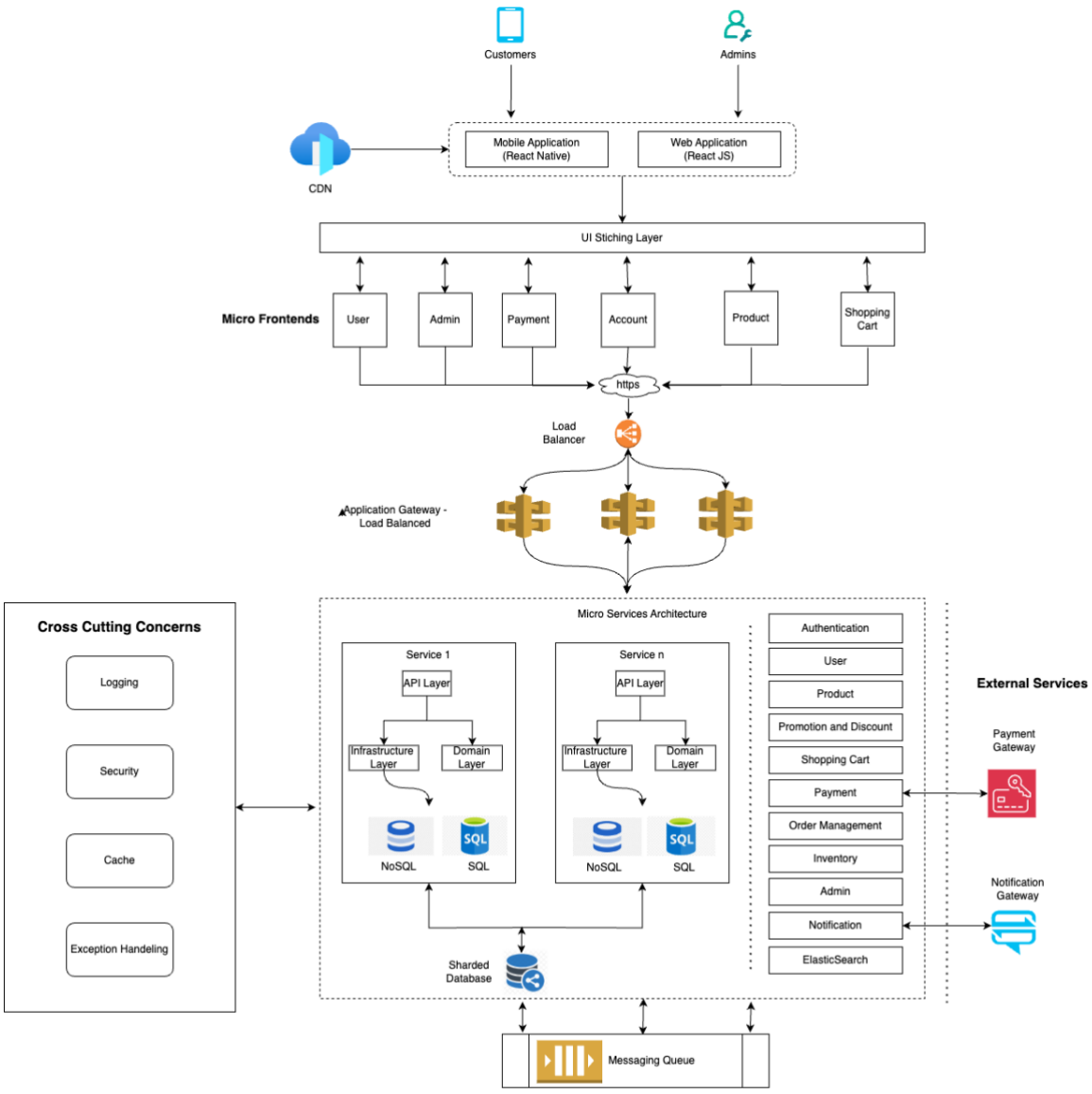
GLAM E-Commerce

Table of Contents

Architecture Diagram:



Architecture Diagram Components:

1. **Micro-Frontend Architecture:**

- The application's front end is structured using a micro-frontend architecture, allowing different sections (e.g., user, admin, product) to be developed and deployed independently.
- **ReactJS** is used for the web browser interface, while **React Native** supports mobile devices, ensuring a seamless and responsive experience across platforms.
- Each micro-frontend can be independently updated or scaled, which improves development agility and reduces deployment times for individual components.

2. **Load Balancer:**

- The **load balancer** serves as a **reverse proxy**, distributing incoming traffic across multiple servers to ensure **high availability** and **fault tolerance**.
- By spreading the load evenly, the balancer helps prevent server overload during peak traffic periods, optimizing resource utilization.
- It also provides **automatic failover**, ensuring uninterrupted service even if some servers go down.

3. **Application Gateway:**

- The **API gateway** acts as a single entry point for all communication between the client (front end) and the back-end microservices.
- It is responsible for **routing API requests** to the appropriate microservices and can also enforce security policies such as **authentication**, **rate limiting**, and **data transformation**.
- The gateway consolidates multiple API endpoints into a single **external endpoint**, simplifying external communication and enhancing security.

4. **Cross-Cutting Concerns:**

- Common concerns like **Logging**, **Security**, **Caching**, and **Exception Handling** are implemented as **cross-cutting services**.
- These services are **reused across all microservices**, ensuring consistency and reducing the need for redundant implementation.
- **Logging** captures application activity for audit trails, **Security** manages authentication/authorization, **Caching** improves performance by storing frequently used data, and **Exception Handling** standardizes error responses across the application.

5. **Messaging Queue:**

- A **messaging queue** (e.g., RabbitMQ, Kafka) is used for **asynchronous communication** between microservices.
- It decouples services, enabling them to communicate without waiting for real-time responses, improving system **resilience** and **fault tolerance**.
- The messaging queue is also used to handle external communications (e.g., sending notifications or processing orders) without blocking the main application flow.

6. **Content Delivery Network (CDN):**

- The **CDN** serves static assets like **JavaScript**, **CSS**, and **image files** from geographically distributed edge servers, reducing **latency** and improving **load times** for end-users.
- By caching static content on edge servers closer to the users, the CDN offloads this traffic from the main server, enhancing the overall performance and scalability of the platform.
- The CDN also accelerates **file delivery** and minimizes bandwidth usage by caching frequently accessed data.

7. **File System:**

- The **file system** is a hybrid setup that uses **AWS S3** for cloud-based storage of non-sensitive files, such as user uploads, product images, and general media.
- For **sensitive documents**, the system incorporates **on-premise storage** to comply with regulatory and security requirements.

- This ensures that all documents are easily accessible, scalable, and backed up, while still adhering to strict security policies for sensitive data.

8. **Sharded Database (SQL + NoSQL):**
   - **SQL** shards handle structured data (e.g., orders, customers) for transactional integrity, while **NoSQL** shards manage unstructured data (e.g., product catalogs, reviews).
   - Sharding distributes data across multiple instances, improving scalability and performance.
   - **SQL** shards integrate with **Order Management** and **Inventory** microservices for accurate transactions and stock control.
   - **NoSQL** shards integrate with **Product** and **Personalization** microservices for dynamic content and real-time updates.
   - This hybrid approach balances **ACID compliance** (SQL) with **scalability** and **flexibility** (NoSQL).

9. **Payment Gateway:**
   - The platform integrates with an **external payment gateway** (e.g., Stripe, PayPal) to handle transactions securely.

10. **Notification Gateway:**
    - An **external notification gateway** (e.g., Twilio, SendGrid) is used to handle all **outgoing notifications**, including emails, SMS, and push notifications.

Services Identified:

1. **Authentication Microservice:**
   - **Handles user login, logout, and authentication** using secure methods (e.g., OAuth2, JWT).
   - Responsible for **generating tokens** upon successful authentication, which other services can validate.
   - Ensures secure password management, including password hashing, encryption, and handling user identity verification.

2. **User Microservice:**
   - Manages all **user-related operations**, such as **user registration**, profile management, and account updates.
   - Stores and retrieves user details, including **preferences, order history**, and account activity.
   - Provides APIs for accessing **user data** securely, ensuring privacy and compliance with data regulations.

3. **Product Microservice:**
   - Manages the **catalog of products**, including product descriptions, prices, images, and availability.
   - Handles **product listing, searching, filtering**, and categorization based on attributes like color, brand, price, and ratings.
   - Provides APIs for front-end components to retrieve product data and display it to users.

4. **Promotion and Discount Microservice:**
   - Manages **promotional campaigns**, discounts, and offers.
   - Applies **business logic** for calculating discounted prices based on predefined rules (e.g., percentage discounts, seasonal promotions).
   - Integrates with the shopping cart and order processing services to apply promotions during checkout.

5. **Shopping Cart Microservice:**
   - Handles the logic for **adding, updating, and removing items** from the user's shopping cart.
   - Supports **session-based or account-based cart persistence**, ensuring the cart remains intact even when users log out or switch devices.
   - Integrates with the **Product** and **Inventory** microservices to ensure real-time availability of items in the cart.

6. **Payment Microservice:**
   - Manages the integration with **payment gateways** like PayPal, Stripe, or Razorpay to process transactions securely.
   - Handles **payment methods**, such as credit cards, bank transfers, or digital wallets, ensuring PCI DSS compliance.
   - Provides APIs to manage transaction history and refunds, and integrate with fraud detection systems.

7. **Order Management Microservice:**
   - Manages the **lifecycle of orders**, from creation and processing to confirmation and fulfillment.
   - Coordinates with the **inventory**, **payment**, and **notification** microservices to confirm stock availability, process payments, and send order confirmations.
   - Tracks the status of orders (e.g., pending, shipped, delivered) and integrates with fulfillment and logistics systems.

8. **Inventory Management Microservice:**
   - Manages real-time tracking of **product inventory levels**.
   - Updates inventory counts based on order confirmations and product returns, ensuring that only available stock can be sold.
   - Integrates with the **Order Management** and **Product** microservices to ensure accurate stock levels and prevent overselling.

9. **Admin Microservice:**
   - Handles all **notifications** sent to users, including order confirmations, promotional emails, and alerts.
   - Supports **multiple channels**, such as email, SMS, and in-app notifications, ensuring users receive timely updates.
   - Manages notification templates and ensures consistency across different platforms.

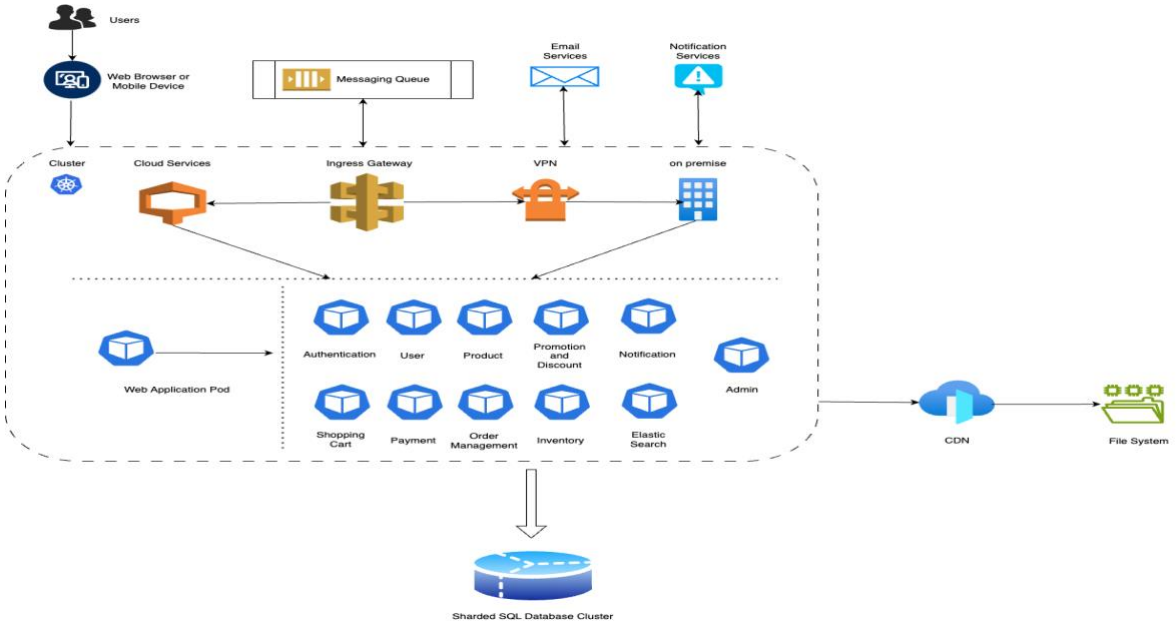
10. **Notification Microservice:**
   - Handles all **notifications** sent to users, including order confirmations, promotional emails, and alerts.
   - Supports **multiple channels**, such as email, SMS, and in-app notifications, ensuring users receive timely updates.
   - Manages notification templates and ensures consistency across different platforms.

11. **Elastic Search Microservice:**
   - Powers **full-text search capabilities** across the platform, allowing users to quickly search for products using keywords or filters.
   - Provides near real-time indexing of product data to ensure that search results are always up to date.
   - Supports advanced search features like **fuzzy matching**, auto-suggestions, and filtering based on user preferences.

Tech Stack for the application

1. Frontend: React Native and React JS with micro frontend:
2. Language: Java, TypeScript
3. Framework: Java Microservices with Spring boot
4. Database: Oracle
5. File Storage: Aws S3
6. Messaging Queue: Azure Queue
7. Code Repo : Azure/GIT
8. CI/CD : Azure pipelines
9. Code Review: Gerrit

Deployment Diagram:



Deployment Diagram Components:

1. **Ingress Gateway:**
   - The Ingress Gateway serves as the entry point into the service mesh and acts as the load balancer.
   - It manages all incoming requests, balancing traffic between cloud services and on-premise servers based on load conditions.
   - The gateway is responsible for service discovery, traffic routing, and ensuring security policies like SSL termination.
   - The gateway auto-scales by creating and destroying pods dynamically using Kubernetes, ensuring optimal resource allocation during traffic spikes and downtimes. This helps maintain high availability and resilience of the platform.

2. **VPN (Virtual Private Network):**
   - The **VPN** ensures secure routing of traffic between the **ingress gateway** and **on-premise servers**.
   - It encrypts data as it passes through external networks, ensuring that sensitive data (e.g., transactions, user information) remains secure when traveling between cloud and on-premise infrastructure.

3. **Servers:**
   - The deployment uses a combination of **on-premise** and **cloud servers** to handle application workloads.
   - **Cloud servers** are flexible, elastic resources that scale automatically to meet the demand. **On-premise servers** handle specific tasks, such as handling sensitive data or meeting legal/regulatory requirements where cloud usage is restricted.
   - This hybrid approach ensures that the application benefits from the flexibility of the cloud while leveraging the control and security of on-premise infrastructure.

4. **Web Application Pods:**
   - The web application is deployed in a **Kubernetes-managed environment**, where **pods** are dynamically created and destroyed by the Ingress Gateway based on traffic patterns.
   - The **pods** host microservices (authentication, user management, product catalog, etc.) and allow the platform to **auto-scale**, ensuring the application remains responsive even during peak load conditions.

- The **Kubernetes orchestrator** ensures high availability by distributing the pods across nodes and continuously monitoring the health of the services.

5. **Messaging Queue:**

- The **Messaging Queue** (e.g., RabbitMQ, Kafka) decouples microservices, allowing asynchronous communication between different components like **email services**, **notification services**, and the core application.
- It ensures reliable delivery of messages and maintains system integrity, even during periods of heavy load.

6. **CDN (Content Delivery Network):**

- The **CDN** is responsible for serving static assets like images, CSS, and JavaScript files. It helps reduce the load on the origin servers by caching these assets and serving them from edge locations closer to users.
- The CDN also accelerates content delivery, minimizing latency and improving user experience globally.