- **Marketing & Advertising Partners: Agencies or service providers that help the marketplace promote its platform, attract new buyers, and increase seller visibility.**
- **Legal & Compliance Authorities: These stakeholders ensure the platform complies with local and international laws regarding e-commerce, data privacy, consumer protection, and taxation.**

**Why they're important:**
**Although they don't interact directly with the marketplace interface, secondary stakeholders contribute to the platform's infrastructure. Without their services, the marketplace might struggle with payment failures, delayed shipments, or legal issues, impacting overall user satisfaction.**

---

**3. Internal Operational Stakeholders**

**Internal stakeholders are individuals or teams within the organization that work behind the scenes to maintain, improve, and evolve the marketplace platform. These stakeholders may not directly interact with the users but are vital to the platform's ongoing success and innovation.**

**Examples of Internal Operational Stakeholders:**

- **Development Team (Engineers & Designers): Responsible for building, maintaining, and updating the marketplace platform. This team ensures that the platform is technically sound, scalable, and secure.**
- **Product Managers & Analysts: They define the platform's features, functionality, and user experience, aligning development priorities with market needs and business objectives.**
- **Customer Support Team: Handles inquiries, complaints, and issues from buyers and sellers, ensuring a high level of user satisfaction.**
- **Operations Team: Manages day-to-day activities such as overseeing product listings, monitoring platform performance, and resolving any operational issues.**

**Why they're important:**
**These stakeholders ensure the platform's technical stability and ongoing usability. They directly impact how the platform evolves to meet user needs and how effectively the platform responds to challenges, both technical and operational.**

---

**4. External Integrations & Systems**

**This category includes third-party services and external systems that are integrated into the platform to enhance its functionality. These systems may not be internal to the platform but are crucial for enabling specific features, processes, or business operations.**

**Examples of External Integrations & Systems:**

- **Third-Party APIs: These may include tools for fraud detection, AI-powered recommendations, or customer behavior analytics. External APIs may also handle tasks like tax calculations, currency conversions, or multi-language support.**
- **External Marketplaces & Channels: Platforms like Amazon, eBay, or social media channels (Instagram, Facebook) that are integrated with the marketplace to enable cross-platform selling or increased exposure.**
- **Enterprise Resource Planning (ERP) Systems: Larger sellers or the marketplace platform itself might integrate an ERP system to manage inventory, finances, and supply chains more efficiently.**
- **Security & Authentication Providers: External tools for ensuring the platform's security, including two-factor authentication, encryption, and user verification services.**

**Why they're important:**
**External integrations play a key role in expanding the capabilities of the marketplace, ensuring the platform can deliver services that are beyond its inherent capabilities. By leveraging third-party tools and systems, the platform can improve the user experience, enhance operational efficiency, and scale more easily.**

# 2.2 Primary Stakeholders

## 2.2.1 Buyers (End Users)

**Role:**
Buyers are the end customers who visit the marketplace to explore, compare, and purchase products or services. They represent the primary revenue-driving audience of the platform, and their overall satisfaction directly impacts marketplace growth, retention, and brand reputation.

---

**Goals**

Buyers expect a seamless, efficient, and trustworthy shopping experience. Their main objectives include:

- **Find products quickly using search and filters**
  Buyers rely on accurate search results, intuitive filtering, well-structured categories, and clear product information to make informed purchasing decisions.
- **Easy checkout and secure payment**
  The checkout process must be frictionless, with minimal steps, multiple payment options, and robust security measures to ensure trust during the transaction.
- **Order tracking and fast delivery**
  Buyers expect clear visibility into order status and timely delivery updates. Real-time tracking and proactive notifications enhance confidence and reduce support inquiries.
- **Relevant recommendations**
  Personalized product suggestions—based on browsing history, preferences, and popularity—help buyers discover items more efficiently and increase overall engagement.

---

**Pain Points**

To deliver a superior experience, the system must identify and address the key frustrations that buyers commonly encounter:

- **Slow search results**
  Laggy or imprecise search experiences create friction and may drive users away.
- **Poor product discovery**
  Insufficient filtering, irrelevant recommendations, or cluttered category structures hinder the ability to find suitable products.
- **Unreliable order tracking**
  Missing or outdated delivery updates reduce trust and increase customer support demands.
- **Payment failures**
  Issues during checkout—including declined transactions, slow verification, or confusing payment flows—lead to abandoned carts and lost revenue.

---

**System Implications**

To satisfy buyer needs and address their pain points, the platform must:

- Deliver **fast, accurate search and filtering** backed by optimized indexing and intelligent ranking.
- Provide an **intuitive checkout flow** with stable, secure payment integrations.
- Implement **real-time order tracking** through reliable logistics integrations and proactive notifications.
- Offer **personalized recommendations** using data-driven algorithms and user behavior analysis.
- Maintain a **high-performance, user-centered interface** that is responsive, accessible, and easy to navigate across all devices.

## 2.2.2 Sellers (Vendors / Tenants)

Each seller represents an independent business operating within the marketplace ecosystem. Sellers rely on the platform to reach customers, manage their products, and complete transactions efficiently. Their success directly influences marketplace inventory, diversity, revenue, and long-term sustainability.

---

**Goals**

Sellers require a robust, intuitive, and reliable set of tools that enable them to run their business with minimal complexity. Their key objectives include:

- **Quick Onboarding**
  Seamless account creation, business verification, and storefront setup to begin selling as quickly as possible.
- **Easy Catalog and Inventory Management**
  Straightforward interfaces for adding, editing, and organizing products, with bulk upload tools, media management, and automated stock-level updates.
- **Accurate Sales Insights**
  Access to real-time dashboards, performance analytics, and historical data to guide pricing strategies, promotions, and operational decisions.
- **Low Operational Friction**
  Streamlined workflows for order fulfillment, returns management, customer communication, and compliance processes to reduce overhead and maximize efficiency.

---

**Pain Points**

Sellers often face operational challenges that can hinder productivity and profitability. Key issues include:

- **Slow Product Upload Tools**
  Inefficient or error-prone upload processes, especially for sellers with large catalogs, lead to delays and frustration.
- **Inventory Inconsistencies**
  Out-of-sync stock levels cause overselling, order cancellations, and customer dissatisfaction.
- **Delayed Payments**
  Slow or unclear payout cycles reduce cash flow and negatively impact the seller's ability to operate the business effectively.
- **Lack of Analytics**
  Limited visibility into sales performance, buyer behavior, or marketplace trends restricts a seller's ability to optimize and grow.

---

**System Implications**

To effectively support sellers and maintain marketplace integrity, the platform must deliver:

- **Comprehensive seller tools**
  High-performance dashboards, catalog management utilities, and automated workflows to handle inventory, pricing, and product data at scale.
- **Strict data isolation**
  Ensuring each seller's data—inventory, transactions, and analytics—is isolated and accessible only to the authorized seller, preventing cross-tenant data exposure.
- **Reliable payments infrastructure**
  Transparent payout schedules, automated settlements, and integration with trusted payment partners.

- **Advanced analytics capabilities**
  Actionable insights presented through intuitive dashboards and customizable reports to help sellers optimize operations.
- **Scalable performance**
  Support for large product catalogs, high traffic, and peak-period sales without degradation of seller tools or operations.

## 2.2.3 Marketplace Admin

Marketplace Administrators are responsible for overseeing the entire ecosystem and ensuring that the platform operates smoothly, securely, and in compliance with policies and regulations. They serve as the central authority for tenant management, operational monitoring, and issue resolution across all user types.

---

**Goals**

Administrators require visibility, control, and governance tools that allow them to efficiently manage multi-tenant operations. Their core objectives include:

- **Monitor Tenant Activity**
  Track seller behavior, performance, compliance, and marketplace health across multiple tenants.
- **Manage Platform-Level Configurations**
  Configure global settings, feature toggles, policies, fees, and system-wide parameters that impact all users.
- **Handle Disputes & Prevent Fraud**
  Review escalations between buyers and sellers, detect suspicious behavior, and intervene when fraudulent activity is detected.
- **Ensure Compliance and Platform Stability**
  Maintain adherence to legal, security, and operational standards while ensuring the platform remains resilient and performant.

---

**Pain Points**

Administrators face complex oversight challenges, especially in distributed architectures and multi-tenant environments:

- **Lack of Visibility into Microservice Health**
  Without centralized monitoring, identifying issues across services is difficult and time-consuming.
- **Slow Detection of Fraudulent Transactions**
  Insufficient analytics or alerting mechanisms may delay the identification of fraud patterns or policy violations.

- **Limited Operational Control**
  Without proper tools, administrators may struggle to enforce policies, audit changes, or monitor critical platform metrics.

---

**System Implications**

To support effective governance, the platform must provide:

- **Comprehensive Admin Dashboards**
  Centralized views of tenant activity, financial metrics, system health, and operational alerts.
- **Real-Time Alerts & Monitoring Tools**
  Automated detection of anomalies, performance issues, and fraud signals, coupled with timely notifications.
- **Audit & Compliance Features**
  Full audit trails for administrative actions, configuration changes, and system events.
- **Microservice Observability**
  Metrics, logs, traces, and error tracking consolidated into an accessible monitoring environment.

---

---

# 2.3 Secondary Stakeholders

## 2.3.1 Delivery & Logistics Partners

Although the marketplace may not directly handle shipments, delivery and logistics partners play a crucial role in the buyer experience. They rely on timely, accurate data to successfully fulfill orders and ensure smooth last-mile delivery.

---

**Goals**

Logistics partners require efficient integration with marketplace operations to reduce delivery errors and improve fulfillment outcomes. Their main objectives include:

- **Access Order Details**
  Ability to retrieve order information such as pickup location, recipient address, delivery instructions, and package specifications.
- **Update Delivery Status**
  Provide real-time tracking updates—shipment picked, in-transit, out for delivery, delivered, or failed—back to the marketplace.
- **Reduce Failed Deliveries**
  Minimize misinformation, delivery retries, and customer dissatisfaction caused by incomplete or incorrect data.

---

**Pain Points**

Operational challenges often emerge when logistics data is incomplete or delayed:

- **Delayed Updates**
  Slow or inconsistent tracking pushes affect buyer trust and increase customer support burden.
- **Missing Order Context**
  Insufficient order-level information—such as contact details or delivery notes—can lead to delivery failures.
- **Integration Difficulties**
  Poorly structured APIs or unreliable communication channels hinder collaboration with logistics providers.

---

**System Implications**

To enable smooth logistics integration, the marketplace must deliver:

- **Robust APIs and Webhooks**
  Well-documented endpoints for retrieving orders and sending real-time status updates, with guaranteed message delivery.
- **Clear Contextual Data Sharing**
  Provide accurate, complete order details and delivery metadata required for fulfillment.
- **Reliable Event-Driven Architecture**
  Support event notifications for order creation, shipment updates, and delivery completions.
- **Secure Partner Access**
  Ensure that logistics partners only access authorized data through scoped permissions and secure authentication.

## 2.3.2 Payment Service Providers (PSPs)

Payment Service Providers (PSPs) such as Razorpay, Stripe, PayPal, and others handle the financial transactions between buyers and sellers. Their integrations ensure secure, compliant, and seamless payment experiences across the marketplace. PSPs play a crucial role in enabling multi-method payments, managing risk, and ensuring regulatory compliance for all monetary flows on the platform.

---

**Goals**

PSPs aim to maintain secure, efficient, and compliant payment processing while minimizing operational risks. Their primary objectives include:

- **Secure, Reliable Payment Processing**
  Execute transactions with low failure rates, high uptime, and strong encryption to ensure user trust and financial accuracy.
- **Risk Scoring and Fraud Detection**
  Evaluate transactions for fraud patterns, mitigate high-risk behavior, and provide security insights back to the marketplace.
- **Smooth Integration with Platform Payment Flows**
  Ensure their APIs, callbacks, and settlement processes integrate seamlessly with the marketplace's payment microservices.

---

**Pain Points**

PSPs often encounter challenges when working with marketplace platforms, especially if integrations are not implemented correctly:

- **Incorrect Parameters Sent by Platforms**
  Invalid payloads, mismatched signatures, wrong metadata, or incorrect order amounts lead to transaction failures and disputes.
- **High Chargeback Rate**
  Excessive disputes and refunds not only impact PSP trust but may result in higher fees, tighter restrictions, or account freezes.
- **Lack of Standardized Payment Flows**
  Poor implementation of callbacks, retries, or reconciliation processes can impact settlement accuracy and increase operational work.

---

**System Implications**

To maintain strong PSP relationships and ensure seamless payment operations, the marketplace must:

- **Comply with PSP Requirements**
  Follow each provider's API specifications, payload formats, authentication methods, webhook standards, and reconciliation guidelines.
- **Adhere to PCI-DSS Standards**
  Ensure secure storage, processing, and transmission of card data, with proper tokenization and restricted access to sensitive information.
- **Implement a Payment Microservice Architecture**
  Payment services must be isolated, secure, and resilient, supporting retries, idempotency, and consistent transaction state management.
- **Provide Accurate Metadata & Error Handling**
  Validate all parameters sent to PSPs, handle errors gracefully, and reconcile transactions on a scheduled basis.

- **Maintain Fraud Detection and Risk Monitoring Tools**
  Integrate PSP fraud insights and supplement them with marketplace-level monitoring to minimize risk exposure and chargebacks.

## 2.3.3 Customer Support Teams

Customer Support Teams act as the primary point of contact for both buyers and sellers when issues arise. They help resolve disputes, process refunds, clarify order questions, and ensure smooth resolution of operational problems. Their effectiveness directly impacts user satisfaction, retention, and trust in the marketplace.

---

**Goals**

Customer support teams require fast, reliable, and centralized access to platform data so they can respond to issues efficiently. Their core objectives include:

- **Quickly Access Order Data**
  Support agents must retrieve complete order histories, payment statuses, delivery updates, and customer details in real time to assist users effectively.
- **Resolve Disputes**
  Whether related to product quality, delivery issues, or payment discrepancies, support teams need tools to investigate, mediate, and resolve complaints.
- **Process Refunds**
  Agents require streamlined workflows that allow them to initiate refunds, manage returns, or escalate cases to administrators when needed.
- **Provide a Smooth Customer Experience**
  Faster access to accurate information reduces resolution time, improves satisfaction scores, and lowers operational costs.

---

**Pain Points**

Support teams often struggle with fragmented or hard-to-access data sources, which increases handling time and reduces efficiency:

- **Lack of Centralized View**
  Data may be scattered across multiple microservices or systems, forcing agents to switch between tools to assemble a clear picture of an issue.
- **Slow Data Retrieval**
  Delays in fetching order or transaction details directly impact customer experience and prolong resolution time.
- **Inconsistent Information**
  Without a unified data layer, support teams may encounter conflicting records from different services.

---

**System Implications**

To ensure effective support operations, the marketplace must provide:

- **A Dedicated Support Portal / API Layer**
  A centralized interface or API layer that aggregates data from orders, payments, logistics, and user services, giving support teams a unified view.
- **Real-Time Data Access**
  Low-latency access to order timelines, payment events, delivery updates, and conversation history.
- **Role-Based Access Control (RBAC)**
  Ensures support agents only access the data necessary for their role, maintaining privacy and compliance.
- **Integrated Dispute & Refund Workflows**
  Streamlined tools for initiating refunds, escalating cases, and tracking dispute outcomes.
- **Activity Logging & Auditing**
  Every support action should be logged to support compliance, monitoring, and accountability

# 2.4 Internal Operational Stakeholders

## 2.4.1 DevOps & SRE Teams

DevOps and Site Reliability Engineering (SRE) teams are responsible for ensuring the platform's operational excellence. They maintain system reliability, performance, scalability, and security across all environments. Their work enables continuous delivery, stable deployments, and efficient monitoring of the platform's microservices and infrastructure.

---

**Goals**

DevOps & SRE teams aim to maintain a resilient, self-recovering, and highly observable system. Their key objectives include:

- **Auto-Scaling Reliability**
  Ensure workloads scale automatically based on demand—without service degradation—through efficient resource management, horizontal scaling, and load balancing.
- **Zero-Downtime Deployments**
  Enable continuous integrations and rolling updates so that deployments do not interrupt user experience, leveraging blue-green or canary deployment strategies.
- **Incident Response**
  Detect, triage, and resolve incidents rapidly, with clear runbooks, automated alerts, and root-cause analysis mechanisms.

- **Observability Coverage (Logs, Metrics, Traces)**
  Achieve end-to-end visibility across services with structured logging, real-time metrics, distributed tracing, and centralized dashboards.

---

**Pain Points**

Operational challenges often arise when systems lack cohesion, visibility, or automation:

- **High Noise Alerts**
  Excessive or non-actionable alerts lead to alert fatigue, making it harder to identify critical issues in real time.
- **No Unified Health Dashboard**
  Fragmented dashboards across different tools make it difficult to monitor the health of microservices or understand system-wide performance at a glance.
- **Manual Scaling or Deployments**
  Without proper automation, scaling and deployment activities become slow and error-prone.
- **Opaque Failure Modes**
  Limited observability makes diagnosing failures or tracing issues across distributed services time-consuming.

---

**System Implications**

To support operational excellence, the architecture must be designed with DevOps and SRE needs in mind:

- **Cloud-Native Architecture**
  Built on containerization, orchestration (e.g., Kubernetes), and managed cloud services to support scalability and reliability.
- **Strong Observability Design**
  Full-stack observability—including logs, metrics, traces, dashboards, and alerting—must be integrated from day one.
- **Automated Deployment Pipelines**
  CI/CD pipelines must support automated testing, blue-green/canary deployment, rollback strategies, and artifact versioning.
- **Unified Monitoring & Health Dashboards**
  A single pane of glass for service health, dependency maps, latency breakdowns, and error visuals.
- **Resilience and Self-Healing**
  Auto-restart, auto-scale, circuit breakers, rate limiting, and retry patterns help maintain platform stability under load or partial failure.
- **Incident Management Workflows**
  Built-in processes for alert routing, escalation policies, incident tracking, and postmortems.

## 2.4.2 Data Engineering & Analytics Team

The Data Engineering & Analytics Team is responsible for building and maintaining the data foundation that powers insights, reporting, and intelligent features across the marketplace. This team ensures that data is collected, processed, and transformed reliably so that business users, sellers, and automated systems (such as recommendation engines) can make informed decisions.

They enable everything from real-time operational dashboards to machine learning models that personalize user experiences and optimize seller performance.

---

**Goals**

The team's objectives revolve around producing high-quality, consistent, and actionable data:

- **Extract Accurate Marketplace Data**
  Collect structured and unstructured data from multiple microservices—orders, catalog, payments, logistics—and ensure validation, deduplication, and transformation into reliable datasets.
- **Maintain Recommendation Engine Pipelines**
  Build and monitor ML pipelines that process user behavior, product interactions, and historical trends to serve real-time or batch recommendation models.
- **Build Real-Time Dashboards for Sellers**
  Provide sellers with up-to-date insights on sales, traffic, inventory, and customer behavior through streaming data pipelines and analytics layers.
- **Enable Business Intelligence & Reporting**
  Support marketplace leaders with aggregated insights to drive strategy, detect anomalies, and measure performance.

---

**Pain Points**

Data teams face significant challenges when upstream systems and event flows are not well-structured:

- **Poor Data Quality**
  Incomplete fields, inconsistent values, missing events, or inaccurate timestamps reduce trust and require manual cleanup and error handling.
- **Inconsistent Data Formats**
  Different microservices may use varying schemas, naming conventions, or event structures, complicating ingestion and transformation processes.
- **Lack of Standardized Event Models**
  Without unified event schemas, it becomes harder to correlate data across services, hindering analytics and ML accuracy.

---

**System Implications**

To support strong analytics capabilities and ML-driven features, the platform must:

- **Adopt Event Sourcing Architecture**
  Capture all critical business events (orders, payments, interactions) as immutable logs, supporting replayability, historical analysis, and auditability.
- **Implement Scalable Data Lake Pipelines**
  Use cloud-native storage and processing layers (e.g., object storage + distributed compute) to store raw, curated, and enriched datasets for analytics and ML.
- **Use Stream Processing Frameworks**
  Enable real-time transformations and dashboards using technologies such as Kafka, Kinesis, Pulsar, or similar.
- **Define Unified Data Schemas**
  Standardize event structures, metadata fields, and naming conventions across microservices.
- **Ensure Data Contract Governance**
  Teams must follow strict data contracts to avoid breaking pipelines with incompatible changes.
- **Support ML Lifecycle Management**
  Provide pipelines for training, versioning, evaluating, and deploying machine learning models, integrated with monitoring and rollback mechanisms.

## 2.4.3 Compliance & Security Teams

Compliance & Security Teams are responsible for safeguarding the marketplace against legal, regulatory, and operational risks. They ensure that all data—especially sensitive information such as Personal Identifiable Information (PII) and payment-related data—is handled securely and in accordance with regional laws, industry standards, and organizational policies.

Their work is essential for maintaining user trust, avoiding regulatory penalties, and ensuring long-term platform resilience.

---

**Goals**

These teams focus on enforcing security best practices, maintaining auditability, and ensuring compliance across all microservices and data flows. Their core objectives include:

- **Secure Handling of PII and Payment Data**
  Ensure that user and transaction data is stored, processed, and transmitted safely using encryption, tokenization, and strict access policies aligned with GDPR, PCI-DSS, and other regulatory frameworks.
- **Audit Trails**
  Maintain complete, tamper-proof audit logs of critical system events—such as user actions, admin changes, access attempts, and configuration updates—to support investigations, compliance checks, and forensics.

- **Access Control Compliance**
  Enforce robust role-based (RBAC) or attribute-based access control (ABAC) to ensure users only access data and resources permitted for their roles.
- **Security Monitoring & Threat Detection**
  Continuously monitor the platform for potential vulnerabilities, suspicious activities, fraud signals, or policy violations.

---

**Pain Points**

Security and compliance teams often face challenges when the platform lacks centralization or standardization:

- **Inconsistent Logging Across Services**
  Variability in log formats, log levels, and event structures complicates audits, makes incident investigations harder, and increases the risk of missing critical events.
- **Lack of Centralized Identity Management**
  Fragmented identity systems make it difficult to enforce consistent authentication, authorization, and access policies.
- **Manual or Incomplete Compliance Checks**
  Without automation or standardized processes, ensuring continuous compliance becomes resource-intensive and error-prone.
- **Limited Visibility into Data Flows**
  Poor documentation or missing traces across microservices reduce transparency and complicate compliance reporting.

---

**System Implications**

For strong security posture and regulatory adherence, the architecture must incorporate security measures at every layer:

- **Security Built Into Service-Level Architecture**
  Every microservice must follow secure coding practices, standardized logging, proper secrets management, and encrypted communication (mTLS/TLS).
- **Centralized Identity and Access Management (IAM)**
  Implement single sign-on (SSO), token-based authentication (e.g., OAuth2, JWT), and unified RBAC/ABAC across all internal and external services.
- **Consistent, Structured Logging**
  Apply standardized log schemas, centralized log aggregation, and retention policies to support auditing and incident response.
- **Compliance Automation**
  Integrate automated checks for vulnerabilities, configuration drift, dependency scanning, and policy enforcement (e.g., OPA, security scanners).

- **Data Governance & Protection**
  Implement data classification, masking, retention rules, and lifecycle policies to ensure compliant handling of sensitive data across services.
- **Security Incident Response Processes**
  Establish workflows for alerting, investigation, escalation, and post-incident review, integrated with observability and SIEM tools.

## 2.5 External Systems & Integrations

External systems and third-party integrations play a critical role in extending the functionality of the marketplace platform. These systems handle essential capabilities such as payments, delivery updates, tax calculations, notifications, search, machine learning, and fraud detection. The marketplace ecosystem depends on these integrations to deliver a seamless, compliant, and scalable experience for both buyers and sellers.

Because these systems operate outside the core platform boundary, their interactions must be carefully designed for reliability, security, and smooth day-to-day operations.

---

**Key External Systems**

The primary categories of external integrations include:

- **Payment Gateways**
  Services such as Stripe, Razorpay, PayPal, or Adyen that manage transaction processing, settlements, chargebacks, and fraud scoring.
- **Logistics APIs**
  Third-party delivery and courier systems that retrieve shipping labels, update tracking statuses, and send delivery confirmations.
- **Tax Calculation Systems**
  Services (e.g., Avalara, GST APIs, VAT calculators) that compute taxes, duties, or region-specific compliance charges for products or orders.
- **SMS / Email Notification Providers**
  Platforms like Twilio, SendGrid, or AWS SNS/SES that send transactional alerts (order confirmations, OTPs, delivery updates) and marketing communications.
- **Search / ML Services**
  Engines such as Elasticsearch, OpenSearch, Pinecone, or managed vector databases that power product search, recommendations, personalization, and discovery features.
- **Fraud Detection APIs**
  External services that analyze transactions, user behavior, or historical patterns to identify high-risk activities and minimize revenue leakage.

---

**Integration Requirements**

To ensure smooth interaction with these external systems, the marketplace must support:

**1. Stable and Well-Defined APIs**

- Clear request/response structures
- Versioned endpoints
- Idempotent operations to avoid double processing
- Proper error codes and fallback mechanisms

Stable APIs minimize integration failures, simplify debugging, and support long-term compatibility.

**2. Secure Communication**

- Encrypted data transmission (HTTPS/TLS or mTLS)
- Key and secret rotation policies
- OAuth2 / API key authentication
- Strict access controls and IP whitelisting where applicable

Security is especially critical for payment, PII, and fraud-related integrations.

**3. Resilience and Retry Strategies**

- Retries with exponential backoff
- Dead-letter queues for persistent failures
- Circuit breakers to prevent cascading failures
- Timeout and rate-limiting configurations
- Eventual consistency handling for asynchronous callbacks

These measures ensure resilience even when third-party systems experience outages or latency spikes.

**4. Observability & Monitoring**

- Detailed logs of request/response interactions
- Integration-specific dashboards
- Alerting on failure rates, slowdowns, or error spikes
- Tracking of webhook health and callback delivery

Visibility into third-party behavior helps maintain operational stability.

**5. Failover & Graceful Degradation**

- Fallback tax formulas if tax systems fail
- Cached search results if search engines are down
- Queued notifications when email/SMS providers are unavailable
- Payment status reconciliation when callbacks fail

Graceful degradation ensures the platform remains partially functional under external failures.

---

**System Implications**

To support all external integrations effectively, the marketplace architecture must:

- Encapsulate each integration within **dedicated microservices** or adapters
- Use **asynchronous messaging** for high-volume or latency-sensitive operations
- Implement **API gateways** to centrally manage authentication, throttling, and routing
- Maintain **integration documentation and sandbox environments** for testing
- Ensure **auditability and traceability** for external calls and callbacks

## 2.6 User Personas

**2.6 User Personas**

Personas help model user behavior and define experience flows.

---

**Persona 1 — "Ravi", 26, Online Shopper**

**Goals:**

- Find best prices quickly
- Trust the platform
- Smooth checkout

**Behavior:**

- Uses mobile
- Applies multiple filters and sorts
- Looks at reviews before buying

**Needs from System:**

- Fast search
- Strong recommendations
- Transparent order tracking

---

**Persona 2 — "Priya", 33, Small Business Owner (Seller)**

**Goals:**

- Increase sales

- Upload products quickly
- Track performance metrics

**Behavior:**

- Updates inventory daily
- Checks analytics dashboard frequently

**Needs from System:**

- Streamlined seller portal
- Real-time inventory updates
- Automated settlement reports

---

**Persona 3 — "Arun", 40, Marketplace Administrator**

**Goals:**

- Ensure compliance
- Monitor tenant behavior
- Maintain platform uptime

**Behavior:**

- Uses admin dashboards
- Monitors fraud alerts
- Handles seller disputes

**Needs from System:**

- Centralized observability
- Role-based access control
- Audit event logs

# SECTION 3 — Requirements (Functional & Non-Functional Requirements)

## 3.1 Functional Requirements (FR)

**3.1.1 Tenant & Seller Management**

**FR-1: Tenant Onboarding**

- Sellers must be able to register as tenants.
- System must verify identity (KYC), bank details, and legal documents.

**FR-2: Tenant Configuration**

- Each tenant must have:
    - custom store name
    - branding (logo, theme, colors)
    - tax settings
    - shipping preferences

**FR-3: Tenant Isolation**

- Each seller's catalog, orders, customers, payments, and analytics must remain isolated from other sellers.

**FR-4: Role-Based Access**

- Seller admins can invite staff (managers, inventory operators).

---

**3.1.2 Product & Catalog Management**

**FR-5: Product Creation**

- Sellers must be able to create, edit, delete products with:
    - Title
    - Description
    - Images
    - Price
    - Category
    - Attributes

**FR-6: Bulk Upload**

- Sellers must be able to upload products in bulk (CSV, Excel, API).

**FR-7: Catalog Versioning**

- All updates must be versioned for rollback and audit.

**FR-8: Category Management**

- System admin must manage platform-wide categories.

**FR-9: Real-Time Search Index Updates**

- Whenever a product is added/updated, search index must update asynchronously.

---

**3.1.3 Inventory Management**

**FR-10: Stock Tracking**

- System must maintain stock count for every SKU.

**FR-11: Reserved Inventory**

- Inventory must be reserved during checkout until payment succeeds or fails.

**FR-12: Inventory Alerts**

- Sellers receive alerts when stock falls below threshold.

**FR-13: Multi-Warehouse Support**

- Inventory can be distributed across warehouses.

---

**3.1.4 Buyer Experience**

**FR-14: Buyer Registration**

- Buyers must sign up using email, phone, or social login.

**FR-15: Product Browsing**

- Buyers can browse by categories, filters, and sort.

**FR-16: Full Text Search**

- Buyers can search across all tenants.

**FR-17: Recommendations**

- Provide personalized product recommendations (homepage, PDP, cart).

**FR-18: Marketplace Cart**

- Cart persists across devices.
- Cart tracks multi-seller items.

**FR-19: Checkout Workflow**

- Multi-step checkout:
    - Address
    - Shipping method
    - Payment method

**FR-20: Order Confirmation**

- Buyers must receive order confirmation via SMS/email.
    - Price
    - Category
    - Attributes

**FR-6: Bulk Upload**

- Sellers must be able to upload products in bulk (CSV, Excel, API).

**FR-7: Catalog Versioning**

- All updates must be versioned for rollback and audit.

**FR-8: Category Management**

- System admin must manage platform-wide categories.

**FR-9: Real-Time Search Index Updates**

- Whenever a product is added/updated, search index must update asynchronously.

## 3.1.5 Order Management

**FR-21: Order Placement**

- System must atomically create order, reserve stock, process payment, and confirm.

**FR-22: Order Status Tracking**

Status transitions:

- Pending
- Confirmed
- Packed
- Shipped
- Delivered
- Cancelled
- Returned

**FR-23: Order Splitting**

- Multi-seller orders must split into separate sub-orders.

**FR-24: Returns & Refunds**

- Buyers can initiate return requests.
- Refunds processed via payment gateway.

## 3.1.6 Payment & Settlement

**FR-25: Payment Gateways**

- Support multiple gateways (Razorpay, Stripe, PayPal).

**FR-26: Payment Methods**

- Cards
- Net banking
- UPI
- Wallets

**FR-27: Retry Logic**

- If payment fails, user should retry without losing order state.

**FR-28: Seller Settlement**

- Platform must automatically settle payments to sellers after defined time.

---

## 3.1.7 Reviews & Ratings

**FR-29: Review Submission**

- Buyers can review purchased products.

**FR-30: Moderation**

- Auto moderation + admin approval pipeline.

---

## 3.1.8 Notifications

**FR-31: Notification Channels**

- Email
- SMS
- Push notifications

**FR-32: Event-Based Notifications**

Triggered on:

- Order placed
- Order shipped
- Order delivered
- Refund issued

---

## 3.1.9 Analytics & Dashboards

**FR-33: Seller Dashboard**

Shows:

- Sales
- Revenue
- Orders
- Product performance

**FR-34: Platform Metrics**

- Traffic
- Conversion
- Search performance
- Inventory health

---

# 3.2 Non-Functional Requirements (NFR)

---

## 3.2.1 Scalability Requirements

**NFR-1: Horizontal Scaling**

All critical services must scale under load.

**NFR-2: Search Scalability**

Search must support millions of SKUs.

**NFR-3: Event Throughput**

Kafka (or alternative) must support 50k events/sec for indexing, orders, recommendations.

---

## 3.2.2 Performance

**NFR-4: API Latency**

- 95th percentile response time < **150ms**
- 99th percentile < **250ms**

**NFR-5: Search Query Latency**

- <100ms for indexed products.

**NFR-6: Page Load Speed**

- Largest Contentful Paint < **2.5 seconds**

---

### 3.2.3 Availability & Reliability

**NFR-7: Uptime**

Platform availability must be **≥ 99.9%**.

**NFR-8: Fault Tolerance**

- Services must auto-restart on failures.
- Circuit breakers must isolate failing dependencies.

**NFR-9: DR & Backup**

- RPO: 15 minutes
- RTO: 1 hour

---

### 3.2.4 Security Requirements

**NFR-10: Data Encryption**

- All sensitive data encrypted at rest and transit.

**NFR-11: PCI-DSS Compliance**

- Payment-related microservices must meet compliance.

**NFR-12: Access Control**

- RBAC for internal and external users.

**NFR-13: Audit Logging**

- Every change must generate audit events.

---

### 3.2.5 Multi-Tenant Requirements

**NFR-14: Data Isolation**

- No tenant should access another's data.

**NFR-15: Namespace Partitioning**

- Search, cache, and database must support per-tenant partitions.

**NFR-16: Throttling**

- Each tenant receives configurable API rate limits.

---

### 3.2.6 Maintainability

**NFR-17: Microservice Independence**

- Each service independently deployable.

**NFR-18: Code Maintainability**

- Follow clean code principles & standardized API patterns.

**NFR-19: Monitoring Coverage**

- 100% coverage for:
  - logs
  - metrics
  - distributed tracing

---

### 3.2.7 Usability Requirements

**NFR-20: UX Standards**

- Platform must support user-friendly UI for buyers and sellers.

**NFR-21: Mobile Responsiveness**

- Optimize for Android & iOS browsers.

---

### 3.2.8 Observability Requirements

**NFR-22: Real-Time Metrics**

- Latency, error rate, throughput.

**NFR-23: Centralized Logs**

- All logs must be stored in ELK/CloudWatch/Splunk.

**NFR-24: Distributed Tracing**

- OpenTelemetry must trace every request.

# SECTION 4 — High-Level Architecture (with Diagrams & Explanations)

Professional, industry-grade, extremely detailed, and structured for your 50-page document.

## 4.1 Architecture Overview

The Multi-Tenant E-Commerce Marketplace follows a **cloud-native, microservices-based, event-driven architecture**.

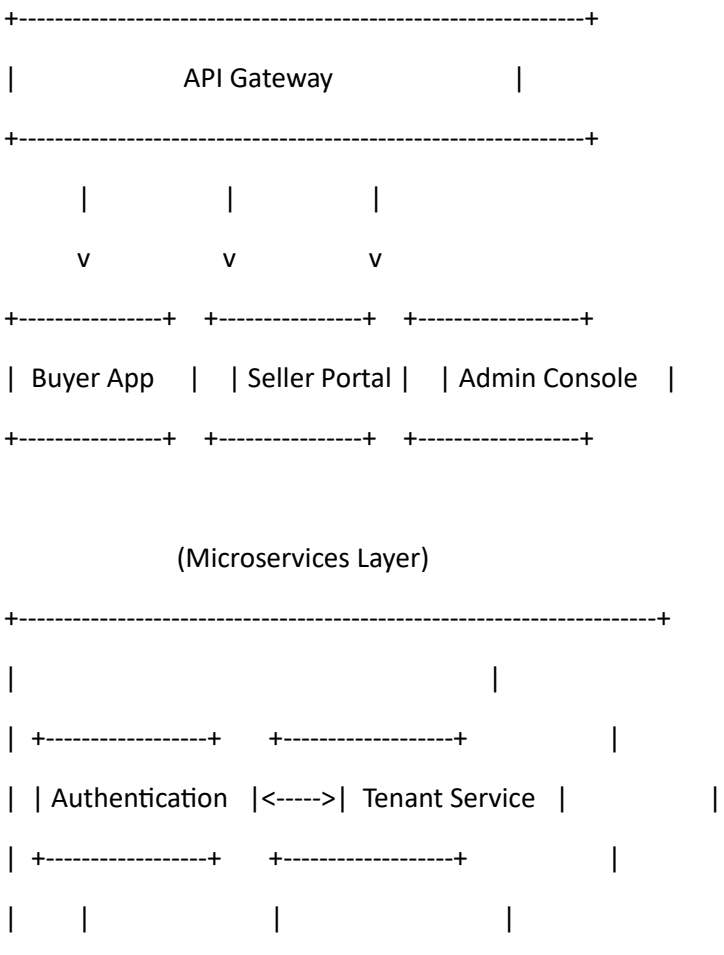
The goals of the architecture are:

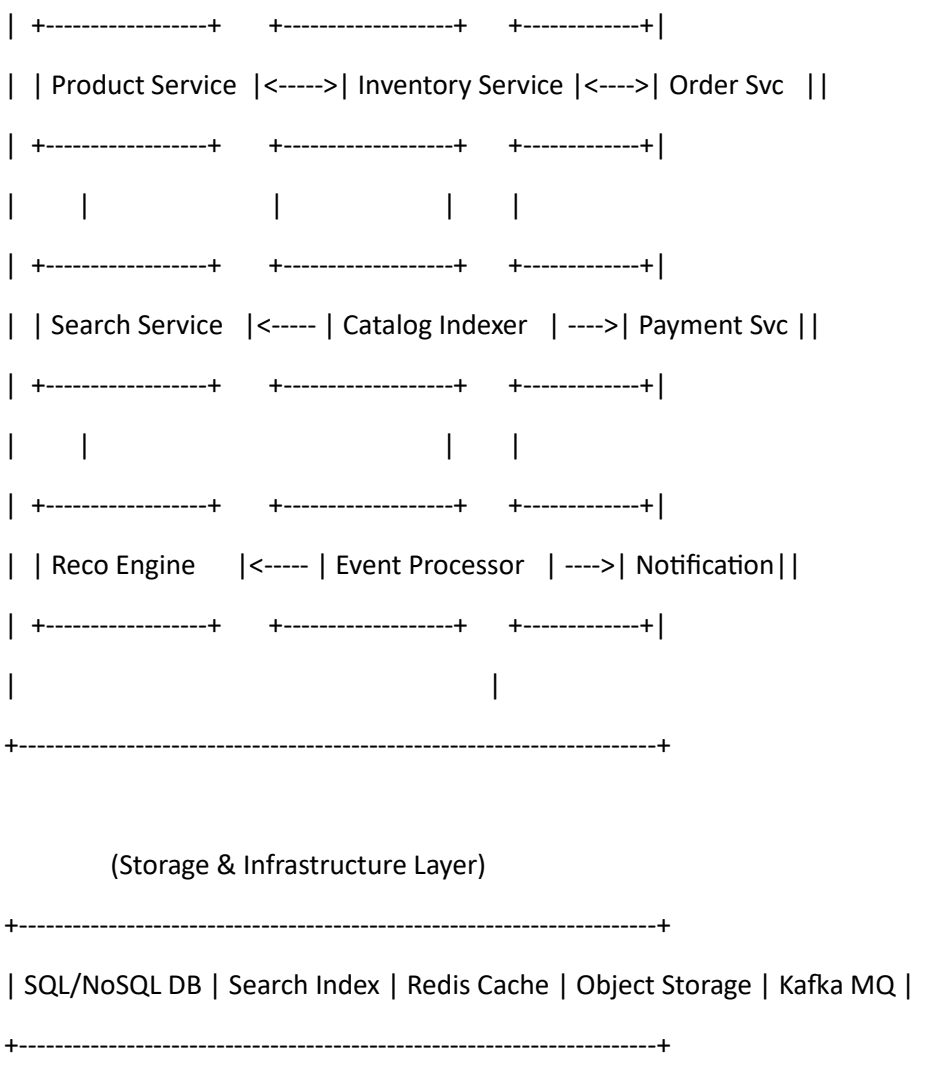
- High scalability
- Tenant isolation
- Independent service deployment
- Real-time updates (inventory, indexing, recommendations)
- High availability through distributed systems
- Strong observability across all services

---

## 4.2 Logical Architecture Diagram

Below is the **text-based logical architecture diagram**, suitable for documentation.
(In DOCX, I will convert it into a proper graphic diagram.)






---

## 4.3 Microservices Breakdown

**Core Services**

| Service | Description |
|---|---|
| **Tenant Service** | Handles onboarding, isolation rules, store branding. |
| **Product Service** | Stores product info, categories, variants. |
| **Inventory Service** | Tracks stock, SKU updates, reservations. |
| **Order Service** | Manages full order lifecycle, splitting, events. |
| **Payment Service** | Integrates with Razorpay/Stripe; settlement engine. |
| **Search Service** | Handles search queries, autocomplete, filtering. |
| **Search Indexer** | Updates Elasticsearch/OpenSearch index. |

| Service | Description |
| --- | --- |
| **ML Recommendation Engine** | Provides personalized suggestions. |
| **Notification Service** | SMS/Email/Push events. |
| **Auth Service** | JWT, OAuth2, buyer/seller/admin roles. |

**Supporting Services**

- Event Processor (Kafka consumers)
- Fraud Detection module
- Logging service
- Metrics exporter
- CDN & media storage service

---

## 4.4 Multi-Tenant Architecture Model

There are three approaches to multi-tenancy:

**Model A: Shared Database, Tenant-ID Based Partitioning**

- All tenants share same DB tables.
- Tenant ID acts as partition key.

**Pros:** Low cost, easy scaling
**Cons:** Higher risk if tenant isolation rules fail

**Model B: Separate Schemas per Tenant (Recommended)**

- Each tenant has its own schema:
    - Products
    - Orders
    - Inventory
    - Reviews

**Pros:**

- Strong data isolation
- Query performance improves
- Easier auditing

**Cons:**

- More complex schema migrations

**Model C: Separate Databases per Tenant**

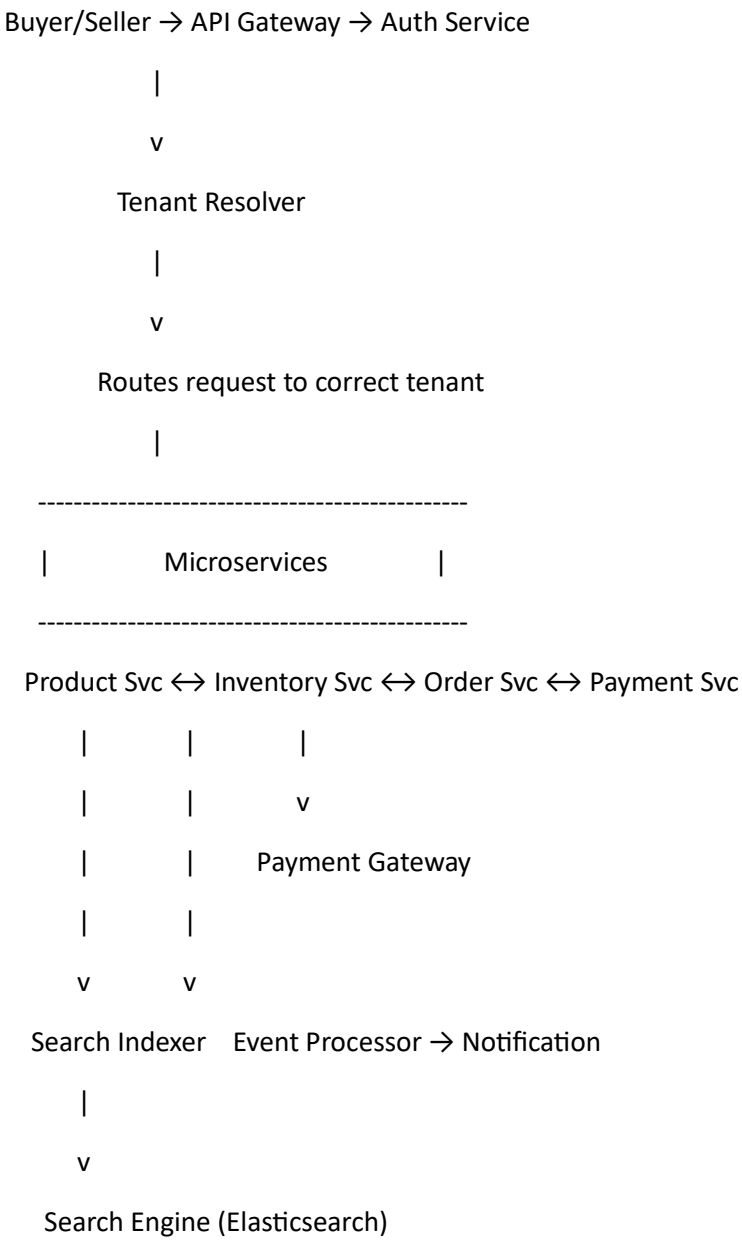
- Used by enterprise systems (like Shopify Plus).

  **Pros:** Maximum security
  **Cons:** Expensive, hard to scale for many tenants

  **Chosen Model for This System:**
  ➡️ **Model B: Schema-level Tenant Isolation**

---

## 4.5 Detailed Component Interaction Diagram




---

## 4.6 Sequence Diagram (Add-to-Cart & Checkout)

Buyer → API Gateway: Add item to cart

API Gateway → Cart Service: store item

Cart Service → Inventory Service: validate stock availability

Inventory Service → Cart Service: stock OK

Buyer → Checkout: place order

Checkout → Order Service: create order

Order Service → Inventory Service: reserve stock

Order Service → Payment Service: initiate payment

Payment Service → Payment Gateway: process

Payment Gateway → Payment Service: success

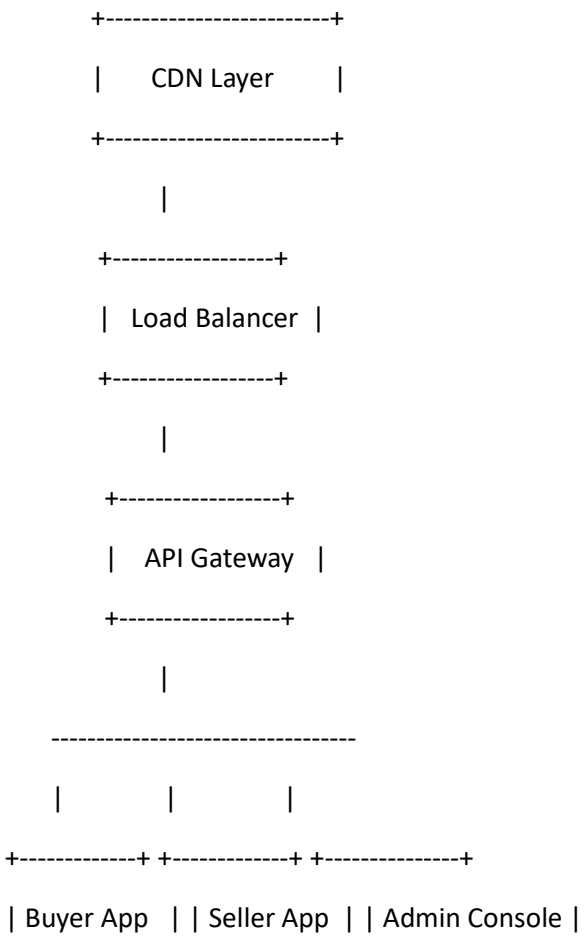
Payment Service → Order Service: confirm order

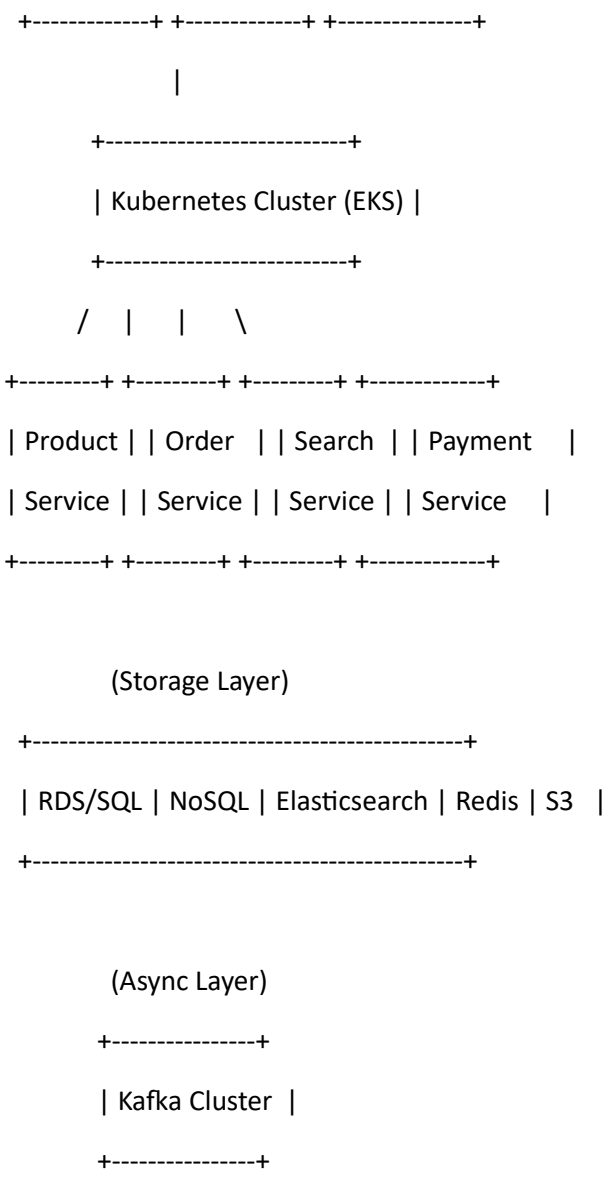
Order Service → Inventory Service: reduce stock

Order Service → Notification: send confirmation

---

## 4.7 Physical Architecture (Cloud Deployment)






---

## 4.8 Architecture Style & Principles Applied

### 4.8.1 Microservices Architecture

Each service is independently deployable and scalable.

**4.8.2 Event-Driven Architecture**

Used for:

- Order updates
- Inventory reservation
- Search indexing
- Analytics pipeline

**4.8.3 Domain-Driven Design (DDD)**

Bounded contexts:

- Catalog
- Inventory
- Checkout
- Payment
- Search
- Recommendation

**4.8.4 Twelve-Factor App Principles**

Mandatory for cloud-native deployment.

**4.8.5 API-First Approach**

Every service exposes REST APIs with versioning.

**4.8.6 High Observability**

- Centralized logging
- Distributed tracing (OpenTelemetry)
- Real-time metrics

---

## 4.9 Key Trade-Offs

**Trade-Off 1: SQL vs NoSQL**

SQL chosen for:

- Transactional consistency
- Complex queries
- ACID operations

NoSQL used for:

- Search
- Recommendations
- Caching

**Trade-Off 2: Sync vs Async Communication**

Sync for:

- Cart operations
- Checkout

Async for:

- Indexing
- Events
- Notifications

**Trade-Off 3: Tenant Isolation Model**

Schema-level isolation chosen for best balance of:

- Performance
- Cost
- Security

# SECTION 5 — Microservices & Component Design

## 5.1 Overview

This section describes each major microservice, its responsibilities, data model highlights, APIs, scaling & deployment guidance, fault-tolerance patterns, observability, security considerations, and interactions with other services. Services are designed as independently deployable units following Domain-Driven Design (DDD) bounded contexts. Communication is primarily REST + gRPC for low-latency internal calls, and event-driven (Kafka) for asynchronous workflows.

Core microservices covered:

- Tenant Service
- Auth Service
- Product Service
- Catalog/Search Indexer
- Inventory Service
- Cart Service
- Order Service
- Payment Service
- Recommendation (Reco) Service
- Notification Service
- Analytics / Data Ingestion Service
- Fraud Detection Service
- Admin / Billing Service

---

## 5.2 Service: Tenant Service

**Purpose:** Manage tenant lifecycle (onboarding, configuration, billing plans, tenant metadata, tenant-specific feature toggles).

**Responsibilities**

- Tenant registration & KYC orchestration
- Tenant configuration storage (branding, tax, shipping)
- Tenant isolation metadata (schema mapping, quotas, rate limits)
- Tenant lifecycle operations (suspend, delete, upgrade plan)

**Data model (high-level)**

- tenant_id (UUID)
- name, contact_info, legal_docs, status
- schema_name or schema_version
- plan (free/pro/enterprise)
- quotas (api_calls_per_minute, products_limit)

**APIs (examples)**

- POST /v1/tenants — register tenant (returns tenant_id)
- GET /v1/tenants/{tenant_id} — tenant details
- PUT /v1/tenants/{tenant_id}/config — update config
- POST /v1/tenants/{tenant_id}/verify — trigger KYC process

**Scaling & deployment**

- Low write rate, moderate read rate; single primary replica with read replicas.
- Horizontally scalable stateless API layer, persists to RDS; use connection pooling.
- Cache tenant metadata in Redis for fast tenant resolution at API Gateway.

**Failure modes & mitigation**

- KYC provider outage: implement retry with exponential backoff, queue KYC requests in Kafka.
- Loss of tenant metadata cache: fallback to DB; warm cache on startup.

**Security**

- Strong RBAC for admin operations
- Sensitive tenant docs stored encrypted in object store (S3) with access logs

**Observability**

- Metrics: tenant_creations, tenant_updates, tenant_suspensions

- Traces: tenant onboarding flow

---

## 5.3 Service: Auth Service

**Purpose:** Authentication and authorization for buyers, sellers, and platform admins.

**Responsibilities**

- Issue and validate JWTs / opaque tokens
- OAuth2 support for social login
- Refresh tokens, password reset
- RBAC/ABAC enforcement integration

**Data model**

- user_id, tenant_id (nullable for buyers), roles, permissions, password_hash, mfa_enabled

**APIs**

- POST /v1/auth/login
- POST /v1/auth/refresh
- POST /v1/auth/logout
- GET /v1/auth/userinfo

**Design choices**

- Stateless JWTs for scale; keep short expiry and support token revocation via a token blacklist in Redis.
- Use Authorization middleware at API Gateway for protected endpoints.

**Security**

- Password hashing (argon2id), rate-limit auth attempts, support MFA, store secrets in KMS.

**Observability**

- Failed login rate, auth latency, token issuance counters.

---

## 5.4 Service: Product Service

**Purpose:** CRUD for products, validation, media attachment metadata, categories, attribute definitions.

**Responsibilities**

- Product creation, updates, soft-deletes
- Versioning & audit trail for product changes

- Manage variants and attribute schemas
- Emit product events to Kafka for indexing and analytics

**Data model (simplified)**

- product_id, tenant_id, title, description, attributes (jsonb), skus: [{sku_id, price, weight, dimensions, images}], category_id, status, created_at, updated_at

**APIs**

- POST /v1/products — create product
- GET /v1/products/{id} — fetch product
- PUT /v1/products/{id} — update
- POST /v1/products/bulk — bulk upload

**Indexing & integration**

- On product create/update -> publish product.updated event (topic: products.v1) consumed by Search Indexer & Analytics

**Scaling**

- CPU-light writes; scale API horizontally; use DB partitioning by tenant schema or partition key.
- Use S3 for images and CDN; store only image metadata in Product DB.

**Fault tolerance**

- If indexer is down, write events to Kafka with retention for replay.

**Observability**

- product_upserts, product_reads, bulk_job_durations

---

## 5.5 Service: Search Indexer & Search Service

**Purpose:** Maintain full-text search indices and serve search queries (facets, suggestions, autosuggest, ranking).

**Components**

- Indexer (consumer): consumes product.updated, inventory.updated, price.updated events and updates index.
- Search API: query layer exposing search endpoints, caching common queries.

**Technology**

- Elasticsearch / OpenSearch / OpenSearch Service. Consider managed cluster with hot/warm nodes.

**Index design**

- Per-tenant index prefix or tenant field in documents (depending on isolation model).
- Use nested mappings for variants/SKUs.
- Analyze fields for stemming, n-grams for suggestions.

**APIs**

- GET /v1/search?q=...&filters=...&sort=...
- GET /v1/search/suggest?q=... (autocomplete)

**Scaling**

- Separate read (query) and write (indexing) concerns: scale query nodes; dedicate indexing nodes.
- Use index aliases for zero-downtime reindexing.

**Consistency**

- Near real-time indexing. Use refresh_interval tuned to traffic — e.g., 1s for active clusters.

**Performance optimizations**

- Result caching in Redis for frequently repeated queries.
- Shard by category or by tenant prefix to reduce shard hotness.

**Failure modes**

- Index corruption / mapping errors: maintain snapshot backups and automated reindex strategy.
- High write load: backpressure via Kafka and batch indexing.

**Observability**

- query_latency_histogram, index_lag (events to index), cache_hit_ratio

---

## 5.6 Service: Inventory Service

**Purpose:** Real-time stock tracking, reservations, multi-warehouse support, stock reconciliation.

**Responsibilities**

- Maintain stock levels per SKU per location
- Reserve stock for ongoing checkouts
- Release reservations on timeout or order cancellation
- Sync with external warehouse systems

**Data model**

- sku_id, tenant_id, warehouse_id, available_qty, reserved_qty, incoming_qty

**APIs**

- GET /v1/inventory/{sku_id}
- POST /v1/inventory/{sku_id}/reserve {quantity, order_id, reservation_ttl}
- POST /v1/inventory/{sku_id}/release {reservation_id}
- POST /v1/inventory/adjust (manual corrections)

**Concurrency control**

- Use pessimistic locking or optimistic CAS depending on DB:
    - For high contention, use distributed locks (Redis Redlock) for per-sku operations.
    - Alternative: use event-sourced inventory model with idempotent handlers.

**Integration with Order Service**

- Order Service calls reserve during checkout; upon payment success, Order Service confirms and requests decrement.

**Scaling**

- Partition inventory by SKU hash or tenant, scale horizontally.
- Use CQRS: keep an authoritative write store and denormalized read model for queries.

**Reliability**

- Periodic reconciliation jobs to compare warehouse feeds with DB.

**Observability**

- inventory_reservations, reservation_timeouts, reconciliation_mismatch_count

---

## 5.7 Service: Cart Service

**Purpose:** Manage buyer carts (session persistence, multi-seller items).

**Responsibilities**

- Create & update carts, persist across devices
- Validate price & availability during checkout
- Support abandoned cart notifications (via Notification Service)

**Data model**

- cart_id, user_id, tenant_id, items: [{sku_id, qty, price_snapshot, reserved_id?}], updated_at, expires_at

**Design**

- Cache-first approach: store carts in Redis for performance with periodic durable writes to a document DB for long-term persistence.

- On checkout, verify prices and availability by calling Product & Inventory.

**APIs**

- GET /v1/carts/{cart_id}
- POST /v1/carts/{cart_id}/items
- POST /v1/carts/{cart_id}/checkout (hand-off to Order Service)

**Edge cases**

- Price changes between add-to-cart and checkout: apply price policy (lock price for X minutes, or reprice at checkout with buyer notification).

**Scaling**

- Very high read/write; Redis cluster with persistence (AOF) for durability.

**Observability**

- cart_abandon_rate, cart_checkout_conversion

---

## 5.8 Service: Order Service

**Purpose:** Orchestrate the full lifecycle of orders: creation, splitting (multi-seller), fulfillment, cancellations, returns, and settlements.

**Responsibilities**

- Create atomic order construct (order + sub-orders per seller)
- Coordinate inventory reservation, payment initiation, shipment initiation
- Maintain order state machine & ensure idempotency
- Publish order events for downstream consumers (notifications, analytics, fulfillment)

**Order lifecycle states**

- CREATED → RESERVED → PAYMENT_PENDING → PAID → PACKED → SHIPPED → DELIVERED → RETURN_REQUESTED → REFUNDED/CLOSED

**APIs**

- POST /v1/orders — create order (idempotency-key header supported)
- GET /v1/orders/{order_id}
- POST /v1/orders/{order_id}/cancel
- POST /v1/orders/{order_id}/capture (for delayed capture flows)

**Transactions & Consistency**

- Use Saga pattern for distributed transactions:
    - Step 1: create order record (COMPENSATABLE)

    - Step 2: reserve inventory (compensate release)
    - Step 3: initiate payment (compensate refund/cancel)
    - Step 4: confirm order, emit order.completed
- Implement outbox pattern: persist outgoing events in the same DB transaction, then async deliver to Kafka.

**Idempotency**

- All externally visible commands accept an Idempotency-Key to avoid double-charges and duplicate orders.

**Scaling**

- Partition by tenant or by seller to scale order throughput.
- Use Kafka for downstream processing (fulfillment, notification).

**Fault handling**

- If payment fails → saga triggers inventory release & order canceled.
- For partial failures (some sub-orders succeed), keep per-suborder reconciliation & user-visible status.

**Observability**

- order_creation_rate, saga_failure_rate, order_fulfillment_latency

---

## 5.9 Service: Payment Service

**Purpose:** Abstract payment provider integrations, handle authorizations, captures, refunds, settlement scheduling.

**Responsibilities**

- Provide unified payment API over multiple PSPs
- Tokenize payment methods (do not store raw cards)
- Handle 3DS flows, UPI redirections, webhooks
- Manage settlement schedules & reconciliation with bank statements

**APIs**

- POST /v1/payments/authorize {order_id, amount, payment_method}
- POST /v1/payments/capture {authorization_id}
- POST /v1/payments/refund {payment_id, amount}
- POST /v1/payments/webhook (PSP -> platform)

**Security & Compliance**

- Isolate PCI surface area: host payment integrations in a dedicated VPC/subnet and use tokenization.
- Use PSP-managed tokens where possible (Stripe tokens, Razorpay tokens).
- Handle sensitive data via vaults, and ensure regular PCI audits.

**Resilience**

- Implement retry and idempotent webhook handlers.
- Use exponential backoff for PSP transient errors, and fallback PSP if configured.

**Observability**

- payment_success_rate, chargeback_rate, payment_latency

---

## 5.10 Service: Recommendation (Reco) Service

**Purpose:** Provide personalized and contextual product recommendations (homepage, PDP, cart, post-purchase).

**Components**

- Real-time feature store / event consumer (Kafka)
- Offline training pipelines (Spark/Databricks)
- Serving infra (model API endpoints / low-latency vector DB)

**Architecture**

- Events (pageviews, clicks, add-to-cart, purchases) -> Kafka -> Feature generator & offline store
- Training pipeline produces models (collaborative filtering, item2vec, matrix factorization, or transformer-based)
- Serving: use a vector database (Pinecone / Milvus) or approximate nearest neighbors (Faiss) for embeddings and provide ranked lists.

**APIs**

- GET /v1/recommendations?user_id=...&context=homepage
- POST /v1/recommendations/batch (for carousel generation)

**Latency & Scalability**

- Cache top recommendations per user in Redis.
- Batch precompute recommendations for active users to reduce compute.

**Evaluation**

- A/B testing harness integrated (traffic split, metric tracking).
- Offline metrics: MRR, Precision@K, Recall@K; online: CTR uplift, conversion lift.

**Observability**

- recommendations_served, model_latency, model_freshness

---

## 5.11 Service: Notification Service

**Purpose:** Send transactional & promotional notifications (email, SMS, push) and provide templating.

**Responsibilities**

- Template management
- Channel adapters (SendGrid, Twilio, Firebase)
- Retry & dead-lettering for undeliverable messages
- Subscription preferences (do-not-disturb windows)

**APIs**

- POST /v1/notifications/send {channel, template_id, recipient, payload}

**Design**

- Event-driven: consumes notification events (order.confirmation, shipment.update) and dispatches to channels.
- Use prioritized queues for transactional messages.

**Failure handling**

- Use DLQ for persistent failures; provide admin UI to retry.

**Observability**

- notifications_sent, delivery_rate_by_channel, bounce_rate

---

## 5.12 Service: Analytics / Data Ingestion

**Purpose:** Collect event data for analytics, reporting, ML pipelines, and data warehouse.

**Architecture**

- Event producers (microservices) publish to Kafka.
- Stream processing (Flink / Spark Streaming / Kafka Streams) transforms and writes to:
    - OLAP store (Redshift / BigQuery / Snowflake)
    - Feature store for ML
    - Time-series DB for metrics (Prometheus + long-term storage)

**Responsibilities**

- Data validation & schema registry (Avro/Protobuf)
- Data lineage tracking
- Snapshot & incremental ETL

**Observability**

- events_ingested_per_minute, pipeline_lag, failed_records

---

## 5.13 Service: Fraud Detection Service

**Purpose:** Real-time and batch fraud scoring based on heuristics & ML models.

**Responsibilities**

- Real-time scoring during checkout
- Flag suspicious transactions for review
- Provide rules engine for admins (thresholds, auto-block)

**Integration**

- Called by Payment & Order Service synchronously during checkout
- Emits events for manual review queue

**Design**

- Combine rule-based checks (velocity, IP anomalies) with ML risk score
- Maintain a blacklist/whitelist DB

**Observability**

- fraud_score_distribution, blocked_transactions

---

## 5.14 Service: Admin & Billing Service

**Purpose:** Admin UI backend for platform configuration, dispute management, billing, and settlements.

**Responsibilities**

- Billing and commission calculations
- Settlement schedule and remittance to sellers
- Refund approvals & reversal flows
- Admin audit trails and reports

**APIs**

- GET /v1/billing/settlements?tenant_id=...

- POST /v1/admin/settlement/trigger

**Accounting**

- Use double-entry ledger microservice pattern for cash flows to ensure auditability.

---

## 5.15 Cross-Cutting Concerns

**5.15.1 Eventing & Message Contracts**

- All event payloads must follow schemas (Avro/Protobuf) and be versioned.
- Use topics like: products.v1, inventory.v1, orders.v1, payments.v1, user.events.v1, analytics.events.v1.
- Publish-subscribe semantics; critical events use at-least-once delivery and consumers must be idempotent.

**5.15.2 Outbox Pattern**

- Services that emit events persist outbox records in the same DB transaction that changed state. A dedicated deliverer extracts outbox rows and pushes to Kafka to guarantee consistency.

**5.15.3 Saga Orchestration**

- Use a coordinator (Order Service) for long-running transactions or choreographed sagas via events for decentralized coordination. Maintain compensation handlers for each step.

**5.15.4 Idempotency & Exactly-Once Considerations**

- Expose idempotency-key header on write endpoints (orders, payments) and store idempotency metadata to avoid duplicates.
- Consumers should deduplicate by event id.

**5.15.5 Security**

- All services authenticate via mTLS or mutual TLS between services and use short-lived service tokens from identity provider.
- Secrets in KMS (AWS KMS, Azure Key Vault); rotate keys regularly.

**5.15.6 Observability & Tracing**

- Trace context passed through HTTP/gRPC and Kafka messages (W3C traceparent).
- Use OpenTelemetry for spans and propagate sampled traces to Jaeger/Tempo.
- Centralized logs (ELK / CloudWatch / Splunk) with structured JSON logging including tenant_id and correlation IDs.

**5.15.7 Rate Limiting & Throttling**

- Enforce per-tenant and global quotas at API Gateway (rate-limiting by token bucket).

- Provide feedback headers (X-RateLimit-Remaining) and graceful 429 responses with Retry-After.

**5.15.8 Backpressure Patterns**

- Use circuit breakers and bulkheads in API clients.
- When downstream services degrade, degrade gracefully (e.g., return cached search results, simplified recommendations).

---

## 5.16 API Design Guidelines (Cross-Service)

- Use consistent versioning format /v1/...
- Request/response schemas defined via OpenAPI (Swagger) and auto-generated client SDKs.
- Use HTTP status codes properly: 200/201/202, 400, 401, 403, 404, 409 (conflict), 422 (validation), 429 (rate limit), 500 (server).
- Support filtering, pagination (cursor-based), and sorting on list endpoints.
- Accept Accept: application/json and support gzip compression for responses.
- Include X-Correlation-ID header to trace requests across services.

---

## 5.17 Deployment & CI/CD Patterns per Service

- Each service packaged as Docker container; use Kubernetes (EKS/GKE/AKS) for orchestration.
- Pipelines:
    - Build: lint → unit tests → security scans (SAST) → container build
    - Deploy: canary or blue/green deployments, automated smoke tests, promote to production after health checks
- Use GitOps model (ArgoCD/Flux) for declarative infra.
- Feature flags (LaunchDarkly or internal) to toggle features per tenant.

---

## 5.18 Testing Strategy

- Unit tests for business logic
- Integration tests for service interactions using test containers or staging clusters
- Contract testing using Pact for service-to-service API contracts
- End-to-end tests for critical flows (checkout)
- Chaos engineering exercises for resilience (simulate service failures and network partitions)

---

## 5.19 Monitoring & SLOs

- Define SLOs per service (e.g., Order Service 99.95% successful order placement over 30 days).
- Use Prometheus for metrics scraping; Grafana for dashboards.
- Central alerting with escalation runbooks; noise reduction via alert thresholds and suppression windows.

---

## 5.20 Example Interaction: Checkout Sequence (detailed)

1. Buyer starts checkout → Cart Service validates cart content.
2. Cart Service calls Inventory Service to reserve items (idempotent reserve).
3. On success, Cart Service calls Order Service to create order (idempotency-key).
4. Order Service persists order and writes outbox event order.created.
5. Order Service triggers Payment Service with authorize call.
6. Payment Service calls PSP; receives callback webhooks which are reconciled to payment.succeeded or payment.failed.
7. On payment.succeeded, Order Service moves state to PAID, emits order.paid, and triggers fulfillment notifications. Inventory Service is asked to commit reserved stock (decrement).
8. Index updates and analytics events are published asynchronously.

   All steps have compensating actions: e.g., if payment fails after reservation, reserve is released and order marked canceled. Each external call must be idempotent and have retry/backoff.

---

## 5.21 Data Residency & GDPR Considerations

- Tenant-configurable data residency (store tenant data in region-specific schemas/databases when requested).
- Provide GDPR features: right to erasure, data export endpoints, audit logs for data access.
- Pseudonymize personally identifiable information (PII) in analytics streams.

---

## 5.22 Summary & Next Steps

This section defined microservice responsibilities, APIs, data contracts, scaling, fault tolerance, and operational practices. The next deliverable (Section 6) will provide **Database Design** with ER diagrams, schema definitions, indexing strategy, sharding plan, and example DDL for core tables (Products, Orders, Inventory, Tenants). It will also include migration approaches and backup/restore strategy.

# SECTION 6 — Database Design (ERD, Schema, Partitioning, Indexing, and Storage Strategy)

## 6.1 Overview of Database Strategy

A multi-tenant e-commerce marketplace requires a **hybrid database approach** to balance:

- Strong ACID guarantees (orders, payments, inventory)
- Fast search and filtering (search index)
- Scalable catalog and analytic workloads (NoSQL + event storage)
- Tenant isolation (schema-level or partition-level)

The system uses a **polyglot persistence model**:

| Use Case | Recommended Storage |
|---|---|
| Orders, Payments, Tenants, Inventory | Relational DB (PostgreSQL / MySQL / Aurora) |
| Search & Filtering | Elasticsearch / OpenSearch |
| Product Catalog (read-heavy) | SQL + Redis caching + Search Index |
| Event Logs & Analytics | Kafka + Data Lake (S3) |
| Recommendations | Vector DB (Pinecone / Milvus) |
| Sessions, Carts | Redis Cluster |

---

## 6.2 Multi-Tenant Isolation Strategy

We adopt **Schema-Level Multi-Tenancy**, where:

- Each tenant gets its own schema:
  tenant_123.products, tenant_123.orders, tenant_123.inventory
- Shared metadata tables remain global:
  tenants, users, payment_providers

**Why Schema-Level?**

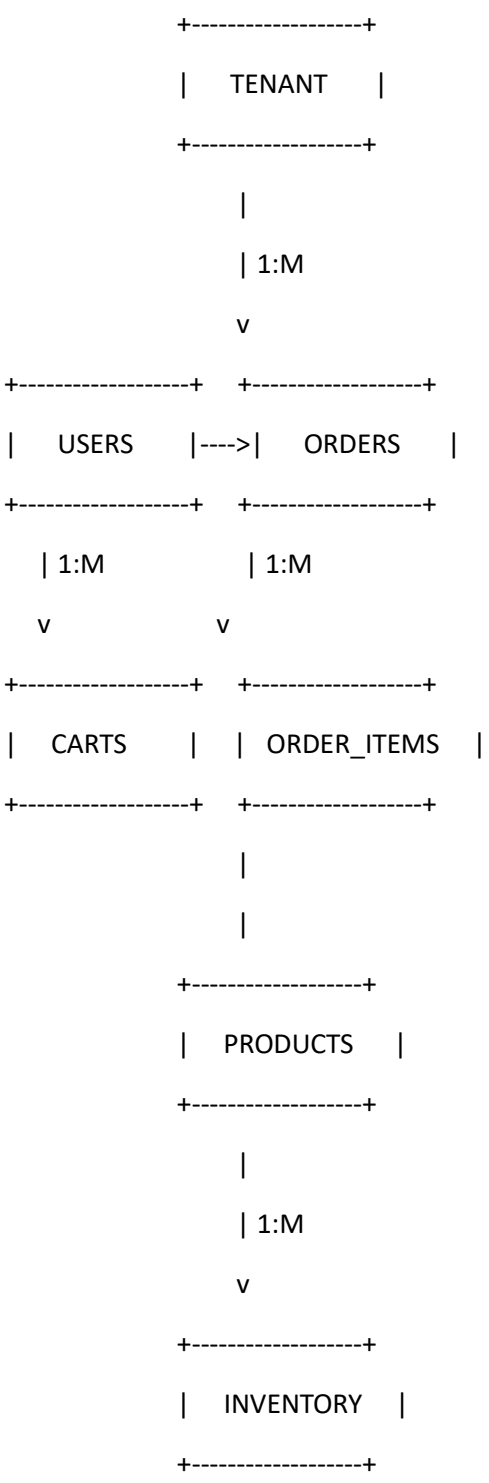
- Stronger isolation than row-level partitioning
- Lower cost and easier than separate DB per tenant
- Simplifies GDPR processes
- Query performance improved by smaller tables

This is the same model used by Shopify and Lightspeed Commerce.

---

## 6.3 High-Level ER Diagram (Conceptual)

Below is a text-based ERD (converted into diagram in DOCX):




Another extended conceptual diagram:

TENANTS → SELLERS → PRODUCTS → SKUS → INVENTORY

| | | |

| | | → PRICE_HISTORY

|  |

→ SETTINGS  → PRODUCT_ATTRIBUTES

→ BILLING  → MEDIA

---

## 6.4 Detailed Table Designs

### 6.4.1 Tenants Table (Global Schema)

| Column | Type | Description |
|---|---|---|
| tenant_id | UUID | Primary key |
| tenant_name | VARCHAR | Seller name |
| schema_name | VARCHAR | Schema for this tenant |
| onboarded_at | TIMESTAMP | Registration date |
| status | ENUM(active, suspended, deleted) | Tenant lifecycle status |
| plan | ENUM(free, pro, enterprise) | Billing level |
| api_rate_limit | INT | Per-tenant request limit |

**Indexes**

- idx_tenant_status
- idx_tenant_schema_name

---

### 6.4.2 Users Table (Global or Tenant-Specific)

| Column | Type | Notes |
|---|---|---|
| user_id | UUID | PK |
| tenant_id | UUID | nullable for buyers |
| role | ENUM(buyer, seller_admin, seller_staff, platform_admin) | |
| email | VARCHAR | Unique |
| phone | VARCHAR | Optional |
| password_hash | VARCHAR | Argon2 |
| created_at | TIMESTAMP | |

**Indexes:**

- unique(email)
- idx_users_tenant_role

---

**6.4.3 Products Table (Tenant Schema)**

| Column | Type | Notes |
|---|---|---|
| product_id | UUID | PK |
| title | TEXT | Indexed for search |
| description | TEXT | |
| category_id | INT | FK |
| attributes | JSONB | Dynamic attributes |
| status | ENUM(active, inactive, draft) | |
| created_at | TIMESTAMP | |
| updated_at | TIMESTAMP | |

**Indexes**

- GIN(attributes)
- BTREE(category_id)
- GIN(to_tsvector(title || description))

---

**6.4.4 SKUs Table (Tenant Schema)**

| Column | Type | Notes |
|---|---|---|
| sku_id | UUID | PK |
| product_id | UUID | FK |
| price | DECIMAL | Latest price |
| currency | VARCHAR | |
| weight | FLOAT | |
| barcode | VARCHAR | Unique optional |
| image_urls | JSONB | Array of image links |

---

#### 6.4.5 Inventory Table

| Column | Type | Notes |
|---|---|---|
| inventory_id | UUID | PK |
| sku_id | UUID | FK |
| warehouse_id | UUID | FK |
| available_qty | INT | |
| reserved_qty | INT | |
| incoming_qty | INT | |
| updated_at | TIMESTAMP | |

**Indexes**

- idx_inventory_sku_warehouse
- idx_inventory_available

---

#### 6.4.6 Carts Table (Redis-first, SQL-persistence)

| Field | Description |
|---|---|
| cart_id | UUID |
| user_id | UUID |
| tenant_id | UUID |
| items | JSONB → [{sku_id, qty, price_snapshot}] |
| updated_at | TIMESTAMP |

Stored in Redis for speed but periodically written to SQL.

---

#### 6.4.7 Orders Table

| Column | Type | Notes |
|---|---|---|
| order_id | UUID | PK |
| buyer_id | UUID | FK |
| tenant_id | UUID | FK |
| total_amount | DECIMAL | |

| Column | Type | Notes |
|---|---|---|
| payment_status | ENUM(pending, success, failed) | |
| order_status | ENUM(created, packed, shipped, delivered, returned) | |
| created_at | TIMESTAMP | |
| updated_at | TIMESTAMP | |

---

#### 6.4.8 Order Items Table

| Column | Type | Notes |
|---|---|---|
| order_item_id | UUID | PK |
| order_id | UUID | FK |
| sku_id | UUID | FK |
| quantity | INT | |
| price_at_purchase | DECIMAL | Snapshot |

**Indexes**

- idx_order_items_order_id

---

#### 6.4.9 Payments Table

| Column | Type | Notes |
|---|---|---|
| payment_id | UUID | PK |
| order_id | UUID | FK |
| provider | ENUM(razorpay, stripe) | |
| amount | DECIMAL | |
| status | ENUM(init, pending, success, refund, failed) | |
| transaction_reference | VARCHAR | PSP reference |
| created_at | TIMESTAMP | |

---

#### 6.4.10 Reviews Table

| Column | Type | Notes |
|---|---|---|
| review_id | UUID | PK |
| product_id | UUID | FK |
| buyer_id | UUID | FK |
| rating | INT (1-5) | |
| comment | TEXT | |
| created_at | TIMESTAMP | |

---

## 6.5 Indexing Strategy

Proper indexing is critical for a scalable marketplace.

#### 6.5.1 Products

- GIN Full-text index for search:
  GIN(to_tsvector('english', title || ' ' || description))
- JSONB index for attributes:
  GIN(attributes)

#### 6.5.2 Inventory

- Composite index:
  (sku_id, warehouse_id)
- Index on available_qty for fast filtering.

#### 6.5.3 Orders

- Tenant + date index → fast analytics queries
  (tenant_id, created_at)

#### 6.5.4 Users

- Unique index on email
- Role index for permission queries

#### 6.5.5 Payments

- Index on transaction_reference
- Index on status for reconciliation jobs

---

## 6.6 Sharding Strategy (Horizontal Partitioning)

Sharding is required to support thousands of tenants and millions of SKUs.

**Sharding dimensions:**

**Option A: Tenant-based Sharding**

Distribute tenants across multiple database clusters.

**Pros:** Strong isolation
**Cons:** Hotspot risk for large tenants

**Option B: Catalog-based Sharding**

Shard by:

- Category
- Product ID hash
- SKU hash

**Pros:** Distributes load evenly
**Cons:** Harder to manage multi-seller queries

**Option C: Time-based Partitioning (for Orders & Analytics)**

Partition by month or quarter:

- Faster queries
- Easier archival
- Reduced index bloat

**Final Strategy Used:**

- **Tenant sharding for transactional services**
- **Time-based partitions for orders/payments**
- **Category/SKU sharding for large product catalogs**

---

## 6.7 Read & Write Optimization

**Write Optimization**

- Use batch inserts for bulk product uploads
- Use write-ahead logs for high-speed ingestion
- Keep transaction sizes small to avoid lock contention

**Read Optimization**

- Redis caching (TTL 5–30 minutes)

- Denormalized search index
- Precomputed recommendation lists
- Materialized views for dashboards

---

## 6.8 Database Scaling Strategy

**Vertical Scaling (initial phase)**

- Start with a single RDS instance
- Add Read Replicas for read-heavy services

**Horizontal Scaling (growth phase)**

- Introduce sharding across multiple clusters
- Move catalog searches fully to search engine
- Redis caching for hot datasets

**Storage Layer Auto-Scaling**

- Object storage (S3/Blob) for product images
- ElasticSearch tiered storage for old documents

---

## 6.9 Backup, Recovery & Disaster Management

**Backups**

- Daily full backups
- Hourly incremental backups
- WAL archiving for point-in-time recovery

**High Availability**

- Multi-AZ deployments
- Automatic failover

**Disaster Recovery**

- RPO: 15 minutes
- RTO: 1 hour
- Standby cluster in second region

---

## 6.10 Data Flow Example: Product Update

1. Seller updates product → Product Service (SQL Write)
2. Product Service writes to DB + Outbox Event
3. Outbox to Kafka (product.updated)
4. Search Indexer consumes and updates Elasticsearch
5. Analytics Pipeline receives event and stores in Data Lake
6. Recommendation Service updates embeddings
7. Redis cache invalidated

   This ensures **eventual consistency** while maintaining **fast reads**.

---

## 6.11 Final Notes

This database design section ensures:

- Efficiency
- Security
- Tenant isolation
- Scalability to millions of products and orders

It aligns with modern enterprise designs used by platforms like **Shopify, Amazon Seller Central, Walmart Marketplace, and Flipkart Seller Hub**.