ASSINMENT 2

One-page draft scenario combining **Microservices Architecture**, **Event-Driven Architecture**, and **SOLID**, along with **DRY** and **KISS** principles for a hypothetical e-commerce platform

Scenario: E-Commerce Platform - "ShopSwift"

"ShopSwift" is a scalable, cloud-based e-commerce platform that sells electronics, apparel, and home essentials. The system is built using **Microservices Architecture**, ensuring each core function operates independently and communicates via lightweight protocols. It also utilizes **Event-Driven Architecture (EDA)** to handle asynchronous events and maintain loose

Microservices Architecture in Practice

- **User Service**: Handles user registration, authentication, and profile management.
- **Product Catalog Service**: Manages product listings, details, and search functionality.
- **Order Service**: Responsible for order placement, tracking, and history.
- **Payment Service**: Handles payments and invoices.
- **Inventory Service**: Updates product stock levels.
- **Notification Service**: Sends email/SMS notifications for events (order placed, shipped, etc.).

Event-Driven Architecture in Action

- When an order is placed, the **Order Service** publishes an **OrderPlaced** event.
- **Inventory Service** subscribes and updates stock levels.
- **Payment Service** processes the transaction upon receiving the event.
- **Notification Service** listens for multiple events like OrderPlaced, PaymentConfirmed, or OrderShipped and notifies the user accordingly.

This architecture ensures services remain decoupled and can scale independently while maintaining responsiveness.

Applying SOLID Principles

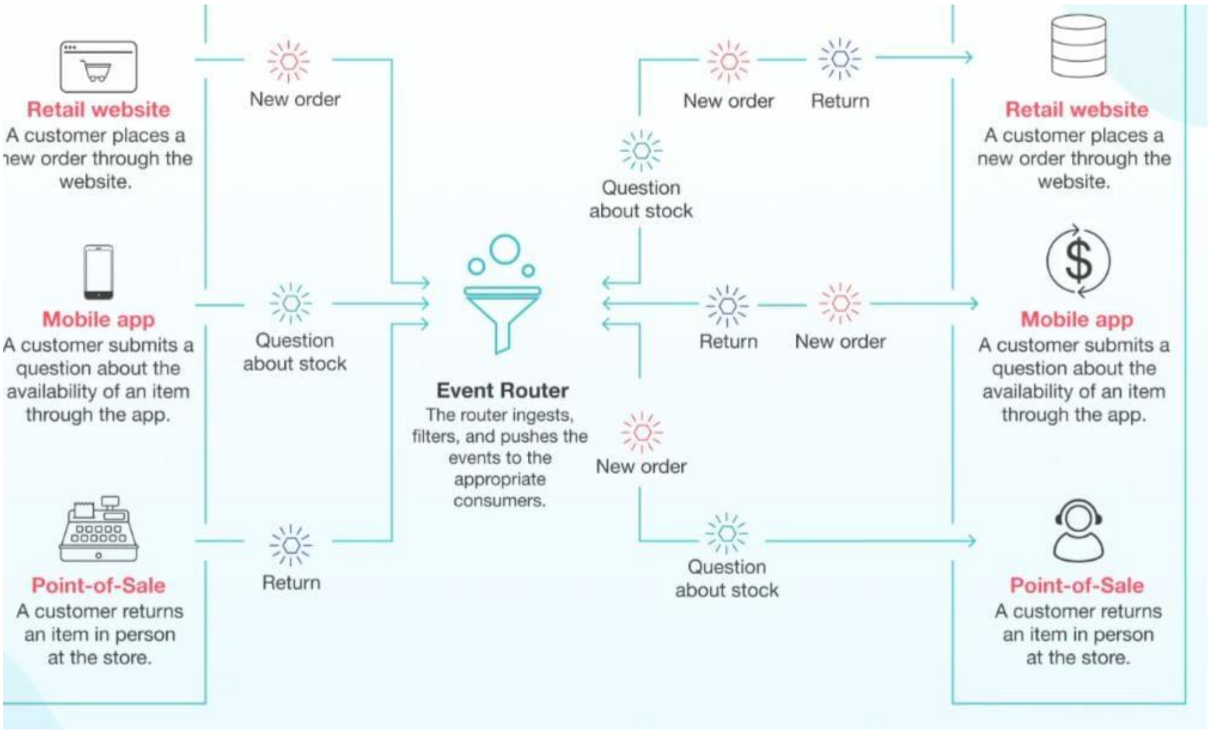
- **S - Single Responsibility**: Each service has one clearly defined responsibility (e.g., PaymentService only processes payments).

- **O - Open/Closed**: Services are open for extension via events (e.g., adding LoyaltyPointsService that listens to OrderCompleted) without modifying existing ones.
- **L - Liskov Substitution**: Interfaces like NotificationSender can be replaced with SMS or email implementations without altering core logic.
- **I - Interface Segregation**: Clients interact only with methods they need (e.g., PaymentService interface exposes pay() but not refund methods unless required).
- **D - Dependency Inversion**: Services depend on abstractions. For example, NotificationService depends on INotifier interface, allowing easy integration of new channels.

Observing DRY and KISS Principles

- **DRY (Don't Repeat Yourself)**:
    - Centralized logging service used by all microservices.
    - Reusable utility libraries for validation, error handling, and message formatting.
    - Shared API contracts (OpenAPI/Swagger) to avoid duplicate endpoint definitions.
- **KISS (Keep It Simple, Stupid)**:
    - Each service performs a small, manageable task.
    - Event messages follow a simple, consistent schema.
    - Minimal shared state—communication happens through well-defined events.

Fig 1.1 Microservices Design Principal(ref google)


Retail website
A customer places a
new order through the
website.
Mobile app
A customer submits a
question about the
availability of an item
through the app.
Point-of-Sale
A customer returns
an item in person
at the store.
New order
Question
about stock
Return
Event Router
The router ingests,
filters, and pushes the
events to the
appropriate
consumers.
New order
Return
Question
about stock
Return
New order
New order
Question
about stock
Retail website
A customer places a
new order through the
website.
Mobile app
A customer submits a
question about the
availability of an item
through the app.
Point-of-Sale
A customer returns
an item in person
at the store.