Software Design and Architecture
Milestone 1

Requirements Analysis:

A) CORE FUNCTIONAL REQUIREMENTS:

1. Inventory Management: Track inventory levels, manage stock across warehouses, and issue alerts for low or full stock levels.
2. Supplier Management: Manage supplier information and track supplier performance and delivery times.
3. Order Processing: Handle customer orders from order placement through fulfillment, including order tracking and updates.
4. Demand Forecasting: Analyze historical sales data to predict future demand and assist in inventory planning.
5. Logistics and Shipping Management: Track shipments from suppliers to warehouses and from warehouses to customers.
6. Quality Control: Set up quality checkpoints for incoming materials and outgoing products. Log defects, issue recalls if necessary, and manage rework processes.
7. Analytics and insights: Provide analytics to support decision-making, identify bottlenecks, and efficient metrics for decision-making.
8. User Access Control: Set role-based access controls, ensuring each user has access only to relevant information and tasks.
9. Procurement Management: Enable purchase order creation, approval workflows, and tracking of order statuses with suppliers.
10. Warehouse Management: Manage locations, storage bins, and space optimization in each warehouse.
11. Returns Management: Facilitate the return process, including approval, tracking, and restocking procedures.
12. Real-Time Reporting: Generate real-time reports on inventory, orders, supplier performance, and logistics costs.
13. Financial Integration: Integrate with accounting to handle transactions like invoicing, payments, and reconciliation.
14. Barcode Scanning: Support barcode scanning to streamline inventory checks, receiving, and shipping processes.
15. Product Lifecycle Management: Manage product details throughout its lifecycle, including creation, updates, and phase-out.
16. Supplier Collaboration Portal: Provide a portal for suppliers to access order statuses, forecasted demand, and performance feedback.

B) Non-Functional Requirements

1. Scalability: The system should handle increased demand (e.g., more suppliers, orders, and inventory items) without performance degradation.
2. Reliability: Ensure high availability with minimal downtime, critical for just-in-time manufacturing.
3. Performance: Fast response times for core operations like inventory lookups and order processing, even with large datasets.
4. Security: Implement encryption for data transmission and storage, user authentication, and authorization.
5. Maintainability: Modular architecture to allow easy updates or replacements of components, e.g., switching out a supplier API.
6. Usability: An intuitive interface for easy use by factory personnel and suppliers, minimizing training needs.
7. Compliance: Adherence to industry regulations and standards, such as ISO certifications for quality management.
8. Data Integrity: Ensure data accuracy, especially in inventory counts and order processing.
9. Availability: The system should have an uptime of 99.9%, ensuring it is always available to users, especially during business hours.
10. Localization: The system should support multiple languages and currencies to serve a global user base, with the ability to display appropriate local formats for dates, times, and currency symbols.

Conceptual Design:

Event-Driven Design:

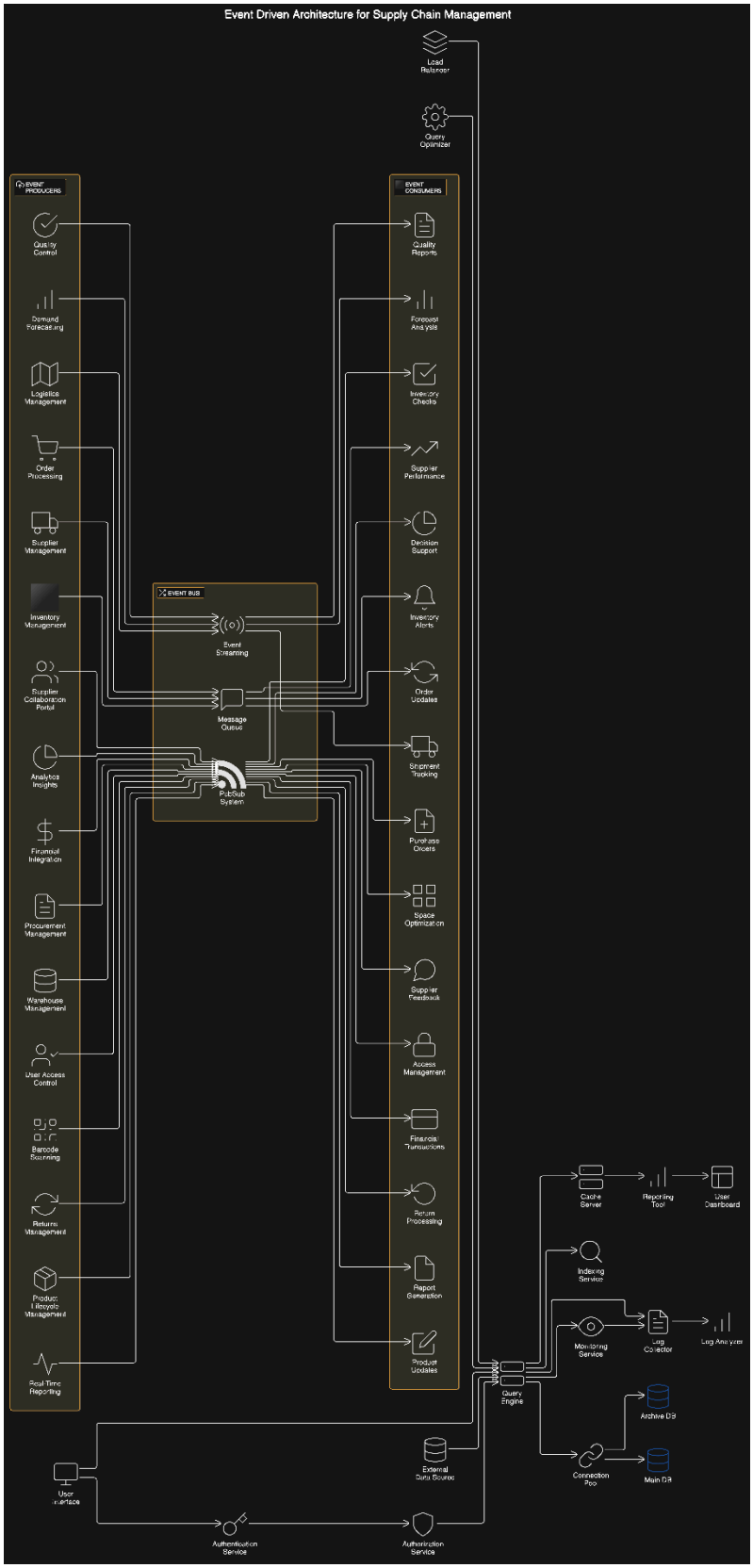
1. Event Producers and Consumers: Producers generate events when something happens, such as a new order placed, inventory updated, or a shipment dispatched. Consumers are services or components that subscribe to specific event types and act on them.
2. Event Bus or Broker: Events flow through an event bus. Producers publish events, and consumers subscribe to relevant topics.
3. Event Types: Examples include Order-Placed, Stock-Low-Alert, Shipment-Dispatched, Return-Initiated, and so on.
4. Microservices Communication: Each core function is a separate microservice, loosely coupled via events. They only interact indirectly by subscribing to and publishing events.
5. Real-Time Updates: As events are processed, they system can update dashboards, notify users, or trigger workflows in real time.
6. Fault Tolerance: Events are persisted in the event bus, ensuring that they are not lost if a consumer is temporarily unavailable.

Microservices Design:

The System's services can be separated as follows:

1. Inventory Service: Track inventory levels, updates stock, and triggers alerts for low/full stock levels.
2. Supplier Service: Manages supplier information, tracks performance, and handles purchase orders.
3. Order Service: Processes customers orders, tracks their status, and manages fulfillment.
4. Forecasting Service: Predicts future demand based on sales history and generates inventory recommendations.
5. Logistics Service: Tracks inbound and outbound shipments and integrates with external shipping APIs.
6. Quality Service: Handles quality control processes, log defects, and manages recalls or reworks.
7. Analytics Service: Consolidates data from other services for insight and decision support.
8. User Management Service: Manages user authentication, authorization, and role-based access controls.

Event-Driven Diagram:


Event Driven Architecture for Supply Chain Management
Load Balancer
Query Optimizer
EVENT PRODUCERS
Quality Control
Demand Forecasting
Logistics Management
Order Processing
Supplier Management
Inventory Management
Supplier Collaboration Portal
Analytics Insights
Financial Integration
Procurement Management
Warehouse Management
User Access Control
Barcode Scanning
Returns Management
Product Lifecycle Management
Real-Time Reporting
EVENT BUS
Event Streaming
Message Queue
Pub/Sub System
EVENT CONSUMERS
Quality Reports
Forecast Analysis
Inventory Checks
Supplier Performance
Decision Support
Inventory Alerts
Order Updates
Shipment Tracking
Purchase Orders
Space Optimization
Supplier Feedback
Access Management
Financial Transactions
Return Processing
Report Generation
Product Updates
Cache Server
Reporting Tool
User Dashboard
Indexing Service
Monitoring Service
Log Collector
Log Analyzer
Query Engine
Archive DB
Connection Pool
Main DB
External Data Source
User Interface
Authentication Service
Authorization Service

Software Decomposition:

1-Functional Decomposition

1. Order Management

Order management includes all operations related to customer orders, from creation to fulfillment. This would cover the following tasks:

- **Order Creation**: Capture customer details, items, quantities, and other relevant information to create a new order.
    - Inputs: Customer data, item selection, quantities, payment method.
    - Outputs: Created order with assigned order ID.
- **Order Validation**: Check if the order is valid (e.g., available stock, payment processed).
    - Inputs: Order details.
    - Outputs: Order status (valid/invalid), payment confirmation.
- **Order Fulfillment**: Process the order for shipment.
    - Inputs: Validated order details, stock availability.
    - Outputs: Shipping instructions, updates to inventory.
- **Order Status Tracking**: Enable tracking of orders, updating statuses (processing, shipped, delivered, etc.).
    - Inputs: Order ID, shipping status.
    - Outputs: Updated order status.

---

2. Inventory Management

This function focuses on the tracking, storing, and managing stock levels across multiple warehouses and locations. It includes:

- **Inventory Tracking**: Track current inventory levels, including quantity and locations.
    - Inputs: Shipments, order fulfillments, returns.
    - Outputs: Updated stock levels.
- **Stock Movement Management**: Manage the movement of stock between different locations (e.g., warehouse to warehouse, warehouse to customer).
    - Inputs: Stock movements, shipments.
    - Outputs: Updated inventory records.

- **Stock Forecasting**: Predict future stock needs based on historical data and trends.
    - Inputs: Historical sales data, trends, current stock levels.
    - Outputs: Predicted stock levels for future periods.
- **Low/Full Stock Alerts**: Generate alerts when stock reaches predefined low or high levels.
    - Inputs: Current stock levels, threshold values.
    - Outputs: Alert notifications.

---

3. Supplier Management

Supplier management encompasses managing supplier relationships, orders, and performance. It includes:

- **Supplier Registration**: Create and maintain supplier profiles with relevant information.
    - Inputs: Supplier details (name, contact, materials supplied).
    - Outputs: Supplier record.
- **Supplier Order Management**: Handle purchase orders placed with suppliers for replenishment.
    - Inputs: Order requests, supplier details.
    - Outputs: Supplier orders, status updates.
- **Supplier Performance Tracking**: Monitor and track the performance of suppliers based on metrics like delivery time, quality, and reliability.
    - Inputs: Delivery logs, quality reports.
    - Outputs: Performance metrics, reports.
- **Supplier Collaboration Portal**: Provide suppliers access to order statuses, forecasted demand, and feedback.
    - Inputs: Supplier requests, orders, feedback.
    - Outputs: Accessed portal, reports.

---

4. Logistics and Shipping Management

This function manages the physical movement of goods, including tracking shipments from suppliers and to customers.

- **Shipment Tracking**: Track and update the status of shipments from suppliers to warehouses and from warehouses to customers.
    - Inputs: Shipment details, delivery status.
    - Outputs: Shipment status updates, delivery confirmation.
- **Shipping Route Optimization**: Plan and optimize delivery routes based on factors like distance, delivery urgency, and inventory availability.
    - Inputs: Delivery locations, stock needs, transportation data.
    - Outputs: Optimized delivery routes.
- **Logistics Cost Tracking**: Monitor and report on logistics-related costs.
    - Inputs: Shipping data, route costs, fuel costs.
    - Outputs: Cost reports.

---

5. Quality Control

This function ensures that all incoming and outgoing goods meet the required quality standards.

- **Quality Check on Incoming Materials**: Perform checks on raw materials received from suppliers.
    - Inputs: Supplier shipments, quality criteria.
    - Outputs: Quality reports, defect logs.
- **Quality Check on Finished Goods**: Inspect finished products before they are shipped to customers.
    - Inputs: Final products, quality standards.
    - Outputs: Quality reports, defect logs.
- **Defect Tracking and Rework**: Log defects and manage rework procedures.
    - Inputs: Defective products, corrective actions.
    - Outputs: Updated product status, reworked goods.
- **Product Recall Management**: If necessary, initiate product recalls and track affected inventory.
    - Inputs: Product defects, customer reports.
    - Outputs: Recall orders, affected product lists.

---

6. Analytics and Reporting

This function enables decision-making by providing critical data insights.

- **Real-Time Reporting**: Provide live updates on inventory levels, orders, shipments, and performance metrics.
    - Inputs: Real-time data from various sources (orders, inventory, etc.).
    - Outputs: Dynamic reports, dashboards.
- **Bottleneck Identification**: Analyze the supply chain to identify bottlenecks in inventory, order processing, or logistics.
    - Inputs: Performance data from all business functions.
    - Outputs: Identified bottlenecks, improvement suggestions.
- **Data Analytics and Insights**: Analyze historical data to predict future trends and optimize processes.
    - Inputs: Historical data (sales, stock, orders).
    - Outputs: Trend reports, forecasts.

---

7. User Access Control

This function manages user roles and permissions within the system.

- **Role-Based Access Control (RBAC)**: Set permissions for different roles (admin, warehouse worker, supplier, etc.).
    - Inputs: User roles, permissions data.
    - Outputs: Access control policies.
- **User Authentication**: Manage login processes, authentication, and session control.
    - Inputs: User credentials.
    - Outputs: Authentication tokens, session data.
- **User Authorization**: Ensure that users only access the data and functions they are authorized to.
    - Inputs: Role data, user actions.
    - Outputs: Access restrictions, role-specific permissions.

---

8. Financial Integration

Integrates with accounting systems to handle invoicing, payments, and financial transactions.

- **Invoice Creation**: Generate invoices for orders based on customer and product details.
    - Inputs: Order details, pricing.
    - Outputs: Generated invoice.
- **Payment Processing**: Process payments from customers, including handling various payment methods.
    - Inputs: Payment data (credit card, bank transfer, etc.).
    - Outputs: Payment status, confirmation.
- **Transaction Reconciliation**: Reconcile order payments with financial records.
    - Inputs: Order payments, bank transaction data.
    - Outputs: Reconciled records.

---

9. Returns Management

Facilitates the return process for customers and suppliers.

- **Return Authorization**: Approve or reject return requests based on policies.
    - Inputs: Return request, product condition.
    - Outputs: Return status.
- **Return Processing**: Manage the actual return process, including updating stock and processing refunds.
    - Inputs: Approved return requests.
    - Outputs: Updated inventory, refunds issued.
- **Restocking**: Restock returned goods if applicable (if they are in sellable condition).
    - Inputs: Returned products, restocking decision.
    - Outputs: Restocked inventory.

2-Object - Oriented Decomposition

1. Order Management System

- **Order**: Represents a customer order.
    - Attributes: orderID , customerID , orderItems[] , paymentStatus , status

    - Methods: createOrder() , validateOrder() , fulfillOrder() , trackStatus() ,calculateTotalPrice()
- **Customer**: Represents customer details.
    - Attributes: customerID , name , address , contactDetails
    - Methods: updateProfile() , viewOrderHistory()

---

2. Inventory Management System

- **Product**: Represents an individual product type in the inventory.
    - **Attributes**: productID , name , description , category , price , stockLevel ,alertLevel ,minThreshold (The minimum stock level to trigger a low-stock alert (e.g., 10 units)), maxThreshold (The maximum stock level to trigger an overstock alert (e.g., 100 units))
- **Methods**:
    - updateStock() : Updates the stock level of the product.
    - getStockLevel() : Returns the current stock level of the product.
    - getPrice() : Returns the price of the product.
    - isLowStock() : Checks if the stock level is below the minimum threshold.
    - isOverstocked() : Checks if the stock level is above the maximum threshold.
    - setAlertLevel() : Set an alert for the stock level of the product (if necessary)
- **Item**: Represents an item stored in a warehouse of a Product type
    - **Attributes**: itemID ,warehouseID , productID , capacity ,qualityDescription
    - **Methods**:
        - getLocation() : Retrieves which warehouse the item is located in
        - getType() : Retrieves the Product type of the item
        - updateQualityDescription() : Updates the quality description of the item during quality checks
        - getPrice() : Retrieves the price of the item from the respective product type
- **Warehouse**: Represents a warehouse where products are stored.
    - **Attributes**: warehouseID , location , capacity , itemList[] (list of all individual items in the warehouse) , productList (list of all different products in the warehouse along with their quantities)
    - **Methods**:
        - addItem() : Adds an item to the warehouse itemList and handles the logic for productList .

        - removeItem() : Removes an item to the warehouse itemList and handles the logic for productList .
        - getItems() : Returns the list of items in the warehouse.
        - getProducts() : Returns the list of products alongside their quantities in the warehouse.
- **Stock**: Represents the overall stock system that spans across multiple warehouses and tracks inventory levels across the entire system.
    - **Attributes**: stockList[] (list of all products across all warehouses),warehouseList[] (list of all warehouses to transfer products between warehouses)
    - **Methods**:
        - trackInventory() : Updates and tracks the stock levels across the entire system.
        - updateAlerts() : Checks all products against their minThreshold and maxThreshold and generates alerts for low or high stock. It then updates the alert level of the product.
        - transferProducts() : Transfers products between warehouses.

---

3. Supplier Management

- **Supplier**: Represents a supplier.
    - Attributes: supplierID , name , contactInfo , materialsSupplied
    - Methods: registerSupplier() , updateSupplier()
- **SupplierOrder**: Represents an order placed with a supplier.
    - Attributes: orderID , supplierID , orderDetails , status
    - Methods: placeOrder() , trackOrderStatus()
- **SupplierPerformance**: Tracks supplier performance metrics.
    - Attributes: supplierID , deliveryTime , qualityRating , reliability
    - Methods: generateReport()

---

4. Logistics and Shipping Management

- **Shipment**: Represents a shipment.

    - Attributes: shipmentID , origin , destination , status , estimatedDeliveryTime
    - Methods: trackShipment() , updateShipmentStatus()
- **Route Optimization**: Optimizes delivery routes.
    - Attributes: origin , destination , deliveryUrgency , routeOptions[]
    - Methods: optimizeRoute()
- **Logistics Cost**: Tracks logistics costs.
    - Attributes: shipmentID , transportationCost
    - Methods: calculateCost()

---

5. Quality Control

- **Quality Check**: Performs quality checks.
    - Attributes: itemId , qualityCriteria , result
    - Methods: performCheck() , logDefects()
- **Product Recall**: Manages product recalls.
    - Attributes: recallID , recallDate , affectedInventory(list of items being recalled)
    - Methods: initiateRecall() , trackRecall()

---

6. Analytics and Reporting

- **Report**: Represents a report generated from the system.
    - Attributes: reportID , dataSource , reportType , generatedAt
    - Methods: generateReport() , viewReport()
- **Analytics**: Handles data analysis for trends and bottlenecks.
    - Attributes: data[] , analysisID , forecast , bottlenecks ,insights
    - Methods: identifyBottlenecks() , generateForecast() , analyzeData()

---

7. User Access Control

- **User**: Represents a system user.
    - Attributes: userID , role , loginDetails

    - Methods: authenticate() , authorize()
- **Role**: Represents different roles in the system (e.g., admin, warehouse worker).
    - Attributes: roleID , roleName , permissions[]
    - Methods: assignRole() , modifyRole()

---

8. Financial Integration

- **Invoice**: Represents an invoice generated for orders.
    - Attributes: invoiceID , orderID , totalAmount , paymentStatus ,customerId ,orderDescription
    - Methods: createInvoice() , sendInvoice()
- **Payment**: Represents a payment transaction.
    - Attributes: paymentID , orderID , amount , paymentMethod , status
    - Methods: processPayment() , confirmPayment()

---

9. Returns Management

- **Return Request**: Represents a customer's return request.
    - Attributes: returnID , orderID , itemID , reason
    - Methods: approveReturn() , rejectReturn()
- **Return Processing**: Manages the return process.
    - Attributes: returnID , refundStatus , restockingStatus
    - Methods: processReturn() , updateInventory()

Domain-Driven Design:

<u>Inventory Context:</u>
This context will handle the stocks levels,updates and reordering.
Product(Entity) : represents the item produced.
Aggregate_Product: containing the product along with their characteristics.
Stock levels(Value Object): indicates the quantity and the threshold reordering.
Warehouse(Entity): locations of where the supplier's products are stocked.

Events:

- StockEmpty: Triggered when product stock is empty.
- StockRestocked: To know when new stock is added.

Services:

- InventoryService: Manages stock level checks, restocking, and adjustments based on orders.

Ordering Context:
Manages customer orders, order details, and status
Order(Entity): customer's order
Aggregate_Order: manages the order while including their details, price, ordering status and updates
OrderItem(Value Object): show the item that is being ordered by its ID and the quantity requested.
Events:

- OrderPlacedEvent: Triggered when a new order is created.
- OrderFulfilled: Emitted once the order has been delivered.
- OrderCanceled: Generated if an order is cancelled.

Services:

- OrderService: Provides methods for creating, updating, and cancelling orders.

Shipping Context:
Oversees the shipping and the delivery process.
Shipment(Entity): tracks delivery status, shipping method and estimated delivery date.
Aggregate_Shipment: Manages shipments and delivery schedules.
Address(Value Object): address of the distributor that made the order.
Tracking Details(Value Object): Captures tracking milestones or events along the shipping route.
Events:

- ShipmentInitiatedEvent: Signals that shipping has started.
- DeliveryConfirmedEvent: Confirms delivery completion.

ShipmentService:

- Coordinates shipments, estimated delivery times, and integrates with third-party shipment providers if necessary.

Supplier context:
Manages supplier interactions for procurement and restocking.
Supplier(Entity): main representation of the supplier, includes ID, name and contact details.
Contact Info(Value Object): communication info of the supplier
Product Offering(Value Object): Details the products a supplier can provide, including prices and minimum order quantities.
Events:

- RestockRequested: Emitted when an inventory restock request is made to a supplier.
- ProductOfferingUpdated: Triggered when a supplier updates available product offerings.
- SupplierOrderFulfilled: Emitted when the supplier fulfils an order.

Services:

- SupplierService: Coordinates with suppliers for restocking and procurement.

Customer context:
Manages the interactions between the retailer and their supplier.
Customer(Entity): Represents the retailer by using its ID, location, name and order history.
CustomerOrder(Aggregate): contains the items ordered along with the quantity, price, order ID, User ID and date of the order.
CustomerContactInfo(Value Object): Communication info of the customer.
CustomerOrderDetails(Value Object): Contains details of an order placed by the customer, such as quantities and products.
Events:

- InventoryRequestCreated: Done when a customer places an inventory request.
- OrderShippedToCustomer: Triggered when an order is shipped to a customer.

Services:

- CustomerService: Provides services for inventory requests, order placements, and order tracking.

Payment context:
Handles the payment transactions
PaymentTransaction(Aggregate): represents the core payment process, including initiation, authorization, and completion.
PaymentMethod(Entity): contains payment options.
Invoice(Entity): represents billing details for transactions.
TransactionID(Value Object): unique payment ID.
Amount(Value Object): value and the ordered products.
PaymentStatus(Value Object): indicates whether it's “pending” or “completed”.
Events:

- PaymentInitiated**:** Payment process started.
- PaymentCompleted**:** Transaction successful.
- PaymentFailed**:** Transaction failed.

Services:

PaymentGatewayService**:** Processes payments with external providers.

Documentation:

Justification for Using Event-Driven Design:

1. Scalability: EDA supports horizontal scaling. New services can be added or scaled independently without disrupting others, making it ideal for handling increased demand.
2. Decoupling: Microservices don’t need to know about each other's implementation. For example, the inventory service only needs to consume OrderPlaced events without knowing the details of the order processing logic.
3. Real-Time Processing: Critical for features like real-time reporting, low-stock alerts, and order tracking. EDA ensures near-instantaneous propagation of updates.
4. Flexibility: Adding new features (e.g., predictive analytics based on events) becomes straightforward. Simply introduce a new service to consume the relevant events.
5. Reliability: Events are persisted in a durable log, enabling recovery in case of failures. Systems can replay events to rebuild state if needed.
6. Better Workflow Automation: EDA supports automating workflows such as notifying suppliers when stock is low, updating customers about shipment statuses, and triggering quality control checks upon receiving goods.
7. High Availability: Services operate independently. If the analytics service goes down, it doesn't affect order processing or inventory management.
8. Data-Driven Insights: Aggregating events provides rich data for analytics and demand forecasting. Real-time streams can feed machine learning models for predictive analysis.

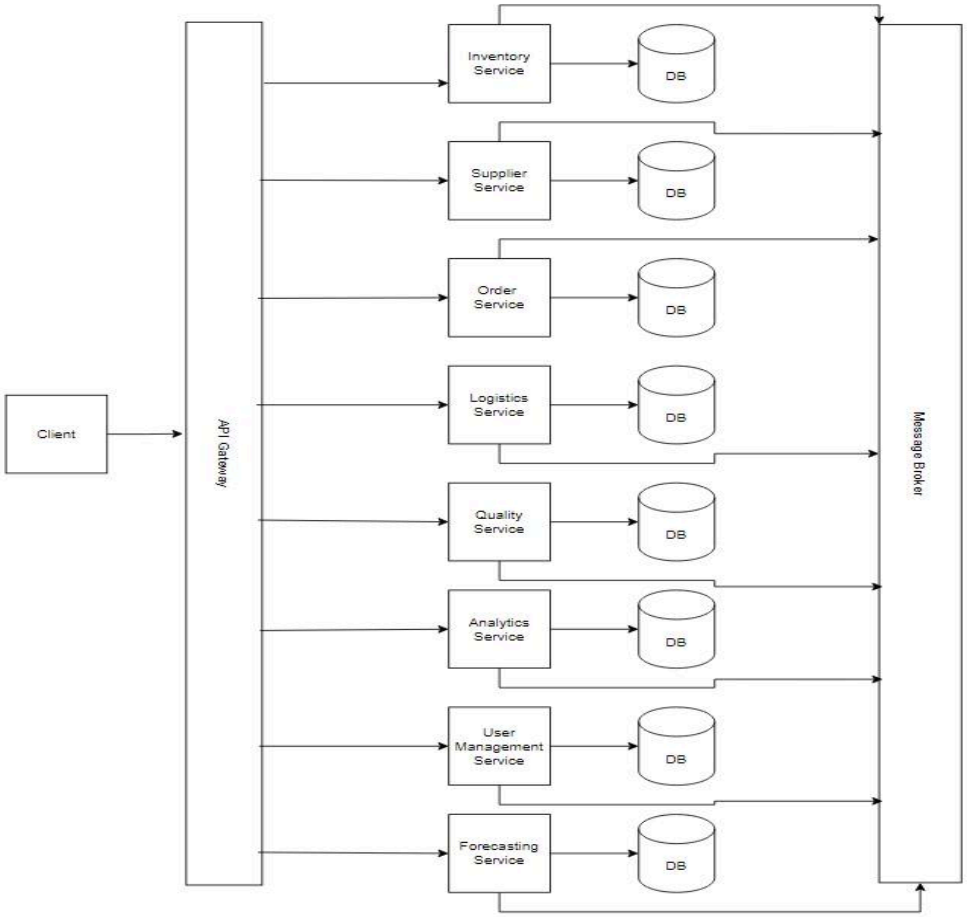
Justification for Using Microservices Design:

1. Scalability: Each service can be independently scaled based on load. For example, during a sales event, the Order Service can scale horizontally without affecting the Supplier Service.
2. Fault Isolation: If one service fails, it does not bring down the entire system. Other services continue operating.
3. Technology Independence: Each service can be built using the technology best suited to its requirements.
4. Continuous Deployment: Teams can update or deploy each service independently, reducing downtime and deployment risks.

Milestone 2

Architectural Styles and Justification and integration with milestone 1

The chosen architectural style is microservices



This microservice diagram aligns with the functional requirements of the supply chain management system where:

Alignment with FRS:

1- Decoupling: Each service focuses on a specific aspect of the supply chain (inventory, orders, suppliers, etc.). This isolation allows for independent development, deployment, and scaling of individual services. For example, if order

volume surges during peak seasons, only the Order Service needs to be scaled, not the entire system

2- Reusability: Services can be potentially reused across different parts of the system or even in other applications, reducing development effort

3-Flexibility: New features or changes to specific areas can be implemented in a modular way. This flexible implementation is crucial in a dynamic supply chain environment, where only the relevant services are affected.

Alignment with NFRS:

1- Scalability: During peak shopping seasons, the Order Service can be scaled horizontally (adding more instances) to handle the increased load, while inventory levels are managed by a separate, independently scalable Inventory Service.

2- Reliability: If the Supplier Service experiences a temporary outage, order processing can still continue, though with limited supplier information

3- Maintainability: Updating the payment processing logic within the Order Service doesn't require changes to the Inventory Service or other components

TradeOffs:

1. **Increased Complexity**

Operational Overhead: Managing a distributed system with multiple services requires more operational overhead compared to a monolithic architecture. This includes deployment, monitoring, and scaling of individual services.

2. Data Consistency

Maintaining Data Consistency: Ensuring data consistency across multiple services can be difficult. For example, updating inventory levels across multiple services simultaneously can lead to inconsistencies if not handled carefully.
3.Real-time Inventory Updates: If real-time inventory accuracy is critical, ensuring immediate consistency across services might be challenging with a microservices approach.

This implementation of the microservices diagram is to compare and contrast between the tradeoffs the above diagram offers and the event driven architecture implemented in milestone 1. The below tradeoffs are related to the EDA architecture:

Benefits:

Decoupling: EDA decouples event producers and consumers. This means services don't need to know about each other directly, improving flexibility and reducing dependencies. For example, a new logistics partner can be integrated without affecting other services.

Real-time Processing: EDA enables real-time or near real-time processing of events, which is crucial for time-sensitive operations in supply chain management, such as inventory updates and order fulfillment.

Alignment with FRs:

Order Processing: EDA enables real-time order processing and updates, ensuring timely fulfillment.

Inventory Management: Real-time inventory updates and low-stock alerts can be implemented efficiently using event-driven mechanisms.

Logistics and Shipping: EDA facilitates real-time tracking of shipments and integration with external logistics providers.

Supplier Management: Events can be used to trigger notifications and actions related to supplier performance and delivery schedules.

Alignment with NFRs:

Scalability: EDA allows for independent scaling of event producers and consumers, ensuring the system can handle increasing order volumes and data loads.

Reliability: Event-driven systems can be made more resilient by using message queues and retry mechanisms.

Maintainability: EDA can make the system more maintainable by decoupling services and allowing for independent updates.

Performance: Event-driven processing can improve performance by enabling parallel processing of events.

Challenges:

Complexity: Implementing and managing an EDA can be complex, especially for large-scale systems. It requires careful planning, design, and operational management.

Data Consistency: Maintaining data consistency across multiple services can be challenging in an event-driven system. It requires careful handling of event ordering and potential conflicts.

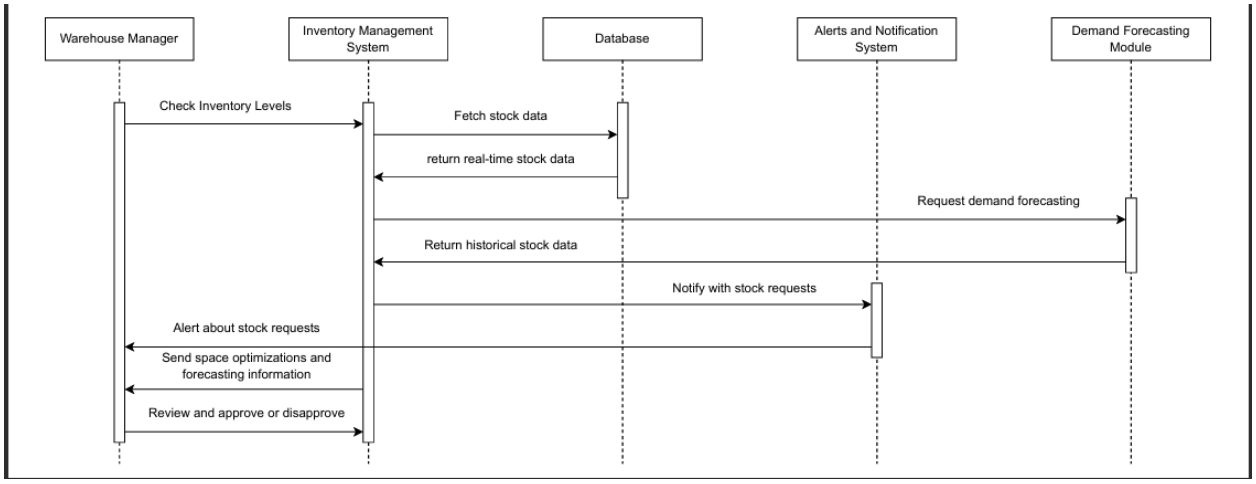
Conclusion:

Based on the analysis above between the the 2 architecture styles , Event driven architecture is the better fit since the primary concern regarding the

implementation was built upon real-time processing, event driven reactions and most importantly async communications (PUB/SUB system in the event bus and message queue)

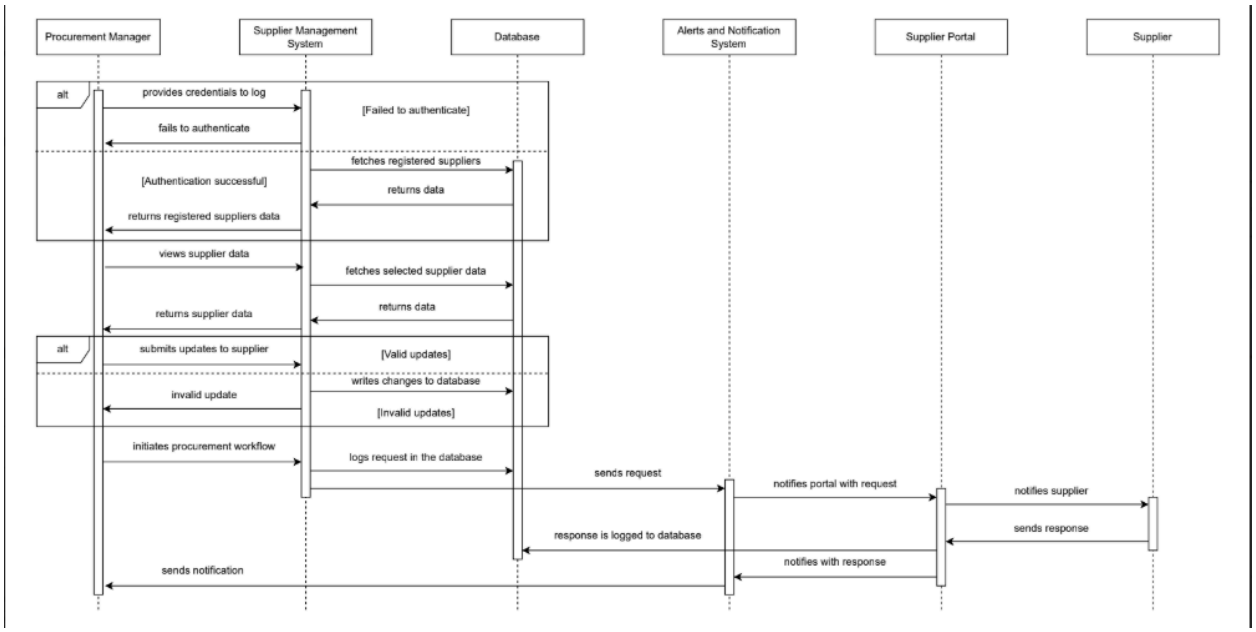
Sequence Diagrams:

1. Inventory Management



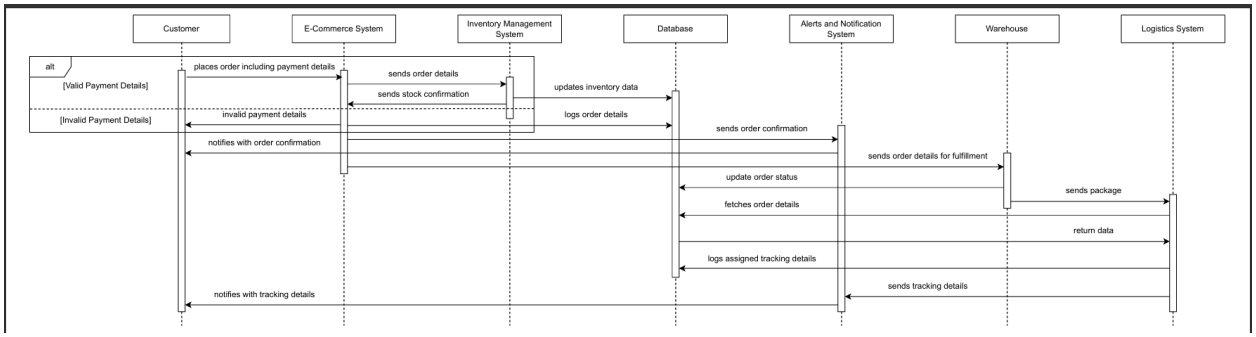
This sequence diagram satisfies the functional requirement related to inventory management, where inventory levels are being tracked , manage stock across warehouses and issue alters for low or full stock levels

2. Supplier Management



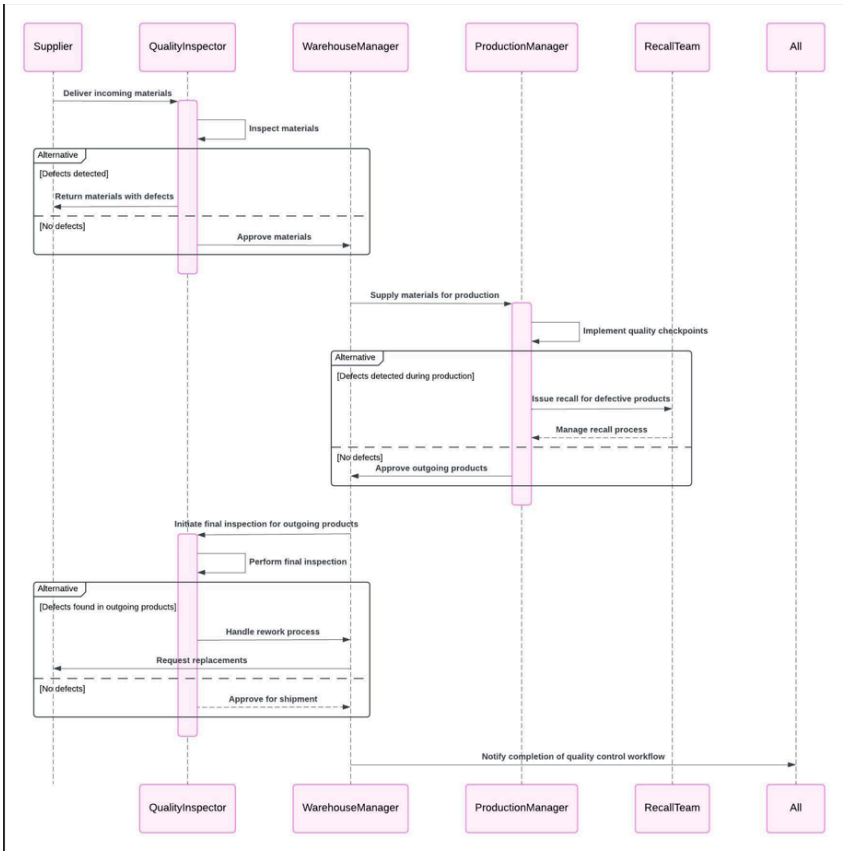
This sequence diagram satisfies the functional requirement related to supplier management, where supplier info is being managed and tracked for performance and delivery times

3. Order Processing



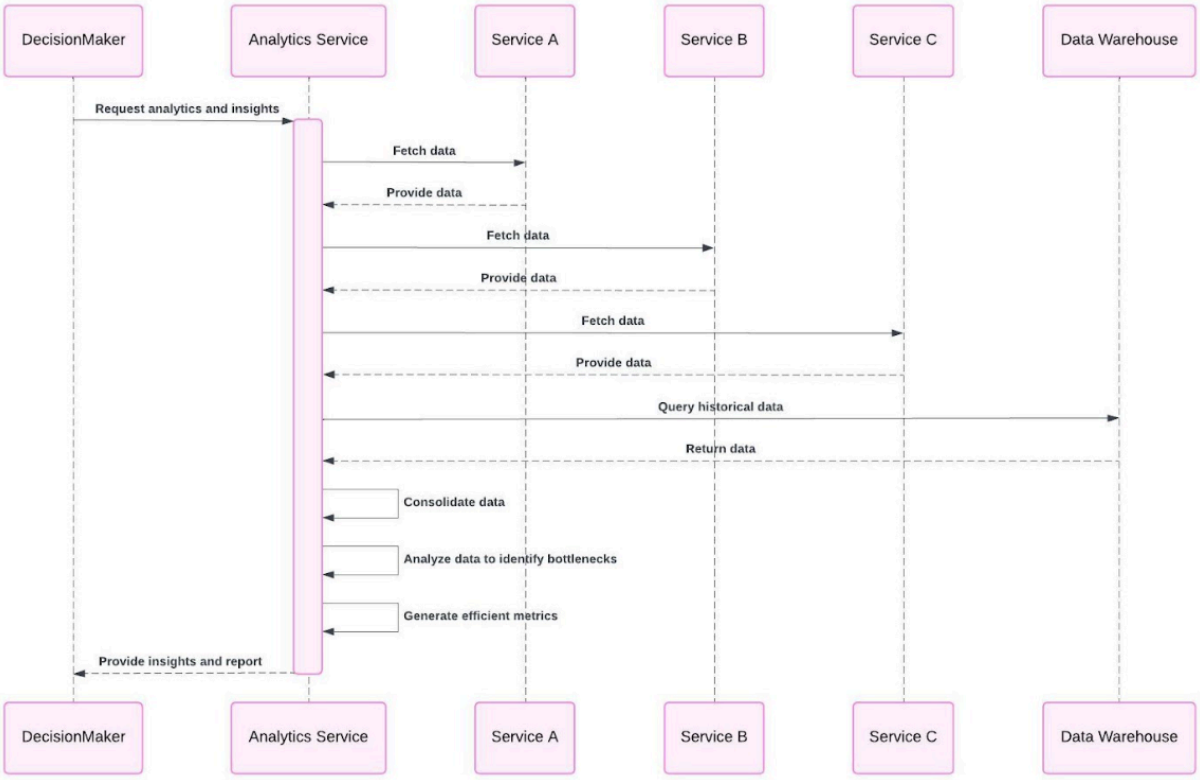
This sequence diagram satisfies the functional requirement related to order and shipping workflow , where customer orders are being handled from order placement through fulfilment , including order tracking and updates

4. Quality Control



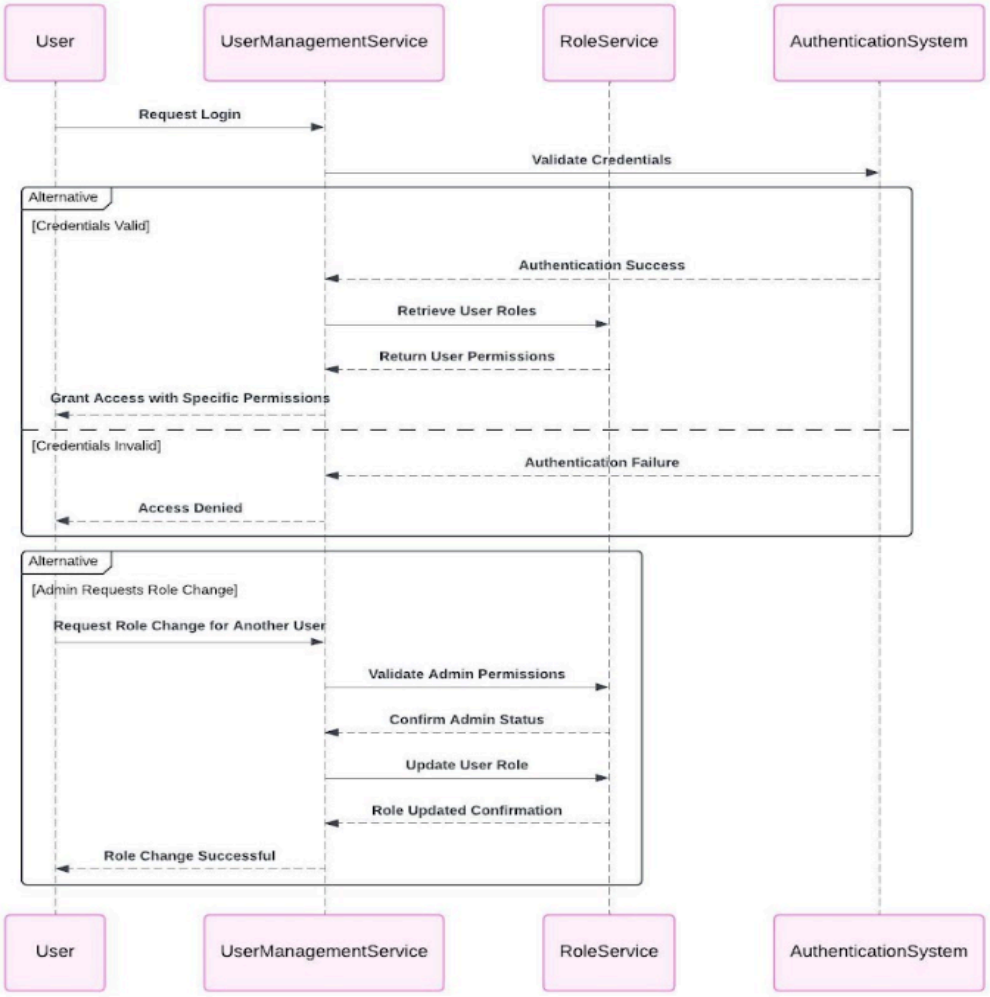
This sequence diagram satisfies the functional requirement related to quality control, where quality check points are being set up for incoming materials and outgoing products , this diagram also registers log defects , issues recalls if necessary and manages rework processes.

5. Analytics and insights



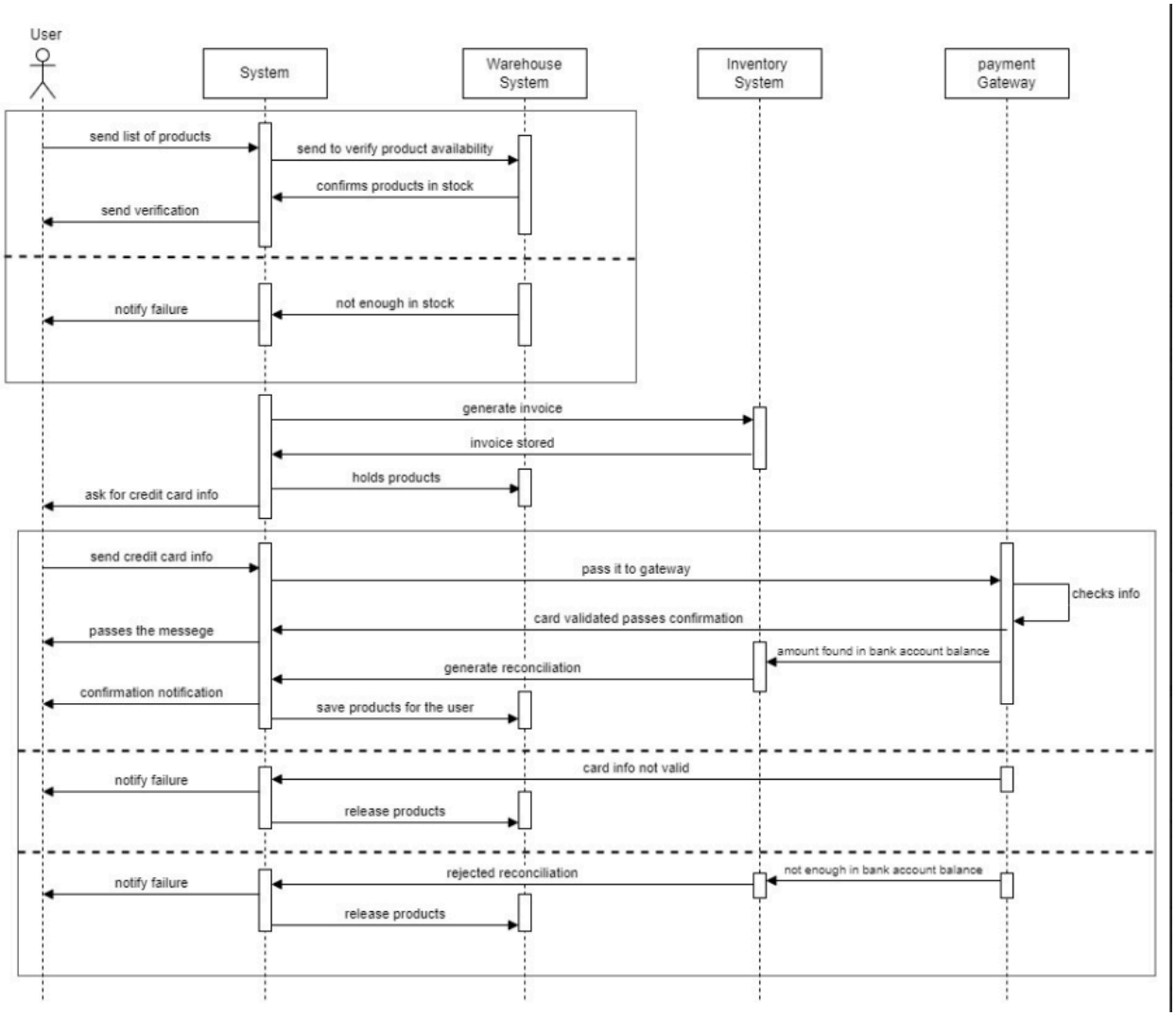
This sequence diagram satisfies the functional requirement related to analytics and insights. to support decision making , identify bottlenecks and efficient metrics for decision making.

6. User Access Control



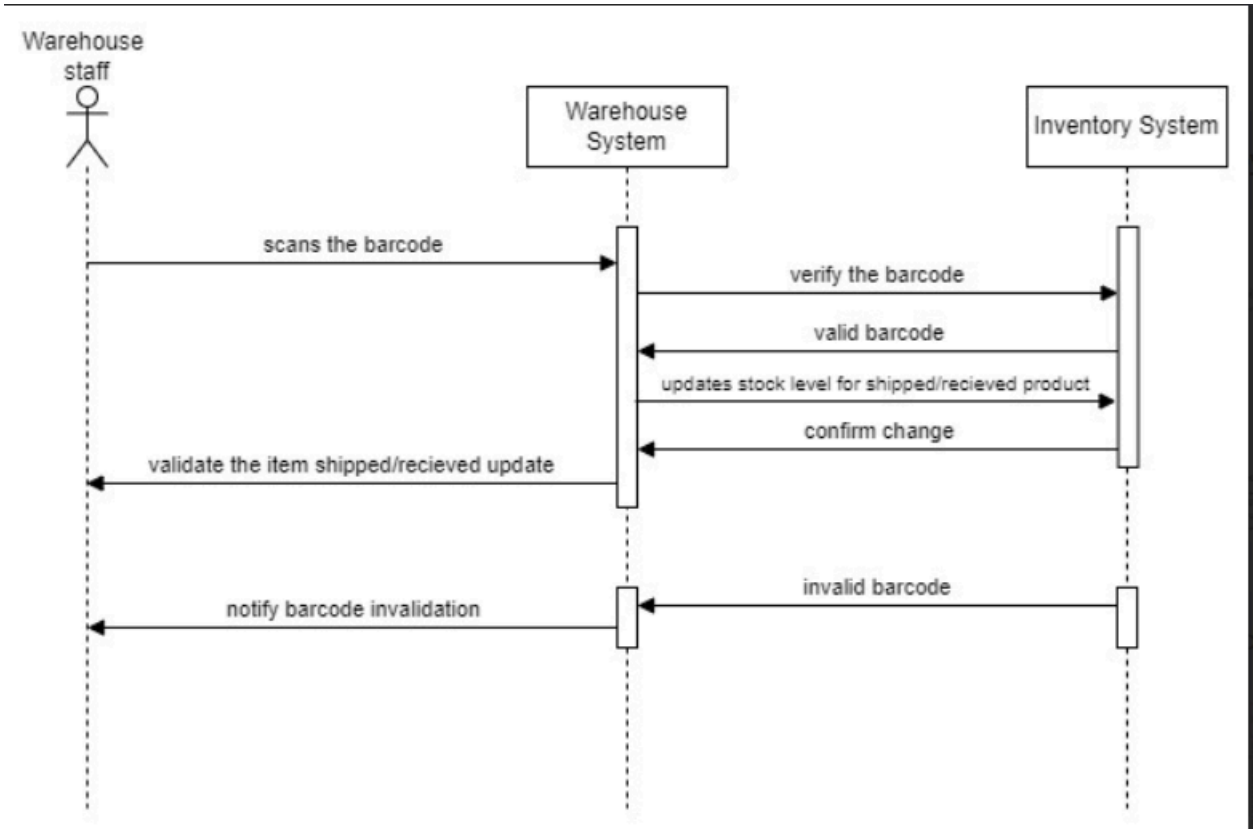
This sequence diagram satisfies the functional requirement related to user access control , where role based access is managed ensuring each user has acess only to relevant info and tasks

7. Financial Integration



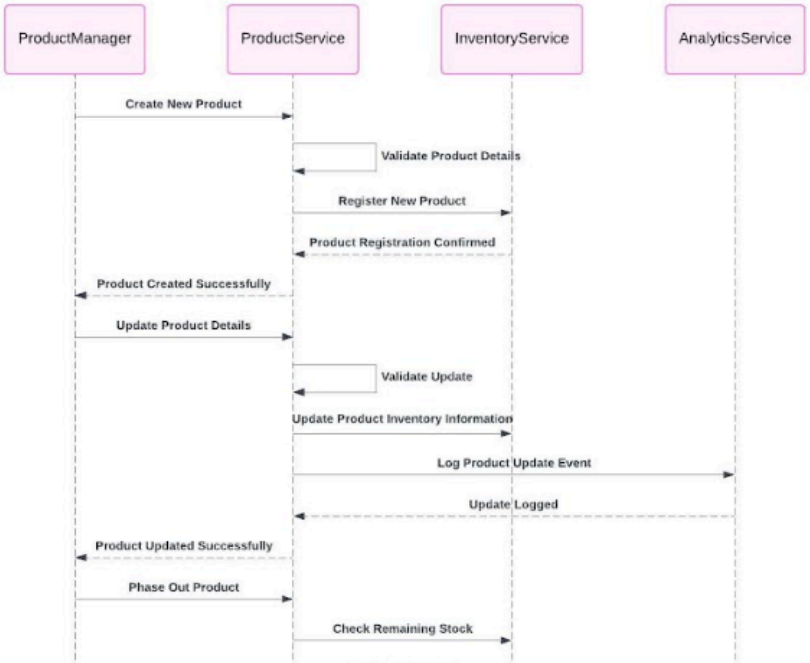
This sequence diagram satisfies the functional requirement related to financial integration, to handle transactions like invoicing, payments, and reconciliation related, but not restricted to accounting

8. Barcode Scanning



This sequence diagram satisfies the functional requirement related to barcode scanning, to streamline inventory checks , receiving and shipping process

9. Product Lifecycle Management



This sequence diagram satisfies the functional requirement related to Product Lifecycle Management, where product details are managed throughout its lifecycle, including creation, updates, and phase-out.

UML Class Diagram: