Production-Ready E-Commerce System Architecture with Java, Spring Boot, and Kafka

1. System Overview and Core Principles

1.1. Architectural Style: Event-Driven Microservices

The proposed e-commerce system is architected as a suite of loosely coupled, independently deployable microservices, forming a distributed system architecture. This approach is a significant departure from traditional monolithic architectures, where all functionalities are tightly integrated into a single, large codebase . The core of this design is an **event-driven architecture (EDA)** , which facilitates asynchronous communication between services. In this model, services communicate by producing and consuming events, which are messages that signify a change in state or an action that has occurred. For instance, when a user places an order, the `Order Service` publishes an `order.placed` event to a central message broker, **Apache Kafka**. Other services, such as the `Inventory Service` and `Payment Service` , subscribe to this topic and react accordingly by reserving stock and processing the payment. This decoupling of services enhances scalability, resilience, and flexibility, as each service can be developed, deployed, and scaled independently without affecting the entire system . The adoption of an event-driven model is particularly well-suited for complex, high-volume e-commerce platforms like Amazon, where a single user action can trigger a cascade of downstream processes across multiple domains.

The decision to use a microservices architecture is driven by the need for a system that can evolve and scale with the business. As noted by Amazon's CTO, Werner Vogels, event-driven architectures and microservices provide a foundation that can be changed and expanded with a minimum number of dependencies . This is in stark contrast to a monolithic architecture, where a change in one component can have unintended consequences on the entire system, making development and deployment a risky and time-consuming process . The microservices approach allows for the use of different technology stacks for different services, a concept known as **polyglot programming**, which enables teams to choose the best tools for the job. For example, a service handling complex machine learning tasks might be written in Python, while the core business services are implemented in Java with Spring Boot. This flexibility is a key advantage in a rapidly evolving technological landscape. Furthermore, the event-driven nature of the system ensures that services remain loosely coupled, which improves fault tolerance. If one service fails, it does not necessarily bring down the entire system, as other services can continue to operate independently.

The transition from a monolithic to a microservices architecture is not without its challenges. It introduces a higher level of complexity in terms of service management, inter-service communication, and data consistency . However, the benefits in terms of scalability, agility, and resilience far outweigh the drawbacks for a large-scale e-commerce platform. The system is designed to handle millions of users and a high volume of transactions, which would be difficult to achieve with a monolithic architecture. The use of a central event bus, Apache Kafka, simplifies the communication between services and provides a reliable and scalable way to handle a large number of events. The system also incorporates various design patterns, such as the **Saga pattern** for managing distributed transactions and the **Outbox pattern** for ensuring event delivery, to address the challenges of a distributed system. The overall goal is to create a robust, scalable, and maintainable e-commerce platform that can support the growth of the business for years to come.

1.2. Key Design Principles

1.2.1. Scalability and High Availability

Scalability and high availability are paramount for a production-ready e-commerce system designed to support millions of users. The microservices architecture is inherently more scalable than a monolithic one, as individual services can be scaled independently based on demand . For example, during a major sales event, the `Order Service` and `Payment Service` might experience a significant increase in traffic. With a microservices architecture, these specific services can be scaled up by deploying more instances, without having to scale the entire application. This targeted scaling is more efficient and cost-effective than scaling a monolithic application, where all components are scaled together, even if some are not under heavy load . The system is designed to be deployed on a **Kubernetes-native infrastructure**, which provides built-in mechanisms for auto-scaling. The **Horizontal Pod Autoscaler (HPA)** can automatically adjust the number of running pods for a service based on CPU utilization or other custom metrics, ensuring that the system can handle traffic spikes without manual intervention. The **Cluster Autoscaler** can also be used to automatically scale the underlying Kubernetes nodes, providing a truly elastic infrastructure that can grow and shrink with demand.

High availability is achieved through a combination of architectural patterns and infrastructure choices. The system is designed to be resilient to failures at multiple levels. At the service level, each microservice is stateless, which means that any instance of a service can handle any request. This allows for load balancing across multiple instances of a service, which not only improves performance but also provides redundancy. If one instance fails, the load balancer can simply route traffic to the remaining healthy instances. At the infrastructure level, the system is deployed across multiple **availability zones (AZs)** in an AWS region. This ensures that if an entire AZ goes down, the system can continue to operate from the other AZs. The use of managed services, such as **Amazon RDS for PostgreSQL** and **Amazon OpenSearch Service**, further enhances availability, as these services are designed with built-in redundancy and failover mechanisms. The event-driven architecture also contributes to high availability. If a service is temporarily unavailable, events can be queued in Kafka and processed once the service comes back online, ensuring that no data is lost.

The design of the system also incorporates the principles of the **AWS Well-Architected Framework**, which provides a set of best practices for building secure, high-performing, resilient, and efficient infrastructure for applications . The five pillars of this framework are operational excellence, security, reliability, performance efficiency, and cost optimization. The system's architecture is designed to align with these pillars, ensuring that it is not only scalable and highly available but also secure, performant, and cost-effective. For example, the use of a content delivery network (CDN) like **Amazon CloudFront** improves performance by caching static content at edge locations around the world, reducing latency for users. The use of auto-scaling and managed services helps to optimize costs by ensuring that resources are only provisioned when they are needed. The overall goal is to create a system that can provide a seamless and reliable shopping experience for users, even during peak traffic periods, while also being efficient and cost-effective to operate.

1.2.2. Service Autonomy and Clear Boundaries

Service autonomy and clear boundaries are fundamental principles of the microservices architecture, and they are essential for building a system that is easy to develop, deploy, and maintain. Each microservice in the system is designed to be a self-contained unit of functionality, with its own codebase, database, and deployment pipeline. This autonomy allows development teams to work independently on different services, without having to coordinate with other teams. This can significantly speed up the development process and improve agility. For example, the team responsible for the

`Product Catalog Service` can make changes to the service and deploy them without having to worry about how those changes might affect the `Order Service` or the `Payment Service` . This is in stark contrast to a monolithic architecture, where a change in one part of the system can have unintended consequences on other parts, requiring extensive testing and coordination .

Clear boundaries between services are essential for maintaining this autonomy. Each service has a well-defined API that specifies how it can be interacted with. This API acts as a contract between the service and its consumers, and it should be stable and well-documented. The internal implementation of a service is hidden behind this API, which allows the service to be refactored or even rewritten without affecting its consumers. This is a key aspect of the **"database per service"** pattern, where each service has its own private database. This ensures that services are not tightly coupled through a shared database, which can lead to a "big ball of mud" architecture where it is difficult to make changes without breaking something . The use of an event-driven architecture further reinforces these boundaries. Instead of services calling each other directly, they communicate through events, which are a more loosely coupled form of communication. This allows services to evolve independently, as long as they continue to produce and consume the same events.

The design of the system also incorporates the principle of **"separation of concerns,"** which is a key tenet of software engineering. Each microservice is responsible for a specific business capability, such as user management, product catalog, or order processing. This makes the system easier to understand and maintain, as each service is focused on a single, well-defined task. The use of a layered architecture within each service can further improve maintainability. For example, a service might have a presentation layer, a business logic layer, and a data access layer. This separation of concerns makes the code easier to read, test, and debug. The overall goal is to create a system that is modular, flexible, and easy to evolve, which is essential for a large-scale e-commerce platform that needs to adapt to changing business requirements.

1.2.3. Fault Tolerance and Resilience

Fault tolerance and resilience are critical for a production-ready e-commerce system, as any downtime can result in lost revenue and a poor user experience. The microservices architecture, when designed correctly, can be more resilient than a monolithic one. In a monolithic application, a single bug or failure can bring down the entire system . In a microservices architecture, on the other hand, a failure in one service does not necessarily affect the rest of the system. For example, if the

`Recommendation Service` is temporarily unavailable, users can still browse the product catalog, add items to their cart, and place orders. This is because the `Recommendation Service` is a non-critical component that can be gracefully degraded. The system is designed to be resilient to failures at multiple levels, from individual service instances to entire availability zones.

The event-driven architecture plays a key role in building a resilient system. By decoupling services through an event bus, the system can handle temporary failures of individual services without losing data or breaking the overall workflow. If a service is down, events can be queued in Kafka and processed once the service is back online. This ensures that the system can recover from failures and continue to operate correctly. The use of the **Outbox pattern** in PostgreSQL-based services further enhances resilience. This pattern ensures that events are reliably published to Kafka, even if the service crashes after committing a transaction to its local database. The system also incorporates various resilience patterns, such as **circuit breakers and retries**, to handle transient failures. A circuit breaker can be used to prevent a service from repeatedly trying to call a failing service, which can lead to a cascade of failures. Instead, the circuit breaker will "trip" and return an error immediately, allowing the calling service to fail gracefully.

The infrastructure is also designed to be resilient. The use of Kubernetes for orchestration provides built-in mechanisms for self-healing. If a pod (a running instance of a service) crashes, Kubernetes will automatically restart it. If a node in the cluster fails, Kubernetes will reschedule the pods that were running on that node to other healthy nodes. The deployment of the system across multiple availability zones provides protection against data center-level failures. The use of managed services, such as **Amazon RDS** and **Amazon OpenSearch Service**, further enhances resilience, as these services are designed with built-in redundancy and failover mechanisms. The overall goal is to create a system that is not only fault-tolerant but also self-healing, so that it can recover from failures automatically and provide a reliable and uninterrupted service to users.

1.2.4. Polyglot Persistence

Polyglot persistence is the principle of using different data storage technologies for different parts of a system, depending on the specific requirements of each part. This is in contrast to the traditional approach of using a single, monolithic database for the entire application. In a microservices architecture, polyglot persistence is a natural fit, as each service can have its own private database that is best suited for its needs. This allows for a more flexible and efficient data layer, as each service can use the right tool for the job. For example, a service that needs to store complex, hierarchical data with a flexible schema might use a NoSQL document database like MongoDB. A service that needs to perform complex queries and transactions might use a relational database like PostgreSQL. A service that needs to provide fast, in-memory access to data might use a key-value store like Redis.

The proposed e-commerce system embraces the principle of polyglot persistence. The `User Service`, `Order Service`, `Inventory Service`, and `Shipping Service`, which require strong consistency and ACID transactions, are backed by **PostgreSQL**. The `Product Catalog Service` and `Review & Rating Service`, which deal with flexible and semi-structured data, use **MongoDB**. The `Cart Service`, which requires high-performance, ephemeral data storage, uses **Redis**. The `Search Service`, which needs to perform full-text search and faceted search, uses **OpenSearch**. This choice of data stores is not arbitrary; it is based on a careful analysis of the requirements of each service. The use of different data stores allows each service to be optimized for its specific workload, which can lead to significant improvements in performance and scalability.

The use of polyglot persistence also has implications for data consistency. In a distributed system with multiple databases, it is not possible to have a single, global transaction that spans all services. Instead, the system must rely on **eventual consistency**, where the data in different services will eventually become consistent. This is achieved through the use of an event-driven architecture. When a service updates its data, it publishes an event to Kafka. Other services can then consume this event and update their own data accordingly. This ensures that the data in the system remains consistent over time, even if it is not immediately consistent. The use of the Saga pattern can also help to manage distributed transactions and ensure data consistency. The overall goal is to create a data layer that is flexible, efficient, and scalable, while also ensuring that the data in the system remains consistent and accurate.

2. Microservices Breakdown

2.1. Core Business Services

2.1.1. User Service

The User Service is a foundational component of the e-commerce platform, responsible for managing all aspects of user identity and authentication. Its primary responsibilities include user registration, login, password reset, and profile management. This service acts as the single source of truth for all user-related data, such as usernames, email addresses, password hashes, and profile information like shipping addresses and payment methods. To ensure the highest level of security and data integrity, the User Service is backed by a **PostgreSQL** database, which provides the ACID compliance necessary for handling sensitive user account information. The service exposes a set of well-defined RESTful APIs for other services and the frontend application to interact with. For example, the Order Service will call the User Service to validate a user's identity before allowing them to place an order. The service also integrates with an identity and access management (IAM) solution like **Keycloak** or a custom JWT-based authentication mechanism to handle the complexities of secure token generation, validation, and session management. This centralized approach to user management simplifies security implementation and ensures a consistent authentication and authorization model across the entire platform.

The User Service is designed to be highly scalable and resilient. It is stateless, allowing it to be deployed as multiple instances behind a load balancer to handle a large number of concurrent authentication requests. The use of a PostgreSQL database with read replicas can further improve performance by distributing read-heavy operations, such as profile lookups, across multiple database instances. The service also publishes events to Kafka, such as `user.registered` or `user.profile.updated` , which can be consumed by other services. For instance, the Notification Service might listen for the `user.registered` event to send a welcome email, and the Recommendation Service might use profile updates to refine its user models. This event-driven interaction decouples the User Service from downstream consumers, enhancing its autonomy and resilience. The service's API endpoints are designed to be OpenAPI-compliant, providing clear documentation and a consistent interface for client applications. For example, a typical endpoint would be `POST /api/v1/users` for registration and `POST /api/v1/auth/login` for authentication.

2.1.2. Product Catalog Service

The Product Catalog Service is the heart of the e-commerce platform's product data, responsible for managing all information related to the products sold on the site. This includes product names, descriptions, categories, tags, images, pricing, and stock availability. A key requirement for this service is the ability to handle a wide variety of product types, each with its own unique set of attributes. For example, a laptop product might have attributes like "CPU," "RAM," and "Storage," while a clothing item might have "Size," "Color," and "Material." To accommodate this need for a flexible and evolving schema, the Product Catalog Service uses **MongoDB** as its primary data store. MongoDB's document-oriented model allows each product to be stored as a self-contained JSON document, making it easy to add, remove, or modify attributes without affecting the entire product collection or requiring complex schema migrations. This flexibility is crucial for an agile e-commerce business that needs to quickly adapt to new product lines and changing market demands.

The service exposes a comprehensive set of RESTful APIs for managing the product catalog. These APIs allow for creating new products, updating existing ones, retrieving product details, and listing products with filtering and pagination. For example, `GET /api/v1/products` might return a paginated list of all products, while `GET /api/v1/products/{id}` would return the full details for a specific product. The service also publishes events to Kafka whenever a product is created, updated, or deleted. These events, such as `product.created` or `product.updated`, are consumed by other services to keep their data in sync. The Search Service, for instance, will consume these events to update its search index, and the Inventory Service might listen for `product.created` events to initialize stock levels for new products. The Product Catalog Service is designed to be highly scalable, with its stateless nature allowing it to be deployed across multiple instances. The use of MongoDB's sharding capabilities can further distribute the product data across multiple servers to handle a massive product catalog and high read/write throughput.

2.1.3. Search Service

The Search Service is a critical component for providing a seamless and efficient user experience, enabling customers to quickly find the products they are looking for. It is responsible for handling all search-related functionalities, including full-text search, faceted filtering (e.g., filtering by category, price range, brand), sorting, and autocomplete suggestions. To deliver the high-performance and advanced search capabilities required for a modern e-commerce platform, the Search Service is built on top of **OpenSearch** (or Elasticsearch). OpenSearch is a distributed, RESTful search and analytics engine that is specifically designed for these types of use cases. It provides powerful full-text search capabilities with support for fuzzy matching, stemming, and synonyms, allowing it to return relevant results even for misspelled or partial queries. The service maintains a search index that is populated with data from the Product

Catalog Service. This is achieved by the Search Service consuming `product.created`, `product.updated`, and `product.deleted` events from a Kafka topic. This event-driven approach ensures that the search index is always up-to-date with the latest product information.

The Search Service exposes a dedicated RESTful API for the frontend application to query. A typical search endpoint would be `GET /api/v1/search?q=laptop&category=Electronics&min_price=500&max_price=1500&sort=price_asc`. This single endpoint can handle complex queries with multiple filters and sorting options, all processed efficiently by the OpenSearch engine. The service is designed to be stateless and horizontally scalable. The OpenSearch cluster itself can be scaled by adding more nodes to handle increasing data volumes and query loads. The service's architecture decouples it from the Product Catalog Service, allowing the two to be scaled and evolved independently. For example, the search relevance algorithms can be fine-tuned or the OpenSearch cluster can be upgraded without any downtime or changes required in the Product Catalog Service. This separation of concerns is a key benefit of the microservices architecture, allowing for specialized optimization of each component.

2.1.4. Cart Service

The Cart Service is responsible for managing the user's shopping cart, a fundamental feature of any e-commerce platform. Its primary function is to allow users to add, update, remove, and view items in their cart as they browse the site. The data for shopping carts is characterized by its ephemeral nature (it is often tied to a user session and can be discarded if the user does not complete a purchase) and the need for extremely fast read and write access to provide a responsive user experience. To meet these requirements, the Cart Service uses **Redis** as its data store. Redis is an in-memory data structure store that offers exceptional performance, with operations typically completing in sub-millisecond timeframes. The cart data for each user is stored as a JSON object in Redis, with the user's ID serving as the key. This allows for very efficient retrieval and modification of the cart's contents. For example, when a user adds an item to their cart, the service simply retrieves the existing JSON object from Redis, adds the new item to the items array, and stores the updated object back in Redis.

The Cart Service exposes a simple and intuitive RESTful API for the frontend to interact with. The endpoints would typically include `GET /api/v1/cart/{userId}` to retrieve the cart, `POST /api/v1/cart/{userId}/items` to add an item, `PUT`

`/api/v1/cart/{userId}/items/{itemId}` to update an item's quantity, and `DELETE /api/v1/cart/{userId}/items/{itemId}` to remove an item. The service is designed to be stateless, allowing it to be scaled horizontally to handle a large number of concurrent users. The use of Redis also provides built-in support for setting expiration times on keys, which can be used to automatically clear abandoned carts after a certain period, helping to manage memory usage. When a user proceeds to checkout, the Cart Service will communicate with the Order Service, typically via a REST call, to transfer the cart items for order creation. The service's reliance on Redis for high-speed data access is a classic example of using the right tool for the job in a polyglot persistence architecture.

2.1.5. Order Service

The Order Service is a cornerstone of the e-commerce platform, responsible for managing the entire lifecycle of a customer's order, from creation to fulfillment. Its core responsibilities include creating new orders, validating order details, calculating totals, managing order status (e.g., pending, confirmed, shipped, delivered), and providing order history for users. Given the critical nature of order data and the need for strict transactional integrity, the Order Service uses a **PostgreSQL** database. This ensures that operations like creating an order and updating inventory are handled with full ACID compliance, preventing issues like duplicate orders or overselling of products. The service's data model in PostgreSQL would include tables for orders, order items, and order status history, with well-defined relationships and constraints to maintain data consistency. For example, an order would have a foreign key to the user who placed it, and order items would have foreign keys to both the order and the corresponding product in the Product Catalog Service.

The Order Service exposes a set of RESTful APIs for order management. Key endpoints would include `POST /api/v1/orders` to create a new order, `GET /api/v1/orders/{orderId}` to retrieve order details, and `GET /api/v1/users/{userId}/orders` to get a user's order history. The process of creating an order is a complex, distributed transaction that involves multiple services. The Order Service orchestrates this process using the **Saga pattern**. When a user places an order, the Order Service first creates a new order record in its database with a "pending" status. It then publishes an `order.placed` event to a Kafka topic. This event is consumed by other services, such as the Inventory Service (to reserve stock), the Payment Service (to process payment), and the Shipping Service (to arrange shipment). Each of these services performs its part of the transaction and publishes a corresponding event (e.g., `inventory.reserved` , `payment.processed`). The Order Service listens for these events and updates the order status accordingly. If any step in the saga fails (e.g., payment is declined), a compensating transaction is triggered to roll back the previous steps (e.g., releasing the reserved inventory). This event-driven saga orchestration ensures data consistency across the distributed system without the need for a centralized, two-phase commit protocol.

2.1.6. Payment Service

The Payment Service is responsible for handling all payment-related operations within the e-commerce platform. Its primary function is to process payments for customer orders by integrating with third-party payment gateways such as Stripe, PayPal, or Braintree. For the purpose of this architecture, it will include a **mock payment integration** to simulate the payment process without handling real financial transactions. The service is designed to be highly secure, as it deals with sensitive payment information. It is crucial to adhere to the **Payment Card Industry Data Security Standard (PCI DSS)** , which mandates strict security controls for handling cardholder data. To this end, the Payment Service is designed to avoid storing sensitive information like full credit card numbers or CVV codes. Instead, it will tokenize payment details using the payment gateway's APIs, storing only non-sensitive information like the payment method type, the last four digits of the card, and a transaction ID. The service uses a **PostgreSQL** database to store these payment records, ensuring transactional integrity.

The Payment Service exposes a RESTful API for the Order Service to call when a payment needs to be processed. A typical endpoint would be `POST /api/v1/payments` , which would receive an order ID and payment details, and return a payment status (e.g., "success," "failed," "pending"). The service listens for `order.placed` events from the Order Service via Kafka. Upon receiving such an event, it initiates the payment process with the external gateway. Once the payment is processed, the service updates its own database and publishes a `payment.successful` or `payment.failed` event back to Kafka. The Order Service consumes this event to update the order status and proceed with the fulfillment workflow or trigger a compensating transaction. This event-driven communication decouples the Order Service from the specifics of the payment processing logic and allows the Payment Service to be developed and scaled independently. The service's design prioritizes security and reliability, with robust error handling and logging to track all payment transactions for auditing and dispute resolution purposes.

2.1.7. Inventory Service

The Inventory Service is a critical component that manages the stock levels of all products in the e-commerce catalog. Its primary responsibilities include tracking the available quantity of each product, handling stock reservations when an order is placed, and updating stock counts when an order is fulfilled or when new stock arrives from suppliers. Maintaining accurate inventory data is essential to prevent overselling, which can lead to customer dissatisfaction and lost sales. To ensure data consistency and reliability, the Inventory Service uses a **PostgreSQL** database. The database schema would include a table for inventory records, with columns for the product ID, the total stock count, and the number of units currently reserved for pending orders. This allows the service to accurately calculate the available stock for sale (total stock - reserved stock) at any given time. The service's logic must be carefully designed to handle concurrent access to inventory data to prevent race conditions.

The Inventory Service communicates with other services through a combination of REST and Kafka. It listens for `order.placed` events from the Order Service. When it receives such an event, it attempts to reserve the required quantity of each product in the order. If sufficient stock is available, it updates the "reserved" count in its database and publishes an `inventory.reserved` event. If stock is not available, it publishes an `inventory.out_of_stock` event, which would trigger a compensating transaction in the Order Service to cancel the order. The service also listens for `order.cancelled` or `order.shipped` events. In the case of a cancellation, it would release the reserved stock back to the available pool. In the case of a shipment, it would decrement both the reserved count and the total stock count. The service may also expose a RESTful API for internal administrative tools to manually adjust stock levels or for the Product Catalog Service to retrieve real-time availability information. The design of the Inventory Service is focused on ensuring data accuracy and consistency, which are paramount for a reliable e-commerce operation.

2.1.8. Shipping Service

The Shipping Service is responsible for managing the shipment of orders to customers. Its core functionalities include integrating with third-party shipping carriers (such as FedEx, UPS, or DHL) to create shipping labels, calculate shipping costs, and track the status of shipments from dispatch to delivery. The service acts as an abstraction layer over the various carrier APIs, providing a unified interface for the rest of the system to interact with. This allows the e-commerce platform to easily add or switch shipping providers without requiring changes in other services like the Order Service. The service stores shipping-related information, such as tracking numbers, carrier details, estimated delivery dates, and shipment status history, in a **PostgreSQL** database. This ensures that the shipping data is reliably persisted and can be easily queried to provide order tracking information to customers.

The Shipping Service listens for `order.confirmed` and `payment.successful` events from the Order Service via Kafka. Once an order is confirmed and paid for, the Shipping Service initiates the shipping process. It communicates with the appropriate carrier's API to create a shipment and obtain a tracking number. It then updates its database with the shipping details and publishes a `shipment.created` event, which is consumed by the Order Service to update the order status to "shipped." The Shipping Service may also run a periodic background job to poll the carrier APIs for shipment status updates. When a status change is detected (e.g., "in_transit," "out_for_delivery," "delivered"), it updates its database and publishes a corresponding event (e.g., `shipment.status_updated`). The Notification Service can consume these events to send proactive updates to the customer via email or SMS. The service exposes a RESTful API, primarily for the frontend to retrieve tracking information for a specific order, such as `GET /api/v1/shipments/{orderId}/tracking`. The design of the Shipping Service focuses on providing a reliable and transparent shipping experience for the customer.

2.1.9. Review & Rating Service

The Review & Rating Service is responsible for managing customer feedback on products. It allows users to submit reviews, which can include a text comment and a star rating, for products they have purchased. The service also supports more advanced features like nested reviews (for questions and answers) and the ability for other users to rate reviews as "helpful." The data for reviews is semi-structured and can vary, making a flexible NoSQL database an ideal choice. Therefore, the Review & Rating Service uses **MongoDB** to store review documents. Each review document would contain the user ID, product ID, star rating, review text, and timestamp. The flexible schema of MongoDB allows for easy addition of new fields in the future, such as "verified purchase" flags or the ability to attach photos to reviews.

The service exposes a RESTful API for creating and retrieving reviews. When a user submits a review, the frontend application sends a request to this service, which then stores the review in its MongoDB database. The service also publishes a `review.created` event to Kafka, which can be consumed by other services. For example, the Product Catalog Service might listen for this event to update the average rating for a product, and the Recommendation Service might use review data to improve its recommendations. The service also provides endpoints for fetching reviews for a specific product, which the frontend uses to display the reviews on the product detail page. These endpoints can support pagination and sorting, allowing users to browse through reviews and see the most helpful or recent ones first. The Review & Rating Service is a key component for building a community around the e-commerce platform and for providing valuable social proof that can influence purchasing decisions.

2.1.10. Notification Service

The Notification Service is a cross-cutting concern in the e-commerce platform, responsible for sending all types of notifications to users, including emails, SMS messages, and push notifications. This service acts as a centralized hub for all communication with customers, ensuring a consistent and reliable delivery of important information. The Notification Service is designed to be decoupled from the other services in the system. It does not need to know the details of why a notification is being sent; it simply listens for specific events on a Kafka topic and sends the appropriate notification based on the event type. For example, it might listen for `user.registered` events to send a welcome email, `order.placed` events to send an order confirmation, `shipment.created` events to send a shipping notification with a tracking link, and `order.delivered` events to send a delivery confirmation.

The Notification Service integrates with various third-party providers for the actual delivery of notifications, such as **SendGrid** or **Amazon SES** for emails, **Twilio** for SMS, and **Firebase Cloud Messaging** for push notifications. This allows the platform to leverage the reliability and scalability of specialized communication platforms. The service uses a template engine (e.g., Thymeleaf or Handlebars) to generate the content of the notifications, allowing for dynamic and personalized messages. For example, an order confirmation email would be populated with the user's name, the order number, and the list of items purchased. The Notification Service also handles the logic for managing user notification preferences, allowing users to opt-in or opt-out of different types of notifications. By centralizing all notification logic in a single service, the system avoids code duplication and ensures that all communications are handled in a consistent and maintainable way. This event-driven approach makes the system highly extensible, as new types of notifications can be easily added by simply creating a new event consumer for a new Kafka topic.

2.1.11. Recommendation Service (ML-based)

The Recommendation Service is a machine learning-based service that provides personalized product recommendations to users on the e-commerce platform. It analyzes user behavior, such as browsing history, purchase history, and product ratings, to generate recommendations that are tailored to each user's interests. The Recommendation Service is designed to be a highly scalable and performant component of the system, as it needs to handle a large volume of user data and to generate recommendations in real-time. The service is implemented as a **Python microservice**, which allows it to leverage the rich ecosystem of machine learning libraries, such as **TensorFlow** and **PyTorch**. The Recommendation Service uses a combination of collaborative filtering, content-based filtering, and deep learning models to generate recommendations.

The Recommendation Service consumes events from a Kafka topic, such as `product.viewed` or `order.placed`, to collect data for training its machine learning models. The service also uses **Apache Spark** for large-scale data processing and model training. The trained models are then deployed to the Recommendation Service, which uses them to generate recommendations for users. The Recommendation Service exposes a RESTful API that is consumed by the frontend application to display personalized recommendations to users on the homepage, product detail pages, and other parts of the site. The Recommendation Service is a key component of the e-commerce platform, as it helps users discover new products and can significantly increase sales and customer engagement. The design and implementation of this service are crucial for providing a personalized and engaging shopping experience for users.

2.2. Supporting Services

2.2.1. API Gateway

The API Gateway is a critical component of the microservices architecture, acting as a single entry point for all external client requests. It is responsible for routing requests to the appropriate microservice, as well as for handling cross-cutting concerns such as authentication, authorization, rate limiting, and logging. The API Gateway is designed to be a highly scalable and performant component of the system, as it needs to handle a large volume of requests. The API Gateway is implemented using a technology like **Spring Cloud Gateway** or **Kong**, which provides a rich set of features for building and managing APIs. The API Gateway is a stateless service, which means that it does not store any data locally. It can be scaled horizontally by adding more instances to handle increased traffic.

The API Gateway plays a crucial role in ensuring the security and reliability of the e-commerce platform. It can be configured to enforce authentication and authorization policies, ensuring that only authenticated users can access protected resources. It can also be configured to implement rate limiting, which can prevent a single client from overwhelming the system with too many requests. The API Gateway can also be used to implement caching, which can improve performance by caching responses from downstream services. The API Gateway is a critical component of the e-commerce platform, and its design and implementation are crucial for ensuring a secure, reliable, and performant system.

2.2.2. Admin Portal Service

The Admin Portal Service is a web-based application that provides a user interface for administrators and staff to manage the e-commerce platform. It is a critical tool for managing the product catalog, processing orders, managing inventory, and moderating user reviews. The Admin Portal Service is a separate application from the main e-commerce site, and it is designed to be used by internal users only. The service is built using a modern web framework, such as React or Angular, and it communicates with the backend microservices through the API Gateway. The Admin Portal Service is designed to be highly secure, with strict authentication and authorization policies to ensure that only authorized staff can access the administrative functions.

The Admin Portal Service provides a rich set of features for managing the e-commerce platform. It provides a user interface for creating, updating, and deleting products in the product catalog. It also provides a user interface for viewing and processing orders, including the ability to update order status, process refunds, and manage returns. The service also provides a user interface for managing inventory, including the ability to view stock levels, update stock counts, and set low-stock alerts. The Admin Portal Service is a critical component of the e-commerce platform, and its design and implementation are crucial for ensuring the smooth and efficient operation of the business.

2.2.3. Analytics/Event Logging Service

The Analytics/Event Logging Service is a critical component of the e-commerce platform, responsible for capturing and storing user and product interaction events. This service is designed to be a highly scalable and performant component of the system, as it needs to handle a large volume of events. The service consumes events from a variety of Kafka topics, such as `product.viewed` , `product.added_to_cart` , and `order.placed` . The service then processes these events and stores them in a data warehouse, such as **Apache Cassandra** or **Amazon Redshift**, for further analysis. The Analytics/Event Logging Service is a key component of the e-commerce platform, as it provides valuable insights into user behavior and product performance.

The Analytics/Event Logging Service is designed to be a highly scalable and resilient component of the system. It is a stateless service, which means that it can be scaled horizontally by adding more instances. The service also uses a distributed data warehouse, such as Cassandra, to store the event data. This allows the service to handle a large volume of writes and to provide fast and efficient access to the data for analysis. The Analytics/Event Logging Service is a critical component of the e-commerce platform, and its design and implementation are crucial for providing a data-driven approach to business decision-making.

2.2.4. Media Storage Service

The Media Storage Service is a critical component of the e-commerce platform, responsible for storing and serving media files, such as product images and videos. The service is designed to be a highly scalable and performant component of the system, as it needs to handle a large volume of media files. The service is built on top of an object storage system, such as **MinIO** or **Amazon S3**, which provides a highly scalable and durable storage solution for media files. The Media Storage Service exposes a RESTful API that allows other services to upload and retrieve media files. The service also provides a content delivery network (CDN) to serve the media files, which can significantly improve performance by caching the files at edge locations around the world.

The Media Storage Service is designed to be a highly scalable and resilient component of the system. It is a stateless service, which means that it can be scaled horizontally by adding more instances. The service also uses a distributed object storage system, such as MinIO, to store the media files. This allows the service to handle a large volume of files and to provide high availability and durability. The Media Storage Service is a critical component of the e-commerce platform, and its design and implementation are crucial for providing a rich and engaging user experience.

2.2.5. ML Pipeline Orchestrator

The ML Pipeline Orchestrator is a critical component of the e-commerce platform, responsible for orchestrating the machine learning pipeline for the Recommendation

Service. The service is designed to be a highly scalable and reliable component of the system, as it needs to handle a large volume of data and to ensure the timely execution of the machine learning pipeline. The service is built on top of an orchestration framework, such as **Apache Airflow**, which provides a powerful and flexible platform for defining, scheduling, and monitoring complex workflows. The ML Pipeline Orchestrator is responsible for defining the workflow for the machine learning pipeline, which includes data ingestion, data preprocessing, model training, and model deployment.

The ML Pipeline Orchestrator is designed to be a highly scalable and resilient component of the system. It is a stateless service, which means that it can be scaled horizontally by adding more instances. The service also uses a distributed orchestration framework, such as Airflow, to manage the machine learning pipeline. This allows the service to handle a large volume of data and to ensure the timely execution of the pipeline. The ML Pipeline Orchestrator is a critical component of the e-commerce platform, and its design and implementation are crucial for providing a personalized and engaging user experience.

2.2.6. Batch Processing Service

The Batch Processing Service is a critical component of the e-commerce platform, responsible for performing large-scale data processing tasks. The service is designed to be a highly scalable and performant component of the system, as it needs to handle a large volume of data. The service is built on top of a distributed computing framework, such as **Apache Spark**, which provides a powerful and flexible platform for processing large datasets. The Batch Processing Service is responsible for performing a variety of data processing tasks, such as data ETL (extract, transform, load), data aggregation, and data analysis. The service is also responsible for training the machine learning models for the Recommendation Service.

The Batch Processing Service is designed to be a highly scalable and resilient component of the system. It is a stateless service, which means that it can be scaled horizontally by adding more instances. The service also uses a distributed computing framework, such as Spark, to process the data. This allows the service to handle a large volume of data and to provide fast and efficient processing. The Batch Processing Service is a critical component of the e-commerce platform, and its design and implementation are crucial for providing a data-driven approach to business decision-making.

3. Technology Stack

3.1. Core Framework and Language

3.1.1. Java and Spring Boot

The core of the backend development will be built using **Java**, a mature, robust, and highly performant programming language that is well-suited for building large-scale, enterprise-grade applications. Its strong typing, extensive ecosystem, and large community of developers make it a reliable choice for a long-term project. The primary framework for developing the microservices will be **Spring Boot**. Spring Boot is an open-source framework that simplifies the creation of stand-alone, production-grade Spring-based applications. It provides a set of conventions and pre-configured templates that significantly reduce boilerplate code and configuration, allowing developers to focus on writing business logic. Its "opinionated" approach to development accelerates the initial setup and ongoing maintenance of the microservices.

The Spring ecosystem offers a rich set of projects that are essential for building a microservices architecture. **Spring Cloud** provides tools for common patterns in distributed systems, such as service discovery, configuration management, circuit breakers, and API gateways. **Spring Data** simplifies data access by providing a consistent and easy-to-use abstraction layer over various database technologies, including PostgreSQL, MongoDB, and Redis. **Spring Security** offers a comprehensive and customizable framework for authentication and authorization, which is crucial for securing the e-commerce platform. **Spring Kafka** provides seamless integration with Apache Kafka, making it straightforward to build event-driven microservices that can produce and consume events. The combination of Java's performance and reliability with Spring Boot's productivity and the extensive Spring ecosystem provides a solid foundation for building a scalable and maintainable e-commerce system.

3.2. Data Storage

3.2.1. PostgreSQL for Transactional Data

PostgreSQL is the primary choice for relational data that requires strict consistency and ACID (Atomicity, Consistency, Isolation, Durability) compliance. This is crucial for core business operations where data integrity is non-negotiable. Services like the **User Service**, **Order Service**, **Payment Service**, **Inventory Service**, and **Shipping Service** will use PostgreSQL. Its robust support for transactions, complex queries, and data integrity constraints makes it ideal for managing user accounts, financial transactions, and order fulfillment processes. The use of PostgreSQL ensures that critical operations, such as placing an order or processing a payment, are executed reliably and without data corruption. In a production environment on AWS, this would typically be managed through **Amazon RDS for PostgreSQL**, which provides a highly available and scalable managed database service.

3.2.2. MongoDB for Flexible Schemas

MongoDB is a document-oriented NoSQL database chosen for its flexibility in handling semi-structured and evolving data. This is particularly well-suited for the **Product Catalog Service** and the **Review & Rating Service**. Product information often has varied attributes depending on the category (e.g., a laptop vs. a book), and user reviews can have a dynamic structure. MongoDB's schema-less nature allows for easy storage and retrieval of this data without the need for rigid, predefined table structures. This agility is essential for an e-commerce platform that needs to quickly adapt to new product lines and changing business requirements. In a production environment on AWS, this would be managed through **Amazon DocumentDB (with MongoDB compatibility)** or a self-managed MongoDB cluster on **Amazon EC2**.

3.2.3. Redis for High-Performance Caching

Redis is an in-memory data store used for its exceptional performance in scenarios requiring low-latency read and write access. The primary use case in this architecture is the **Cart Service**, which needs to manage user shopping carts with high speed and responsiveness. Storing cart data in Redis allows for near-instantaneous retrieval and updates, providing a seamless user experience. Additionally, Redis can be used as a caching layer for frequently accessed data from other services, such as product details or session information, to reduce the load on primary databases like PostgreSQL and MongoDB. Its support for data structures like lists, sets, and hashes makes it a versatile tool for various caching and real-time data processing needs.

3.2.4. OpenSearch for Full-Text Search

OpenSearch (a community-driven, open-source fork of Elasticsearch) is a distributed search and analytics engine specifically designed for full-text search, faceted search, filtering, and autocomplete. The **Search Service** is built on top of OpenSearch to provide a powerful and efficient search experience for users. Indexing the product catalog in OpenSearch allows for complex queries, fuzzy matching to handle typos, and real-time suggestions as users type. Its distributed nature enables horizontal scaling to handle large product catalogs and high query volumes. In a production environment on AWS, this would be managed through the **Amazon OpenSearch Service**, which provides a managed, scalable, and secure way to run OpenSearch clusters.

3.3. Inter-Service Communication

3.3.1. Apache Kafka for Event-Driven Architecture

Apache Kafka is the cornerstone of the system's inter-service communication, serving as the central event bus for all asynchronous, event-driven interactions. Kafka is an open-source distributed event streaming platform that is designed for high-throughput, fault-tolerant, and scalable data pipelines. It is an ideal choice for a microservices architecture because it decouples services, allowing them to communicate without direct dependencies. In this system, services will act as producers, publishing events to Kafka topics, and as consumers, subscribing to topics to receive and process events. For example, when a user registers, the `User Service` will publish a `user.registered` event to a dedicated topic. The `Notification Service` and `Analytics Service` can then consume this event to send a welcome email and log the registration, respectively, without the `User Service` needing to know about these downstream actions .

The use of Kafka provides several key benefits. First, it ensures durability and reliability. Events are persisted on disk and can be retained for a configurable period, which means that even if a consumer is temporarily down, it can catch up on missed events once it recovers. This is crucial for maintaining data consistency across the system. Second, Kafka's distributed and partitioned architecture allows it to scale horizontally to handle massive volumes of events, which is essential for a high-traffic e-commerce platform. Third, it enables real-time data processing. Events are delivered with low latency, allowing services to react to changes in the system almost instantaneously. This is particularly important for features like real-time inventory updates and personalized recommendations. The system will leverage Kafka's features, such as consumer groups for parallel processing and partitioning for scalability, to build a robust and efficient event-driven architecture .

3.3.2. REST/gRPC for Synchronous Communication

While the primary mode of communication is asynchronous and event-driven via Kafka, there are scenarios where synchronous communication is necessary. For example, when a user adds an item to their cart, the `Cart Service` may need to synchronously check with the `Inventory Service` to ensure the item is in stock before allowing the operation. For these use cases, the system will use a combination of **REST** and **gRPC**. **REST (Representational State Transfer)** is a widely adopted architectural style for building web services. It is simple, stateless, and uses standard HTTP methods (GET, POST, PUT, DELETE) for communication. The microservices will expose RESTful APIs for synchronous interactions, and these APIs will be documented using the OpenAPI specification to ensure clarity and consistency.

For performance-critical synchronous communication, **gRPC** will be used. gRPC is a high-performance, open-source RPC (Remote Procedure Call) framework developed by Google. It uses Protocol Buffers, a language-neutral, platform-neutral, and extensible mechanism for serializing structured data, which is more efficient than JSON used in REST. gRPC is ideal for low-latency, high-throughput communication between services, especially in a microservices architecture where there can be a high volume of internal service-to-service calls. The system will use a service mesh like **Istio** or **Linkerd** to manage the communication between services, providing features like load balancing, service discovery, and security (mTLS) for both REST and gRPC traffic. This hybrid approach, using Kafka for asynchronous events and REST/gRPC for synchronous calls, provides the flexibility to choose the most appropriate communication pattern for each interaction, resulting in a more efficient and resilient system.

3.4. Infrastructure and Deployment

3.4.1. Docker for Containerization

Docker is the foundational technology for packaging and deploying the microservices. Each microservice, along with its dependencies, will be packaged into a lightweight, portable Docker container. This ensures consistency across development, testing, and production environments, eliminating the "it works on my machine" problem. Docker containers provide process isolation, making them more secure and efficient than traditional virtual machines. By containerizing the services, the system can achieve greater density, running more services on the same hardware, and simplifying the deployment and management of the application.

3.4.2. Kubernetes for Orchestration

Kubernetes is the container orchestration platform of choice for managing the Docker containers at scale. It automates the deployment, scaling, and management of the containerized microservices. Kubernetes provides a declarative way to define the desired state of the application, and it continuously works to ensure that the actual state matches the desired state. It offers powerful features such as self-healing (automatically restarting failed containers), service discovery and load balancing, and rolling updates with zero downtime. For this e-commerce platform, a managed Kubernetes service like **Amazon Elastic Kubernetes Service (EKS)** will be used to offload the operational overhead of managing the Kubernetes control plane.

3.4.3. Helm for Package Management

Helm is the package manager for Kubernetes, and it will be used to simplify the deployment and management of the microservices and infrastructure components. Helm charts are packages of pre-configured Kubernetes resources that can be easily customized and deployed. The system will use Helm charts to deploy the microservices, as well as the supporting infrastructure components like Kafka, PostgreSQL, MongoDB, and Redis. This approach makes the deployments more reproducible, versionable, and easier to manage, especially in a complex microservices environment with many interdependent components.

3.5. Observability and Monitoring

3.5.1. Prometheus and Grafana for Metrics

Prometheus is an open-source monitoring and alerting toolkit that will be used to collect and store metrics from the microservices. Each service will expose a `/metrics` endpoint that provides a snapshot of its internal state, such as request counts, response times, and error rates. Prometheus will scrape these endpoints at regular intervals and store the time-series data. **Grafana** is a powerful visualization tool that will be used to create dashboards and graphs from the metrics stored in Prometheus. This will provide a real-time view of the health and performance of the entire system, allowing operators to quickly identify and diagnose issues.

3.5.2. Loki for Log Aggregation

Loki is a horizontally-scalable, highly-available, multi-tenant log aggregation system inspired by Prometheus. It will be used to collect, store, and query logs from all the microservices. Unlike other log aggregation systems, Loki does not index the contents of the logs, but rather a set of labels for each log stream. This makes it more cost-effective and easier to operate. The logs from the microservices will be collected by a log shipper like **Promtail** and sent to Loki. Grafana will be used to query and visualize the logs, providing a centralized view of the system's activity and making it easier to troubleshoot issues.

3.5.3. OpenTelemetry for Distributed Tracing

OpenTelemetry is a set of APIs, libraries, agents, and instrumentation that will be used to generate, collect, and export telemetry data (metrics, logs, and traces) from the microservices. For distributed tracing, OpenTelemetry will be used to instrument the microservices to track requests as they flow through the system. This will provide a detailed view of the request lifecycle, including the time spent in each service and the calls made between services. This is invaluable for understanding the performance characteristics of the system and for identifying bottlenecks and latency issues. The trace data will be collected by an OpenTelemetry Collector and exported to a distributed tracing backend like **Jaeger** or **Zipkin**.

3.6. CI/CD and GitOps

3.6.1. GitHub Actions for Pipelines

GitHub Actions will be used to automate the continuous integration and continuous delivery (CI/CD) pipelines for the microservices. When a developer pushes a change to a microservice's repository, a GitHub Actions workflow will be triggered. This workflow will typically include steps to build the application, run unit and integration tests, build a Docker image, and push the image to a container registry like **Amazon Elastic Container Registry (ECR)** . This automated process ensures that code changes are tested and packaged consistently, reducing the risk of human error and accelerating the development cycle.

3.6.2. ArgoCD for GitOps Deployment

ArgoCD is a declarative, GitOps continuous delivery tool for Kubernetes. It will be used to automate the deployment of the microservices to the Kubernetes cluster. With ArgoCD, the desired state of the application is defined in a Git repository using Kubernetes manifests or Helm charts. ArgoCD continuously monitors the Git repository and the Kubernetes cluster, and it automatically applies any changes to the cluster to match the state defined in Git. This GitOps approach provides a clear audit trail of all deployments, makes rollbacks easy, and ensures that the cluster is always in a known, desired state.

3.7. Security

3.7.1. Keycloak for Identity and Access Management

For robust and scalable identity and access management (IAM), the system will integrate **Keycloak**, an open-source identity and access management solution. Keycloak provides a centralized platform for managing user identities, authentication, and authorization. It supports a wide range of authentication protocols, including OAuth 2.0, OpenID Connect, and SAML, which are essential for securing modern web applications and APIs. By using Keycloak, the system can offload the complexities of user management, such as user registration, password policies, and multi-factor authentication, to a dedicated service. This simplifies the development of the individual microservices, as they can delegate authentication and authorization decisions to Keycloak.

The integration with Keycloak will follow a standard OAuth 2.0 / OpenID Connect flow. When a user or a client application wants to access a protected resource, they will first authenticate with Keycloak to obtain a JSON Web Token (JWT). This JWT will contain information about the user (claims) and their permissions (scopes). The client will then present this JWT with each subsequent request to the API Gateway. The API Gateway will validate the token with Keycloak and, if valid, forward the request to the appropriate downstream microservice. The microservices can then use the information in the JWT to make fine-grained authorization decisions. This token-based approach is stateless and scalable, as it eliminates the need for server-side session management. Keycloak's support for features like single sign-on (SSO), social login, and user federation makes it a powerful and flexible choice for managing user identities in a large-scale e-commerce platform.

3.7.2. HashiCorp Vault for Secrets Management

To ensure the security of sensitive configuration data, such as database passwords, API keys, and other credentials, the system will use **HashiCorp Vault**. Vault is an open-source tool for securely managing secrets and protecting sensitive data. It provides a centralized and encrypted storage for secrets, with fine-grained access control and detailed audit logging. Instead of hardcoding secrets in application code or configuration files, which is a major security risk, the microservices will retrieve their secrets from Vault at runtime. This approach, known as "secrets as a service," significantly improves the security posture of the system by ensuring that secrets are never exposed in plain text.

The integration with Vault will be managed through the **Vault Agent**, which can be deployed as a sidecar container alongside each microservice in the Kubernetes cluster. The Vault Agent will be responsible for authenticating with the Vault server, retrieving the necessary secrets, and injecting them into the microservice's environment or a shared memory volume. This process is transparent to the application code, which can simply read the secrets from the environment variables or the file system. Vault's dynamic secrets feature will also be leveraged to generate short-lived, on-demand credentials for databases and other services. This means that instead of using long-lived, static passwords, the microservices will receive temporary credentials that are automatically revoked after a certain period. This minimizes the risk of credential compromise and reduces the operational overhead of rotating secrets. By using Vault, the system can ensure that all sensitive data is securely managed, access is tightly controlled, and all interactions are auditable.

4. Database Design (Per Service)

4.1. PostgreSQL Schema Design

4.1.1. User Service Tables

The User Service will use a PostgreSQL database to store user-related data. The primary table will be the `users` table, which will store the core user information.

Table: `users`

表格 复制

Column Name	Data Type	Constraints	Description
`user_id`	`UUID`	`PRIMARY KEY`	Unique identifier for tl
`email`	`VARCHAR(255)`	`UNIQUE, NOT NULL`	User's email address,
`password_hash`	`VARCHAR(255)`	`NOT NULL`	Hashed password for
`first_name`	`VARCHAR(100)`		User's first name.
`last_name`	`VARCHAR(100)`		User's last name.
`created_at`	`TIMESTAMP`	`DEFAULT NOW()`	Timestamp of user reg
`updated_at`	`TIMESTAMP`	`DEFAULT NOW()`	Timestamp of last pro

A separate table, `user_addresses` , can be used to store multiple shipping and billing addresses for each user, with a foreign key reference to the `users` table.

4.1.2. Order Service Tables

The Order Service will use a PostgreSQL database to store order-related data. The schema will be normalized to ensure data integrity.

Table: `orders`

表格 复制

Column Name	Data Type	Constraints	Description
`order_id`	`UUID`	`PRIMARY KEY`	Unique identifi
`user_id`	`UUID`	`FOREIGN KEY (users.user_id)`	Reference to t
`status`	`VARCHAR(50)`	`NOT NULL`	Current status
`total_amount`	`DECIMAL(10, 2)`	`NOT NULL`	Total price of
`created_at`	`TIMESTAMP`	`DEFAULT NOW()`	Timestamp of

Table: `order_items`

表格 复制

Column Name	Data Type	Constraints	Description
`order_item_id`	`UUID`	`PRIMARY KEY`	Unique iden
`order_id`	`UUID`	`FOREIGN KEY (orders.order_id)`	Reference t
`product_id`	`UUID`	`NOT NULL`	Reference t
`quantity`	`INTEGER`	`NOT NULL`	Quantity of
`price`	`DECIMAL(10, 2)`	`NOT NULL`	Price of the

4.1.3. Inventory Service Tables

The Inventory Service will use a PostgreSQL database to manage stock levels.

Table: `inventory`

表格 复制

Column Name	Data Type	Constraints	Description
product_id	UUID	PRIMARY KEY	Unique identifier for the product.
stock_count	INTEGER	NOT NULL	Total available stock for the produ
reserved_count	INTEGER	NOT NULL	Stock reserved for pending orders

The available stock for sale can be calculated as `stock_count - reserved_count` .

4.1.4. Shipping Service Tables

The Shipping Service will use a PostgreSQL database to store shipping information.

Table: `shipments`

表格 复制

Column Name	Data Type	Constraints	Descrip
shipment_id	UUID	PRIMARY KEY	Unique
order_id	UUID	FOREIGN KEY (orders.order_id)	Referer
carrier	VARCHAR(100)		Name c
tracking_id	VARCHAR(255)		Trackin
status	VARCHAR(50)		Current DELIVE
estimated_delivery	DATE		Estimat

4.2. MongoDB Document Design

4.2.1. Product Catalog Documents

The Product Catalog Service will use MongoDB to store product information in a flexible, JSON-like format.

Document: `Product`

JSON 复制

```
{
  "_id": "uuid",
  "name": "Product X",
  "description": "A detailed description of the product.",
  "category": "Electronics",
  "subcategory": "Laptops",
  "tags": ["laptop", "gaming", "portable"],
  "brand": "BrandName",
  "variants": [
    {
      "variant_id": "uuid",
      "sku": "PROD-X-001",
      "attributes": {
        "color": "Black",
        "storage": "512GB SSD"
      },
      "price": 899.99,
      "images": [
        "https://media.example.com/images/prod-x-black-1.jpg",
        "https://media.example.com/images/prod-x-black-2.jpg"
      ]
    },
    {
      "variant_id": "uuid",
      "sku": "PROD-X-002",
      "attributes": {
        "color": "Silver",
        "storage": "1TB SSD"
      },
      "price": 1099.99,
      "images": [
        "https://media.example.com/images/prod-x-silver-1.jpg"
      ]
    }
  ],
  "specifications": {
    "processor": "Intel Core i7",
    "ram": "16GB",
    "screen_size": "15.6 inch"
  },
  "created_at": "2025-08-14T10:00:00Z",
  "updated_at": "2025-08-14T10:00:00Z"
}
```

4.2.2. Review & Rating Documents

The Review & Rating Service will use MongoDB to store user reviews.

Document: `Review`

JSON 复制

```json
{
  "_id": "uuid",
  "product_id": "product_uuid",
  "user_id": "user_uuid",
  "order_id": "order_uuid",
  "rating": 5,
  "title": "Excellent product!",
  "review": "This product exceeded my expectations. Highly
recommended.",
  "is_verified_purchase": true,
  "helpful_votes": 42,
  "replies": [
    {
      "user_id": "another_user_uuid",
      "reply": "Thanks for the detailed review!",
      "timestamp": "2025-08-14T12:00:00Z"
    }
  ],
  "timestamp": "2025-08-14T11:00:00Z"
}
```

4.3. Redis Data Structures

4.3.1. Cart Service Data Model

The Cart Service will use Redis to store user shopping carts. Each cart will be stored as a JSON object with the user's ID as the key.

Key: `cart:{userId}`

Value (JSON):

JSON 复制

```json
{
  "user_id": "uuid",
  "items": [
    {
      "product_id": "product_uuid_1",
      "variant_id": "variant_uuid_1",
      "quantity": 2,
      "price": 899.99
    },
    {
      "product_id": "product_uuid_2",
      "variant_id": "variant_uuid_2",
      "quantity": 1,
      "price": 19.99
    }
  ],
  "updated_at": "2025-08-14T11:30:00Z"
}
```

The Redis key can be set with an expiration time (e.g., 30 days) to automatically clear abandoned carts.

4.4. OpenSearch Indexing Strategy

4.4.1. Product Search Index

The Search Service will use OpenSearch to create a search index for the product catalog. The index will be designed to support full-text search, filtering, and faceting.

Index Name: `products`

Mapping:

JSON 复制

```json
{
  "mappings": {
    "properties": {
      "product_id": { "type": "keyword" },
      "name": {
        "type": "text",
        "analyzer": "standard",
        "fields": {
          "keyword": { "type": "keyword" }
        }
      },
      "description": { "type": "text", "analyzer": "standard" },
      "category": { "type": "keyword" },
      "subcategory": { "type": "keyword" },
      "tags": { "type": "keyword" },
      "brand": { "type": "keyword" },
      "variants": {
        "type": "nested",
        "properties": {
          "variant_id": { "type": "keyword" },
          "sku": { "type": "keyword" },
          "price": { "type": "double" },
          "attributes": {
            "type": "object",
            "properties": {
              "color": { "type": "keyword" },
              "storage": { "type": "keyword" }
            }
          }
        }
      },
      "created_at": { "type": "date" }
    }
  }
}
```

This mapping allows for searching across `name` and `description`, filtering by `category`, `brand`, and nested `variants.attributes`, and sorting by `price`.

5. Inter-Service Communication and Event Flow

5.1. Kafka Event Design

5.1.1. Core Event Topics (e.g., `user.registered`, `order.placed`)

The event-driven architecture will be built around a set of core Kafka topics that represent significant business events. Each topic will have a clear purpose and a defined set of producers and consumers.

表格 复制

Topic Name	Description
`user.registered`	A new user has successfully registered.
`user.profile.updated`	A user's profile information has been changed.

Topic Name	Description
product.created	A new product has been added to the catalog.
product.updated	An existing product's information has been changed.
product.viewed	A user has viewed a product detail page.
cart.item_added	An item has been added to a user's cart.
order.placed	A new order has been placed by a user.
payment.successful	A payment for an order has been successfully processed.
payment.failed	A payment for an order has failed.
inventory.reserved	Inventory has been successfully reserved for an order.
inventory.out_of_stock	An order could not be placed due to insufficient stock.
shipment.created	A shipment has been created for an order.
shipment.status_updated	The status of a shipment has changed.
review.created	A new review has been submitted for a product.

5.1.2. Event Payload Structure

All events will have a consistent payload structure to ensure interoperability and ease of processing. The payload will be in JSON format and will include a standard set of metadata.

Standard Event Envelope:

JSON 复制

```json
{
  "event_id": "uuid",
  "event_type": "user.registered",
  "event_version": "1.0",
  "timestamp": "2025-08-14T10:00:00Z",
  "correlation_id": "uuid",
  "payload": {
    // Event-specific data
  }
}
```

- `event_id` : A unique identifier for the event.
- `event_type` : The name of the event (e.g., `user.registered`).
- `event_version` : The version of the event schema, allowing for evolution.
- `timestamp` : The time the event was generated.
- `correlation_id` : A unique ID that is passed through all events in a single transaction, used for distributed tracing.
- `payload` : The actual data of the event, which will vary depending on the event type.

Example `order.placed` Payload:

JSON 复制

```json
{
  "event_id": "uuid-123",
  "event_type": "order.placed",
  "event_version": "1.0",
  "timestamp": "2025-08-14T10:05:00Z",
  "correlation_id": "uuid-abc",
  "payload": {
    "order_id": "order-uuid-456",
    "user_id": "user-uuid-789",
    "items": [
      {
        "product_id": "prod-uuid-111",
        "variant_id": "var-uuid-222",
        "quantity": 2,
        "price": 899.99
      }
    ],
    "total_amount": 1799.98
  }
}
```

5.2. Synchronous Communication Patterns

5.2.1. RESTful API Calls

REST will be the primary protocol for synchronous communication between services. Each service will expose a set of RESTful endpoints for operations that require an immediate response. For example, the Cart Service will make a synchronous REST call to the Inventory Service to check stock availability before adding an item to the cart. The APIs will be designed to be stateless and will use standard HTTP methods (GET, POST, PUT, DELETE). The request and response payloads will be in JSON format. The use of a service mesh like Istio will provide features like load balancing, service discovery, and security (mTLS) for these REST calls.

5.2.2. gRPC for High-Performance RPC

For performance-critical synchronous communication, gRPC will be used. gRPC is a high-performance RPC framework that uses Protocol Buffers for serialization, which is more efficient than JSON. This makes it ideal for low-latency, high-throughput communication between services. For example, the Order Service might use gRPC to communicate with the Payment Service to process a payment, as this is a critical operation that requires a fast and reliable response. The use of gRPC will be limited to specific use cases where performance is a major concern.

5.3. Resilience and Reliability Patterns

5.3.1. Circuit Breaker and Retry Mechanisms

To handle transient failures and prevent cascading failures, the system will implement the **Circuit Breaker** and **Retry** patterns. The Circuit Breaker pattern will be implemented using a library like **Resilience4j**. When a service makes a synchronous call to another service, the circuit breaker will monitor the success and failure rates. If the failure rate exceeds a certain threshold, the circuit breaker will "open," and subsequent calls to the failing service will be automatically failed without waiting for a timeout. This prevents the calling service from being blocked and gives the failing service time to recover. The Retry pattern will be used in conjunction with the circuit breaker to automatically retry failed operations a certain number of times with an exponential backoff strategy.

5.3.2. Outbox Pattern for Event Reliability

To ensure the reliable delivery of events in the event-driven architecture, the **Outbox pattern** will be used in services that use PostgreSQL. When a service needs to perform a database transaction and publish an event, these two actions must be atomic. The

Outbox pattern achieves this by storing the event in an "outbox" table within the same database transaction as the business data update. A separate process, such as a Kafka Connect connector or a custom poller, then reads from the outbox table and publishes the events to Kafka. This ensures that even if the service crashes after committing the database transaction but before publishing the event, the event will not be lost. It will be picked up by the outbox processor once the service restarts.

6. Machine Learning Recommendation System

6.1. Architecture Overview

6.1.1. Data Ingestion from Kafka

The ML recommendation system will ingest data from various Kafka topics to build and update user and product profiles. The system will consume events like `product.viewed` , `product.added_to_cart` , `order.placed` , and `review.created` . This data will be used to understand user preferences, product popularity, and relationships between products. The ingestion process will be handled by a dedicated data ingestion service that will read from the Kafka topics and write the data to a data lake or a feature store.

6.1.2. Batch Processing with Apache Spark

Apache Spark will be used for large-scale data processing and feature engineering. Spark jobs will be run periodically (e.g., daily or hourly) to process the raw event data and generate features for the machine learning models. This can include calculating user-item interaction matrices, computing product similarity scores, and aggregating user behavior over time. Spark's distributed computing capabilities make it well-suited for handling the large volumes of data generated by the e-commerce platform.

6.1.3. Pipeline Orchestration with Apache Airflow

Apache Airflow will be used to orchestrate the entire machine learning pipeline. Airflow will be used to schedule and monitor the Spark jobs for data processing, the model training jobs, and the model deployment process. Airflow will provide a visual interface for managing the pipeline and will ensure that the entire process runs smoothly and reliably. The pipeline will be defined as a Directed Acyclic Graph (DAG) in Airflow, which will specify the dependencies between the different tasks.

6.2. Model Training and Serving

6.2.1. Collaborative Filtering and Embeddings

The recommendation models will be built using a combination of techniques, including **collaborative filtering** and **embedding-based models**. Collaborative filtering models, such as matrix factorization, will be used to identify patterns in user-item interactions. Embedding-based models will be used to learn representations of users and products in a low-dimensional space, which can then be used to calculate similarity scores. The models will be trained using the features generated by the Spark jobs.

6.2.2. Model Storage and Versioning

The trained models will be stored in a model registry, such as **MLflow** or **SageMaker Model Registry**. This will provide a centralized repository for managing the models, including versioning, staging, and deployment. The model registry will make it easy to track the performance of different model versions and to roll back to a previous version if necessary.

6.2.3. Recommendation API Endpoint

The Recommendation Service will expose a RESTful API endpoint, such as `GET /api/v1/recommendations?userId=...`, to serve recommendations to the frontend application. When a request is received, the service will retrieve the user's profile and the pre-computed recommendations from a cache or a database. The service will then return a list of recommended products to the frontend. The service will be designed to be highly scalable and performant, with a low-latency response time.

7. Design Patterns

7.1. CQRS (Command Query Responsibility Segregation)

The **CQRS** pattern will be used in services where there is a significant difference between the read and write workloads. For example, in the Product Catalog Service, the write operations (creating and updating products) might have a different set of requirements than the read operations (searching and browsing products). By separating the command (write) and query (read) models, each can be optimized independently. The write model can be designed for consistency and transactional integrity, while the read model can be designed for performance and scalability. This pattern can be implemented by having separate databases for reads and writes, with the read model being updated asynchronously through events.

7.2. Saga Pattern for Distributed Transactions

The **Saga pattern** will be used to manage distributed transactions that span multiple services, such as the order placement process. A saga is a sequence of local transactions, where each step is a transaction in a single service. If any step fails, a series of compensating transactions are executed to undo the changes made by the previous steps. This ensures that the system remains in a consistent state, even in the face of failures. The saga can be orchestrated by a central coordinator (the Order Service) or can be choreographed, with each service publishing events that trigger the next step in the saga.

7.3. Event Sourcing

Event Sourcing is a pattern where the state of an application is determined by a sequence of events. Instead of storing the current state of an entity, the system stores a log of all the events that have affected that entity. The current state can then be derived by replaying the events. This pattern provides a complete audit trail of all changes and makes it easy to rebuild the state of the system at any point in time. This pattern can be used in services like the Order Service, where it is important to have a complete history of all the changes to an order.

7.4. Outbox Pattern

The **Outbox pattern** will be used to ensure the reliable delivery of events in the event-driven architecture. When a service needs to perform a database transaction and publish an event, these two actions must be atomic. The Outbox pattern achieves this by storing the event in an "outbox" table within the same database transaction as the business data update. A separate process then reads from the outbox table and publishes the events to Kafka. This ensures that even if the service crashes after committing the database transaction but before publishing the event, the event will not be lost.

7.5. DTO vs. Entity Separation

The **DTO (Data Transfer Object) vs. Entity separation** pattern will be used to decouple the internal domain model of a service from the data that is exposed to the outside world. Entities will represent the core business objects and will be used for internal business logic. DTOs will be used to transfer data between services and to the frontend. This separation allows the internal domain model to evolve independently of the external API, providing greater flexibility and maintainability.

7.6. Database per Service

The **Database per Service** pattern is a fundamental principle of the microservices architecture. Each microservice will have its own dedicated database, and direct access to another service's database will be strictly forbidden. This ensures that services are loosely coupled and can be developed, deployed, and scaled independently. This pattern also allows each service to choose the most appropriate database technology for its specific needs, a concept known as polyglot persistence.

8. API Specification (OpenAPI-Compliant)

8.1. Product Service Endpoints

表格 复制

Method	Path	Description
`GET`	`/api/v1/products`	Get a paginated list of all products.
`GET`	`/api/v1/products/{id}`	Get detailed information for a specific product.
`POST`	`/api/v1/products`	Create a new product (Admin only).
`PUT`	`/api/v1/products/{id}`	Update an existing product (Admin only).
`DELETE`	`/api/v1/products/{id}`	Delete a product (Admin only).

8.2. Cart Service Endpoints

表格 复制

Method	Path	Description
`GET`	`/api/v1/cart/{userId}`	Get the contents of a user's cart.
`POST`	`/api/v1/cart/{userId}/items`	Add an item to a user's cart.
`PUT`	`/api/v1/cart/{userId}/items/{itemId}`	Update the quantity of an item in t
`DELETE`	`/api/v1/cart/{userId}/items/{itemId}`	Remove an item from the cart.

8.3. Order Service Endpoints

表格 复制

Method	Path	Description
POST	`/api/v1/orders`	Create a new order from the user's cart.
GET	`/api/v1/orders/{orderId}`	Get detailed information for a specific or
GET	`/api/v1/users/{userId}/orders`	Get a list of orders for a specific user.
PUT	`/api/v1/orders/{orderId}/cancel`	Cancel an order.

8.4. Search Service Endpoints

表格 复制

Method	Path	Description
GET	`/api/v1/search`	Perform a full-text search with filters and sort
GET	`/api/v1/search/suggestions`	Get autocomplete suggestions for a search qu

8.5. Recommendation Service Endpoints

表格 复制

Method	Path	Description
GET	`/api/v1/recommendations`	Get personalized product recommendations for

9. Deployment Strategy on AWS

9.1. Kubernetes-Native Deployment

9.1.1. Amazon EKS for Managed Kubernetes

The entire system will be deployed on **Amazon Elastic Kubernetes Service (EKS)** , a managed Kubernetes service that simplifies the process of running Kubernetes on AWS. EKS will handle the provisioning and management of the Kubernetes control plane, allowing the development teams to focus on deploying and managing their applications. The worker nodes, which run the containerized microservices, will be provisioned on **Amazon EC2** instances. The use of EKS provides a highly available and scalable foundation for the e-commerce platform.

9.1.2. Helm Charts for Service Deployment

Helm will be the primary tool for packaging and deploying the microservices and infrastructure components to the EKS cluster. Each microservice will have its own Helm chart, which will define the Kubernetes resources (Deployments, Services, ConfigMaps, etc.) needed to run the service. Helm charts will also be used to deploy the supporting infrastructure, such as Kafka, PostgreSQL, MongoDB, and Redis. This approach makes the deployments more reproducible, versionable, and easier to manage.

9.2. Ingress and Service Mesh

9.2.1. NGINX Ingress Controller

An **NGINX Ingress Controller** will be deployed in the EKS cluster to manage external access to the services. The Ingress Controller will act as a reverse proxy and load balancer, routing traffic from the internet to the appropriate microservice based on the URL path. This provides a single entry point for all external client requests and simplifies the management of external traffic.

9.2.2. TLS Termination with cert-manager

cert-manager will be used to automate the management of TLS certificates for the Ingress Controller. cert-manager will integrate with a certificate authority like **Let's Encrypt** to automatically provision and renew TLS certificates. This will ensure that all communication between the clients and the API Gateway is encrypted using HTTPS, providing a secure connection for users.

9.3. Auto-Scaling

9.3.1. Horizontal Pod Autoscaler (HPA)

The **Horizontal Pod Autoscaler (HPA)** will be configured for each microservice to automatically scale the number of running pods based on resource utilization. The HPA will monitor metrics like CPU and memory usage and will increase or decrease the number of pods to meet the current demand. This ensures that the services can handle traffic spikes without manual intervention and helps to optimize resource usage.

9.3.2. Cluster Autoscaler

The **Cluster Autoscaler** will be configured to automatically scale the number of worker nodes in the EKS cluster. The Cluster Autoscaler will monitor the resource requests of the pods and will add new nodes to the cluster if there are not enough resources to schedule all the pods. Conversely, it will remove nodes from the cluster if they are underutilized. This provides a truly elastic infrastructure that can grow and shrink with the demand of the application.

10. 1-Month Development and Deployment Plan

This plan outlines a 4-week sprint to develop and deploy the core backend services of the e-commerce platform. The plan assumes a team structure where each team is responsible for a separate microservice.

10.1. Week 1: Foundation and Core Services

10.1.1. Day 1-2: Infrastructure Setup

- **DevOps Team:** Set up the AWS account and create the foundational infrastructure, including the VPC, subnets, and security groups.
- **DevOps Team:** Provision the Amazon EKS cluster and configure the worker nodes.
- **DevOps Team:** Deploy the core infrastructure components using Helm, including Kafka, PostgreSQL, MongoDB, and Redis.
- **DevOps Team:** Set up the CI/CD pipelines using GitHub Actions and ArgoCD.

10.1.2. Day 3-4: User and Product Catalog Services

- **User Service Team:** Implement the User Service with basic registration and login functionality.
- **User Service Team:** Set up the PostgreSQL database schema for the User Service.
- **Product Catalog Service Team:** Implement the Product Catalog Service with basic CRUD operations for products.
- **Product Catalog Service Team:** Set up the MongoDB database for the Product Catalog Service.

10.1.3. Day 5: API Gateway and Authentication

- **API Gateway Team:** Deploy and configure the API Gateway (e.g., Spring Cloud Gateway).
- **API Gateway Team:** Integrate the API Gateway with the User Service for authentication.

- **API Gateway Team:** Define the initial set of routes for the User and Product Catalog services.

10.2. Week 2: Shopping Cart and Orders

10.2.1. Day 6-7: Cart and Order Service Implementation

- **Cart Service Team:** Implement the Cart Service with Redis for data storage.
- **Cart Service Team:** Implement the REST endpoints for adding, updating, and removing items from the cart.
- **Order Service Team:** Implement the Order Service with PostgreSQL for data storage.
- **Order Service Team:** Implement the REST endpoint for creating a new order.

10.2.2. Day 8-9: Payment Service Integration

- **Payment Service Team:** Implement the mock Payment Service.
- **Payment Service Team:** Integrate the Payment Service with the Order Service using the Saga pattern.
- **Payment Service Team:** Implement the Kafka event publishing for payment success and failure.

10.3. Week 3: Search, Inventory, and Shipping

10.3.1. Day 10-11: Search Service Implementation

- **Search Service Team:** Deploy and configure the OpenSearch cluster.
- **Search Service Team:** Implement the Search Service to index products from the Product Catalog Service.
- **Search Service Team:** Implement the REST endpoints for full-text search and autocomplete.

10.3.2. Day 12-13: Inventory Service Implementation

- **Inventory Service Team:** Implement the Inventory Service with PostgreSQL for data storage.

- **Inventory Service Team:** Integrate the Inventory Service with the Order Service using the Saga pattern.
- **Inventory Service Team:** Implement the Kafka event publishing for inventory updates.

10.3.3. Day 14–15: Shipping Service Implementation

- **Shipping Service Team:** Implement the mock Shipping Service.
- **Shipping Service Team:** Integrate the Shipping Service with the Order Service using the Saga pattern.
- **Shipping Service Team:** Implement the Kafka event publishing for shipping updates.

10.4. Week 4: Recommendations, Reviews, and Final Deployment

10.4.1. Day 16–17: Review Service Implementation

- **Review & Rating Service Team:** Implement the Review & Rating Service with MongoDB for data storage.
- **Review & Rating Service Team:** Implement the REST endpoints for submitting and retrieving reviews.
- **Review & Rating Service Team:** Integrate the Review & Rating Service with the Product Catalog Service using Kafka events.

10.4.2. Day 18–19: Recommendation Service PoC

- **ML Team:** Set up the ML pipeline infrastructure, including Apache Spark and Apache Airflow.
- **ML Team:** Implement a proof-of-concept for the Recommendation Service using a simple collaborative filtering model.
- **ML Team:** Implement the REST endpoint for serving recommendations.

10.4.3. Day 20–30: Integration, Testing, and Staging Handover

- **All Teams:** Perform end-to-end integration testing of the entire system.
- **QA Team:** Perform comprehensive testing, including functional, performance, and security testing.

- **DevOps Team:** Deploy the system to a staging environment that mirrors the production environment.
- **All Teams:** Fix any bugs and issues identified during testing.
- **All Teams:** Prepare the system for production deployment and handover to the operations team.