Software Design Specifications

for

Shopping Aggregator

Version 2.0

Prepared by:

BudBots

Document Information

Title: Shopping Aggregator SDS	
Project Manager:	**Document Version No:**
	Document Version Date:
Prepared By: Budbots	**Preparation Date:**

Version History

Ver. No.	Ver. Date	Revised by	Description	Filename
1.0	April 5, 2025	Budbots	Initial draft of SDD	
2.0	April 8, 2025	Budbots	Final draft of SDD	

Table of Contents

1. Introduction

This section provides an overview of the **Shopping Aggregator** software design, including its purpose, scope, key terminology, and references. It serves as a foundation for understanding the technical architecture and design decisions outlined in this document

1.1 Purpose

The **Software Design Specifications (SDS)** document translates the requirements defined in the **Software Requirements Specification (SRS)** into a detailed technical blueprint for the Shopping Aggregator platform. It:

- Describes the system architecture, components, and workflows.
- Guides developers in implementing features like real-time data aggregation, AI-driven product matching, and user authentication.
- Ensures alignment with non-functional requirements (e.g., security, performance, scalability).

1.2 Scope

This SDS applies to the following aspects of the Shopping Aggregator:

1. **System Architecture**:
    - Frontend (React.js), backend (Node.js/Express.js), and databases (MongoDB for product data).
    - **Authentication**: Google Sign-In (OAuth 2.0) for user login, eliminating the need for local credential storage.
2. **Core Features**:
    - Real-time product data aggregation from e-commerce APIs (e.g., Amazon, Flipkart).
    - AI-driven product matching and personalized recommendations.
    - **User Session Management**: Token-based sessions using Google OAuth tokens (JWT).
3. **Data Flow**:
    - Product data stored in MongoDB (price history, descriptions).
    - User profile data (name, email) retrieved securely from Google APIs.

Out of Scope:

- **Local User Credential Storage**: No database for passwords or user credentials.
- Payment processing or checkout workflows.

1.3 Definitions, Acronyms, and Abbreviations

[This subsection should provide the definitions of all terms, acronyms, and abbreviations required to properly interpret the **Software Design Specifications**. This information may be provided by reference to the project Glossary.]

1.4 References

1. **Software Requirements Specification (SRS) for Shopping Aggregator** (BudBots, Version 1.0, March 2025) – Internal project document.
2. **IEEE Std 830-1998** – Guidelines for software requirements specifications.
3. **GDPR Compliance Guidelines**
4. **PCI DSS Standard** – Security requirements for payment processing.
5. **COMET Methodology** – Software Modeling and Design by Hassan Gomaa (2011).

2. Use Case View

This section maps **all SRS use cases** to the system design, including those deferred to future phases.

2.1 Use Case

U1: Login/Authentication (Revised for Google Sign-In)
Actors: User, Google Auth Provider
Purpose: Replace local authentication with Google OAuth 2.0.
Priority: High
Design Flow:

1. User selects "Sign in with Google."
2. System redirects to Google's OAuth consent screen.
3. Google returns an id_token (JWT) to the backend.
4. Backend validates the token using Google's public keys.
5. User session is created with a JWT (no local credentials stored).

Components:

- Frontend: AuthButton (React.js)
- Backend: OAuthController (Node.js), GoogleAuthService

U2: Search Products
Actors: User, E-commerce APIs
Purpose: Fetch real-time product data from integrated APIs.
Design Flow:

1. User enters a search query (e.g., "smartwatch").
2. SearchService checks Redis cache → Cache miss triggers API calls.
3. Data is normalized, ranked via AI, and returned to UI.

Components:

- SearchController, ProductAggregatorService, RedisCache

U3: Filter and Compare Products
Actors: User
Purpose: Refine search results and enable side-by-side comparisons.
Design Flow:

1. User applies filters (price range, brand).
2. Frontend (React.js) dynamically updates results.
3. Comparison data is fetched from MongoDB.

Components:

- FilterComponent, ProductComparator

U4: Checkout and Pay (Future Phase)
Actors: User, Payment Gateway
Purpose: Process payments (deferred to future implementation).
Design Notes:

- Placeholder for Stripe/PayPal integration.
- No current database schema for transactions.

U5: Track Order (Future Phase)
Actors: User
Purpose: Monitor order status (requires payment integration).
Design Notes:

- Will rely on external shipping APIs (not implemented).

U6: Manage Product Listings (Administrator)
Actors: Administrator
Purpose: Add/update product data manually (fallback if API fails).
Design Flow:

1. Admin logs in via Google Sign-In (role: "admin").
2. Accesses admin dashboard to edit MongoDB Products collection.

Components:

- AdminController, ProductEditorService

U7: Provide Product Data (Retailer – Future Phase)
Actors: Retailer
Purpose: Allow retailers to submit product data (deferred).
Design Notes:

- Will require a retailer portal and data validation pipeline.

U8: Process Payments (Future Phase)
Actors: Payment Gateway
Purpose: Handle transactions (requires PCI DSS compliance).
Design Notes:

- Placeholder for Stripe API integration.

U9: Monitor System Performance
Actors: Administrator
Purpose: Track system health and API performance.
Design Flow:

1. Admin views Grafana dashboards (CPU, memory, API latency).
2. Alerts are routed to Slack via Prometheus.

Components:

- MonitoringService, GrafanaDashboard

U10: Manage Users
Actors: Administrator
Purpose: Assign roles (e.g., admin) via Google email.
Design Flow:

1. Admin adds a Google email to the Admins collection in MongoDB.
2. Backend validates admin role during login via Google token.

Components:

- AdminService, RoleMiddleware

3. Design Overview

This section provides a comprehensive overview of the architectural design, objectives, and constraints governing the Shopping Aggregator platform. It aligns with the functional and non-functional requirements outlined in the SRS while emphasizing scalability, security, and modularity.

3.1 Design Goals and Constraints

Design Goals:

1. **Scalability**:
    - Support horizontal scaling to accommodate 10,000+ concurrent users.
    - Utilize AWS Elastic Load Balancing and auto-scaling groups for dynamic resource allocation.
2. **Real-Time Responsiveness**:
    - Ensure product data refreshes every 15 minutes via vendor API integrations.
    - Implement Redis caching to reduce latency for frequent search queries.
3. **Security**:
    - Eliminate local credential storage by delegating authentication to Google OAuth 2.0.
    - Encrypt sensitive data (e.g., user preferences) using AES-256 and enforce HTTPS/TLS 1.3.
4. **Modularity**:
    - Decouple components (e.g., authentication, search engine) to facilitate independent updates.
5. **Compliance**:
    - Adhere to GDPR for user data privacy and Google API Services User Data Policy.

Design Constraints:

1. **Technology Stack**:
   - **Frontend**: React.js with Redux for state management.
   - **Backend**: Node.js/Express.js RESTful APIs.
   - **Databases**: MongoDB (product data), Redis (caching).
   - **Authentication**: Google Sign-In (OAuth 2.0) with JWT.
2. **Third-Party Dependencies**:
   - Reliance on external e-commerce APIs (Amazon, Flipkart) for real-time product data.
3. **Budgetary Limitations**:
   - Initial deployment restricted to AWS Free Tier resources (EC2, S3).

3.2 Design Assumptions

1. **Third-Party Service Reliability**:
   - Google OAuth and e-commerce APIs will maintain ≥99.9% uptime.
2. **Data Consistency**:
   - Vendor APIs return structured, normalized product data (JSON/XML).
3. **User Behavior**:
   - 80% of users will access the platform via mobile devices (prioritize responsive design).
4. **Network Stability**:
   - Sufficient bandwidth for real-time API calls and data aggregation.
5. **Regulatory Compliance**:
   - No changes to GDPR or PCI DSS during the development lifecycle.

3.3 Significant Design Packages

The system is decomposed into **four architectural layers** to promote modularity and separation of concerns:

Layer	**Components**	**Responsibilities**
Presentation	React.js UI, Redux State Management	Render user interfaces, handle client-side interactions.
Application	Express.js APIs, Middleware	Process business logic, route requests, manage sessions.
Data	MongoDB, Redis	Store product data, user preferences, and cached results.
Integration	Google OAuth, E-commerce API Clients	Authenticate users, fetch external product data.

Package Dependencies:

- The **Presentation Layer** depends on the **Application Layer** for data via REST APIs.
- The **Application Layer** interacts with the **Data Layer** for persistence and caching. The **Integration Layer** is consumed by the **Application Layer** for authentication and data aggregation.

3.4 Dependent External Interfaces

The table below lists the public interfaces this design requires from other modules or applications.

External Application and Interface Name	**Module Using the Interface**	**Functionality/Description**
Google OAuth 2.0 (https://oauth2.googleapis.com)	AuthService	Validates user identity via OAuth 2.0 protocol and retrieves user profile data (email, name).
Amazon Product API (api.amazon.com/products)	ProductAggregator	Fetches real-time product prices, availability, and descriptions from Amazon.

Flipkart Product API (api.flipkart.com/listings)	ProductAggregator	Retrieves product listings, ratings, and seller details from Flipkart.
Redis Cache (redis://cache)	CacheManager	Stores temporary search results and session data to reduce latency and API calls.
AWS S3 (s3.amazonaws.com/images)	MediaService	Hosts product images as a fallback when vendor image URLs are unavailable.

3.5 Implemented Application External Interfaces (and SOA web services)

The table below lists the implementation of public interfaces this design makes available for other applications.

Interface Name	**Module Implementing the Interface**	**Functionality/Description**
Product Search API (/api/v1/search)	SearchController	Accepts search queries and returns aggregated product results with filters and sorting.
User Preferences API (/api/v1/preferences)	UserPreferencesService	Manages user-specific settings (e.g., deal alerts, favorite categories) linked to Google IDs.
Price Tracking Webhook (/webhook/price-update)	PriceTracker	Receives real-time price updates from vendor APIs and triggers user notifications.
Admin Dashboard API (/api/v1/admin/metrics)	AdminController	Provides system performance metrics (uptime, API latency) for administrative monitoring.

4. Logical View of Shopping Aggregator Software

This section presents the logical decomposition of the Shopping Aggregator system, illustrating how modules and classes interact to fulfil key use cases. The logical view is structured in layers, starting from high-level module interactions down to class-level collaborations.

1. Top-Level Layer: Module Interactions

1.1 Key Modules & Their Responsibilities

Module	Primary Responsibility
User Interface (UI)	Renders web/mobile interfaces, captures user input, and displays aggregated product data
API Gateway	Routes requests to appropriate microservices, handles authentication, and load balancing.
Search & Aggregation	Fetches, normalizes, ranks, and deduplicates product listings from multiple vendors.
Vendor Integration	Manages real-time API connections with external e-commerce platforms (Amazon, eBay, etc.).
User Management	Handles user authentication, profiles, preferences, and search history.
Analytics	Tracks user behaviour, click-through rates, and optimizes recommendations.
Database	Stores product metadata, user data, and transaction logs.

1.2 Interaction Flow for Key Use Cases

Use Case 1: Product Search & Comparison

1. UI sends a search request to API Gateway.
2. API Gateway routes the request to the Search & Aggregation Module.
3. Search Module queries Vendor Integration for real-time product data.
4. Vendor Integration fetches data from external APIs (Amazon, eBay, etc.).
5. Search Module normalizes, ranks, and returns results to the UI.

Use Case 2: Purchase Redirection
1. UI sends a purchase request to API Gateway.
2. API Gateway forwards it to Vendor Integration
3. Vendor Integration generates a vendor-specific checkout URL.
4. Analytics Module logs the transaction for future recommendations.
5. User is redirected to the vendor’s checkout page.

2. Mid-Level Layer: Module Decomposition

2.1 Search & Aggregation Module

Responsibilities:
- Fetch and merge product listings from multiple vendors.
- Apply ranking algorithms (price, relevance, ratings).
- Deduplicate similar products.

Key Classes:

Class	Role
Product Search Service	Orchestrates search flow, calls vendor adapters, and applies ranking.
Product Aggregator	Merges listings, normalizes data, and removes duplicates.
Ranking Strategy	Implements sorting logic (e.g., "Best Price First", "Top Rated").

Collaboration:
1. ‘Product Search Service’ → Calls ‘Vendor Integration’ → Fetches raw product data.
2. ‘Product Aggregator’ → Normalizes and deduplicates listings.
3. ‘Ranking Strategy’ → Sorts results before sending to UI.

2.2 Vendor Integration Module
Responsibilities:
- Manage API connections with external vendors.
- Handle authentication, rate limiting, and error recovery.
- Transform vendor-specific responses into a unified schema.

Key Classes:

Class	Role
Vendor API Factory	Instantiates the correct vendor adapter (Amazon, eBay, etc.).
Amazon Adapter	Handles Amazon API calls and response parsing.
API Rate Limiter	Ensures compliance with vendor API quotas.
Vendor Response Parser	Converts vendor-specific JSON/XML into a standard ‘Product’ object.

Collaboration:
1. ‘Vendor API Factory’ → Creates ‘Amazon Adapter’ or ‘Ebay Adapter’ based on request.
2. ‘API Rate Limiter’ → Throttles API calls to avoid rate limits.
3. ‘Vendor Response Parser’ → Standardizes product data before returning to Search Module.

3. Low-Level Layer: Class Method Details

3.1 ‘Product Search Service’ Class
Methods:
1. Search (query: String, filters: Map<String, String>): List<Product>
   Calls Vendor API Factory.getAdapters() to fetch data from all vendors.
   Passes results to ProductAggregator.merge()
   Applies RankingStrategy.sort() before returning.

2. apply User Preferences (products: List<Product>, userId: String): List<Product>
   Fetches user preferences via   UserManagement.getPreferences(userId)
   Re-orders products based on user’s past behaviour.

Pseudo-Code:
python
```
def search (query, filters):
    vendor adapters = VendorAPIFactory.getAdapters()
    all products = []
    for adapter in vendor adapters:
        products = adapter.fetchProducts(query, filters)
        all_products.append(products)
    merged_products = ProductAggregator.merge(all_products)
    ranked_products = RankingStrategy.sort(merged_products)
    return UserManagement.applyPreferences(ranked_products, current_user)
```

3.2 ‘Amazon Adapter’ Class
Methods:
1. fetch Products (query: String): List<Vendor Product>
   - Sends an authenticated request to Amazon Product API.
   - Uses ‘API Rate Limiter’ to avoid exceeding quota.
   - Parses response with ‘Vendor Response Parser’.

Pseudo-Code:
python
```
def fetchProducts(query):
    if not API Rate Limiter.canMakeRequest("Amazon"):
        raise RateLimitExceededError()
    response = AmazonAPI.get("/search?q=" + query)
    parsed_products = VendorResponseParser.parse(response)
    return parsed_products
```

4. Data Flow Diagram

```
User → UI → API Gateway → Search Module → Vendor Integration → External APIs
                          ↓
                       Product Aggregation → Ranking → UI
                          ↓
                       Analytics & Database Logging
```

5. Key Design Patterns Used

- Facade Pattern (‘API Gateway’ simplifies microservice access).
- Strategy Pattern (‘Ranking Strategy’ allows dynamic sorting rules).
- Adapter Pattern (‘Amazon Adapter’, ‘Ebay Adapter’ standardize vendor APIs).
- Factory Pattern (‘Vendor API Factory’ creates appropriate adapters).

This logical decomposition ensures modularity, scalability, and maintainability while fulfilling all key use cases efficiently. Further refinements can be made for caching, real-time updates, and AI-driven recommendations.

4.1 Design Model

This section provides a **class-centric** design decomposition of the Shopping Aggregator system, detailing key modules, classes, their responsibilities, relationships, and interactions.

1. High-Level Module Breakdown

The system is divided into the following key modules:

1. User Interface (UI) Module
2. API Gateway Module
3. Search & Aggregation Module
4. Vendor Integration Module
5. User Management Module
6. Analytics Module
7. Database Module

2. Class Diagrams & Descriptions

2.1 Search & Aggregation Module

Purpose: Fetches, normalizes, and ranks products from multiple vendors.

Class Diagram

```
mermaid
Copy
classDiagram
  class ProductSearchService {
    +search(query: String, filters: Map~String, String~): List~Product~
    +applyUserPreferences(products: List~Product~, userId: String): List~Product~
  }

  class ProductAggregator {
    +merge(products: List~List~Product~~): List~Product~
    +normalize(product: VendorProduct): Product
  }

  class RankingStrategy {
    +sort(products: List~Product~, strategy: String): List~Product~
  }

  ProductSearchService --> ProductAggregator
  ProductSearchService --> RankingStrategy
```

Key Classes

Class	Responsibilities	Key Attributes/Methods
`ProductSearchService`	Coordinates search flow, calls vendor APIs, and applies ranking.	`+search()`, `+applyUserPreferences()`
`ProductAggregator`	Merges and normalizes product listings from different vendors.	`+merge()`, `+normalize()`
`RankingStrategy`	Implements different sorting algorithms (e.g., price, rating, relevance).	`+sort()`

2.2 Vendor Integration Module

Purpose: Manages real-time API connections with external e-commerce platforms.

Class Diagram

mermaid
Copy

```
classDiagram
    class VendorAPIFactory {
        +getAdapter(vendor: String): VendorAPIAdapter
    }

    class VendorAPIAdapter {
        <<abstract>>
        +fetchProducts(query: String): List~VendorProduct~
    }

    class AmazonAdapter {
        +fetchProducts(query: String): List~VendorProduct~
    }

    class EbayAdapter {
        +fetchProducts(query: String): List~VendorProduct~
    }

    class APIRateLimiter {
        +canMakeRequest(vendor: String): Boolean
    }

    VendorAPIFactory --> VendorAPIAdapter
    VendorAPIAdapter <|-- AmazonAdapter
    VendorAPIAdapter <|-- EbayAdapter
    AmazonAdapter --> APIRateLimiter
    EbayAdapter --> APIRateLimiter
```

Key Classes

Class	Responsibilities	Key Attributes/Methods
`VendorAPIFactory`	Creates the appropriate vendor adapter (Amazon, eBay, etc.).	`+getAdapter()`

Class	Responsibilities	Key Attributes/Methods
`VendorAPIAdapter`	Abstract class defining the interface for vendor API interactions.	`+fetchProducts()`
`AmazonAdapter`	Handles Amazon Product Advertising API calls.	`+fetchProducts()`
`EbayAdapter`	Handles eBay Finding API calls.	`+fetchProducts()`
`APIRateLimiter`	Ensures API calls stay within vendor rate limits.	`+canMakeRequest()`

2.3 User Management Module

Purpose: Manages user accounts, preferences, and search history.

Class Diagram

mermaid
Copy

```
classDiagram
    class UserService {
        +login(email: String, password: String): User
        +register(user: User): Boolean
        +getPreferences(userId: String): Map~String, String~
    }

    class User {
        -id: String
        -name: String
        -email: String
        -preferences: Map~String, String~
    }

    UserService --> User
```

Key Classes

Class	Responsibilities	Key Attributes/Methods
`UserService`	Handles authentication and preference management.	`+login()`, `+getPreferences()`
`User`	Stores user profile data.	`id`, `name`, `preferences`

2.4 Analytics Module

Purpose: Tracks user behavior and optimizes recommendations.

Class Diagram

mermaid
Copy

```
classDiagram
  class AnalyticsService {
    +logSearch(query: String, userId: String)
    +logPurchase(productId: String, userId: String)
    +getRecommendations(userId: String): List~Product~
  }

  AnalyticsService --> Database
```

Key Classes

Class	Responsibilities	Key Attributes/Methods
`AnalyticsService`	Logs user interactions and generates recommendations.	`+logSearch()`, `+getRecommendations()`

3. Key Relationships & Dependencies

Relationship	Description
Aggregation (`ProductSearchService` → `ProductAggregator`)	`ProductSearchService` uses `ProductAggregator` but does not own it.
Inheritance (`VendorAPIAdapter` ← `AmazonAdapter`)	`AmazonAdapter` extends the abstract `VendorAPIAdapter`.
Dependency (`AmazonAdapter` → `APIRateLimiter`)	`AmazonAdapter` depends on `APIRateLimiter` for API call throttling.
Association (`UserService` → `User`)	`UserService` interacts with `User` objects.

4. Summary of Key Operations

Class	Key Methods	Description
`ProductSearchService`	`search(query, filters)`	Fetches and ranks products.
`AmazonAdapter`	`fetchProducts(query)`	Retrieves Amazon product listings.
`UserService`	`getPreferences(userId)`	Returns user’s preferred filters.
`AnalyticsService`	`logPurchase(productId, userId)`	Records a purchase event.

5. Design Patterns Applied

Pattern	Applied In	Purpose
Factory Method	`VendorAPIFactory`	Creates vendor-specific adapters dynamically.

Pattern	Applied In	Purpose
Strategy	RankingStrategy	Allows flexible sorting algorithms.
Adapter	AmazonAdapter, EbayAdapter	Standardizes vendor API interactions.
Facade	API Gateway	Simplifies microservice access.

6. Conclusion

This class-level design model ensures:
Modularity (clear separation of concerns).
Extensibility (easy to add new vendors or ranking strategies).
Maintainability (well-defined responsibilities and relationships).

The next step would be refining database schemas, API contracts, and caching mechanisms for optimization.

4.2 Use Case Realization

This section details how each key use case from Section 2 is realized in the system design. We provide:

1. High-level module interactions (Sequence/Activity Diagrams).
2. Low-level class collaborations (Expanded Sequence Diagrams).

Use Case 1: Product Search & Comparison

Goal: Allow users to search for products and compare prices across multiple vendors.

1. High-Level Module Interaction (Sequence Diagram)

```
actor User
participant UI
participant API Gateway
participant Search Service
participant Vendor Integration
participant User Management
participant Analytics

User->>UI: Enters search query & filters
UI->>API Gateway: POST /search {query, filters}
API Gateway->>Search Service: search(query, filters)
Search Service->>Vendor Integration: fetchProducts(query)
Vendor Integration-->>Search Service: List<VendorProduct>
Search Service->>User Management: get Preferences(userId)
User Management-->>Search Service: User Preferences
Search Service->>Analytics: logSearch(query, userId)
Search Service-->>API Gateway: List<Product>
API Gateway-->>UI: Display ranked products
UI->>User: Shows product comparison
```

2. Low-Level Class Collaboration (Expanded Sequence Diagram)

```
participant Product Search Service
participant Vendor API Factory
participant Amazon Adapter
participant Product Aggregator
participant Ranking Strategy

Product Search Service->>Vendor API Factory: getAdapter("Amazon")
Vendor API Factory-->>Product Search Service: AmazonAdapter
Product Search Service->>Amazon Adapter: fetchProducts(query)
Amazon Adapter-->>Product Search Service: List<VendorProduct>
Product Search Service->>Product Aggregator: merge(products)
Product Aggregator-->>Product Search Service: List<Product>
Product Search Service->>Ranking Strategy: sort(products, "price")
Ranking Strategy-->>Product Search Service: Sorted List<Product>
```

Key Steps:

1. **User** submits a search query via **UI**.
2. **API Gateway** forwards the request to **ProductSearchService**.
3. **VendorAPIFactory** instantiates the correct **AmazonAdapter/EbayAdapter**.
4. **ProductAggregator** normalizes and merges results.
5. **RankingStrategy** sorts products (default: price ascending).
6. **Analytics** logs the search for recommendations.

Use Case 2: Purchase Redirection

Goal: Redirect users to the vendor’s checkout page when they click "Buy Now."

1. High-Level Module Interaction (Sequence Diagram)

```
actor User
participant UI
participant API Gateway
participant Vendor Integration
participant Analytics

User->>UI: Clicks "Buy Now" (productId)
UI->>API Gateway: POST /purchase {productId, userId}
API Gateway->>Vendor Integration: getCheckoutURL(productId)
Vendor Integration-->>API Gateway: checkoutURL
API Gateway->>Analytics: logPurchase(productId, userId)
API Gateway-->>UI: Redirect to checkout URL
UI->>User: Vendor checkout page
```

2. Low-Level Class Collaboration (Expanded Sequence Diagram)

```
participant Vendor API Factory
participant Amazon Adapter
participant API RateLimiter

Vendor Integration->>Vendor API Factory: getAdapter("Amazon")
Vendor API Factory-->>Vendor Integration: AmazonAdapter
Vendor Integration->>Amazon Adapter: getCheckoutURL(productId)
Amazon Adapter->>API Rate Limiter: canMakeRequest()
```

```
API Rate Limiter-->>Amazon Adapter: true
Amazon Adapter-->>Vendor Integration: checkoutURL
```

Key Steps:

1. **User** clicks "Buy Now" on a product.
2. **Vendor API Factory** creates the appropriate **Amazon Adapter**.
3. **API Rate Limiter** ensures the API call is within limits.
4. **Analytics** logs the purchase for future recommendations.
5. User is redirected to the vendor’s checkout page.

Use Case 3: Personalized Recommendations

Goal: Suggest products based on user’s search/purchase history.

1. High-Level Module Interaction (Sequence Diagram)

```
actor User
participant UI
participant API Gateway
participant Analytics
participant Search Service

User->>UI: Views homepage
UI->>API Gateway: GET /recommendations {userId}
API Gateway->>Analytics: getRecommendations(userId)
Analytics->>Search Service: getTrendingProducts()
Search Service-->>Analytics: List<Product>
Analytics-->>API Gateway: Personalized recommendations
API Gateway-->>UI: Display recommendations
UI->>User: Shows suggested products
```

2. Low-Level Class Collaboration (Expanded Sequence Diagram)

```
participant Analytics Service
participant User Service
participant Product Search Service

Analytics Service->>User Service: getPreferences(userId)
User Service-->>Analytics Service: User Preferences
Analytics Service->>Product Search Service: search(basedOnPreferences)
Product Search Service-->>Analytics Service: List<Product>
```

Key Steps:

1. **Analytics Service** fetches user preferences via **User Service**.
2. **Product Search Service** retrieves products matching preferences.
3. Results are ranked and displayed on the UI.

Summary of Use Case Realizations

Use Case	**Key Modules Involved**	**Critical Classes**
Product Search & Comparison	UI, API Gateway, Search, Vendor Integration	ProductSearchService, AmazonAdapter, RankingStrategy
Purchase Redirection	UI, API Gateway, Vendor Integration, Analytics	VendorAPIFactory, APIRateLimiter
Personalized Recommendations	UI, Analytics, Search, User Management	AnalyticsService, UserService

Design Patterns Applied

- **Strategy Pattern** → Dynamic ranking algorithms (RankingStrategy).
- **Factory Pattern** → Vendor adapter creation (VendorAPIFactory).
- **Observer Pattern** → Analytics logging on user actions.

This breakdown ensures **traceability** from use cases to actual class-level implementations. Next steps:

- **Error handling** (e.g., API failures).
- **Performance optimizations** (caching frequent searches).

5. Data View

5.1 Domain Model

Represents the core entities and relationships in the system:

1. **User**:
    - Attributes: googleId (from Google OAuth), email, preferences.
    - Relationships: Linked to Product (tracked items) and SearchHistory.
2. **Product**:
    - Attributes: productId, name, priceHistory, vendor, category.
    - Relationships: Aggregated from multiple vendors (Amazon, Flipkart).
3. **SearchHistory**:
    - Attributes: query, timestamp, filters.
    - Relationships: Linked to User via googleId.

5.2 Data Model (persistent data view)

MongoDB Collections

Collection	**Fields**	**Description**
Users	googleId (PK), email, preferences	Stores user preferences (e.g., deal alerts) linked to Google OAuth IDs.
Products	productId (PK), name, priceHistory, vendor, category	Aggregated product data from vendors.
SearchHistory	googleId, query, timestamp, filters	Logs user search queries for personalized recommendations.

Redis Cache Structure

Key Format	Data Type	Description
search:{query}	Hash	Cached search results (e.g., product IDs, prices) for frequent queries.
session:{jwt}	String	Temporary user session data (e.g., JWT validity, user roles).

5.2.1 Data Dictionary

Name	Type	Description	Possible Values/Format
User_ID	*Integer*	Unique identifier for each user	Auto-incremented integer
Username	*String*	Customer's chosen username	Alphanumeric (max 20 characters)
Email	*String*	Customer's email address	Valid email format
Password	*String*	Hashed password for account security	Encrypted string
Product_ID	*Integer*	Unique identifier for each product	Auto-incremented integer
Product_Name	*String*	Name of the product	Text (max 100 characters)
Price	*Float*	Current product price	Currency format (e.g., 29.99)
Retailer	*String*	Name of the retailer	Text
Rating	*Float*	Average customer rating for the product	Range from 0.0 to 5.0
Deal_Alert_Threshold	*Float*	Price threshold set by a user to trigger a deal notification	Currency format
Price_History	*Array/Record*	Historical pricing data for a product	Array of floats with timestamps
Search_Query	*String*	User-entered search term	Text
Filter_Criteria	*Object*	Criteria used to filter product search results	JSON object (e.g., price range, brand, rating)
Recommendation_List	*Array*	List of product IDs recommended to the user	Array of integers (Product_IDs)
Notification	*Object*	Details of alerts or messages sent to the user	JSON object (e.g., message, timestamp)
User_Preferences	*Object*	Settings for personalized recommendations	JSON object (e.g., preferred categories, price range)

Session_State	*String*	Current state of the user session	"Active", "Inactive", "Expired"
Timestamp	*Date/Time*	Date and time for transactions or system events	Standard datetime (YYYY-MM-DD HH:MM:SS)
API_Response	*JSON*	Data received from external e-commerce API requests	JSON format
API_Status	*Integer/String*	Status code returned from API requests	HTTP status codes (e.g., 200, 404, 500)
MAX_RESULTS_PER_PAGE	*Constant*	Maximum number of products displayed per search result page	Integer (default: 20)
MIN_RATING	*Constant*	Minimum rating filter for product searches	Float (range: 0.0 to 5.0)

6. Exception Handling

This section describes the exceptions defined within the Shopping Aggregator application, the circumstances in which they can occur, how they are logged, and the expected follow-up actions.

6.1 Exception Categories

The system categorizes exceptions into the following types:

1. **Authentication Exceptions**

- **Circumstances**: Invalid Google OAuth tokens, expired sessions, or unauthorized access attempts.
- **Handling**: Log the event, revoke the token if compromised, and redirect the user to the login page.
- **Logging**: Logs include timestamp, userId, IP address, and error details.

2. **API Integration Exceptions**

- **Circumstances**: Vendor API failures (e.g., Amazon/Flipkart API downtime, rate limits exceeded).
- **Handling**: Retry with exponential backoff (max 3 attempts); fall back to cached data if unavailable.
- **Logging**: Logs include API endpoint, response status, timestamp, and retry attempts.

3. **Database Exceptions**

- **Circumstances**: MongoDB/Redis connection timeouts, query failures.
- **Handling**: Retry logic for transient errors; notify admins for persistent issues.
- **Logging**: Logs include query, error code, and database state.

4. **Validation Exceptions**

- **Circumstances**: Invalid user input (e.g., malformed search queries, incorrect filter ranges).
- **Handling**: Return HTTP 400 with descriptive error messages.
- **Logging**: Logs include input data and validation rules violated.

5.

- **Circumstances**: Stripe/PayPal API failures, declined transactions.
- **Handling**: Notify users and log transaction details for manual review.

6.2 Logging and Monitoring\

- **Tools**:
  - **Prometheus/Grafana** for real-time exception tracking.
  - **AWS CloudWatch** for centralized logs.
- **Log Fields**: timestamp, severity (ERROR/WARN), exception type, stack trace, context (e.g., userId).
- **Alerts**: Slack/PagerDuty notifications for critical exceptions (e.g., DB outages).

6.3 Recovery Actions

- **Automated**: Retry transient errors (e.g., API timeouts).
- **Manual**: Admin intervention for unresolved issues (e.g., corrupted data).
- **User Communication**: Friendly error messages (e.g., "Search temporarily unavailable – try again later").

7. Configurable Parameters

This section lists dynamic and static configuration parameters used by the Shopping Aggregator.

7.1 Simple Configurations (Name/Value Pairs)

Configuration Parameter Name	Definition and Usage	Dynamic?
CACHE_TTL_SECONDS	Time-to-live for Redis cached search results (default: 300s).	Yes
GOOGLE_OAUTH_CLIENT_ID	Google OAuth 2.0 client ID for authentication.	No
MAX_API_RETRIES	Maximum retries for failed vendor API calls (default: 3).	Yes
SEARCH_RESULTS_PER_PAGE	Pagination limit for product listings (default: 20).	Yes
PRICE_UPDATE_INTERVAL	Frequency (in minutes) for refreshing product prices (default: 15).	Yes
ADMIN_EMAILS	Comma-separated emails for system alerts.	Yes

7.2 Complex Configurations (XML/JSON Schema)

- **Vendor API Rate Limits**:

```
{
  "amazon": {
    "requests_per_minute": 50,
    "backoff_ms": 1000
  },
  "flipkart": {
    "requests_per_minute": 100,
    "backoff_ms": 500
  }
}
```

- **Feature Toggles**:

```
{
  "enable_ai_recommendations": true,
  "enable_price_alerts": false
}
```

7.3 Dynamic vs. Static Parameters

- **Dynamic**: Adjustable without restart (e.g., CACHE_TTL_SECONDS).
- **Static**: Require redeployment (e.g., GOOGLE_OAUTH_CLIENT_ID).

7.4 Override Mechanisms

- **Environment Variables**: For cloud deployments (AWS Elastic Beanstalk).
- **Admin Dashboard**: UI toggles for dynamic parameters (e.g., disable caching).

Key Takeaways:

- Exception Handling: Focuses on resilience (retries, fallbacks) and auditability (detailed logs).
- Configurable Parameters: Balance flexibility (dynamic changes) and security (sensitive credentials).
- Alignment with SRS: Meets non-functional requirements for reliability (Section 4.1) and security (Section 4.2).

8. Quality of Service

This section outlines the non-functional requirements (NFRs) and design considerations to ensure:
- High Availability
- Security & Compliance
- Performance & Scalability
- Monitoring & Fault Tolerance

1. Availability

Goal: Ensure the system is operational 99.9% of the time (uptime SLA).

Strategies

Approach	Implementation
Multi-Region Deployment	Deploy in AWS us-east-1 (primary) and us-west-2 (failover) with Route53 DNS failover.

Approach	Implementation
Redundant Microservices	Run multiple instances of critical services (e.g., SearchService) using Kubernetes pods.
Database Replication	Use **Amazon RDS Multi-AZ** for automatic failover.
Circuit Breakers	Implement Resilience4j to halt requests to failing vendor APIs (e.g., Amazon downtime).

Recovery Mechanisms

- **Retry Policies**: Exponential backoff for transient vendor API failures.
- **Fallback Responses**: Serve cached product data if real-time APIs are unavailable.

2. Security

Goal: Protect user data and comply with GDPR/CCPA.

Strategies

Approach	Implementation
Authentication	OAuth 2.0 + JWT via **Auth0** or **Amazon Cognito**.
Data Encryption	Encrypt PII at rest (AES-256) and in transit (TLS 1.3).
Rate Limiting	Block >100 requests/min per IP using **API Gateway throttling**.
Vendor API Security	Store vendor API keys in **AWS Secrets Manager** (rotated monthly).
Audit Logging	Log all sensitive actions (e.g., purchases) for compliance (stored in **S3 + Glacier**).

Compliance Checks

- **PCI-DSS** for payment redirections.
- **SOC 2** audit for data handling processes.

3. Performance & Scalability

Goal: Respond to 90% of searches under **500ms** at peak load (10,000 RPM).

Strategies

Approach	Implementation
Caching	Cache frequent search results for 5 mins using **Redis**.
Async Processing	Use **Kafka** to decouple analytics logging from critical path.

Approach	Implementation
Horizontal Scaling	Auto-scale SearchService pods based on CPU >70% (K8s HPA).
CDN for Static Assets	Serve UI assets via **CloudFront**.

Benchmarks

Scenario	Latency Target	Throughput Target
Product search (cold)	≤800ms	5,000 RPM
Product search (cached)	≤200ms	20,000 RPM
Purchase redirection	≤300ms	1,000 RPM

4. Monitoring & Control

Goal: Detect and resolve issues before users are impacted.

Strategies

Approach	Implementation
Real-Time Metrics	Track latency, errors, and throughput via **Prometheus + Grafana**.
Distributed Tracing	Trace requests across microservices using **AWS X-Ray**.
Alerting	Slack/PagerDuty alerts for: • API error rate >1% • Latency >1s (p99).
Synthetic Monitoring	Run hourly scripted searches via **AWS CloudWatch Synthetics**.

Key Metrics to Monitor

Metric	Threshold	Action
API error rate	>1% for 5 mins	Roll back last deployment.
Vendor API latency	>2s (p95)	Switch to backup vendor (e.g., eBay).
Cache hit ratio	<80%	Increase cache TTL or capacity.

5. Disaster Recovery (DR) Plan

Scenario: Primary AWS region (us-east-1) outage.

Recovery Steps

1. **DNS Failover**: Route53 redirects traffic to [redacted].
2. **Database**: Promote RDS read replica to primary.
3. **Microservices**: Spin up pods in secondary region from ECR.
4. **Data Sync**: Replicate user sessions via **ElastiCache Global Datastore**.

Recovery Time Objective (RTO): ≤15 mins
Recovery Point Objective (RPO): ≤5 mins (data loss window)

6. Cost Optimization

Area	Optimization Strategy
Compute	Use **AWS Spot Instances** for stateless services (e.g., [redacted]).
Database	Archive old search logs to **S3 Glacier**.
Caching	Downsize Redis clusters during off-peak hours.

8.1 Availability

1. Business Requirement

The system must maintain **99.9% uptime** (≤43.8 minutes of downtime/month) to ensure uninterrupted price comparisons and purchase redirections for users.

2. Design Strategies for High Availability

2.1 Fault Tolerance

Component	Design Implementation	Impact on Availability
Microservices	Deploy redundant instances across 3+ availability zones (AWS).	Eliminates single-point failures.
Database	Use **Amazon RDS Multi-AZ** with automatic failover.	<5s downtime during primary DB failure.
Vendor APIs	Circuit breakers (Resilience4j) + fallback cached data after 2 retries.	Prevents cascading failures.
API Gateway	Distribute traffic via AWS ALB with health checks.	Routes away from unhealthy nodes.

2.2 Zero-Downtime Deployments

Strategy	Implementation
Blue-Green Deploys	Route traffic to new version only after full health checks (using AWS CodeDeploy).

Strategy	Implementation
Database Migrations	Schema changes applied via backward-compatible flyway scripts (no table locks).

2.3 Disaster Recovery (DR)

Scenario	Recovery Mechanism	RTO/RPO
AWS Region Outage	Route53 DNS failover to secondary region (us-west-2).	RTO: 5 mins, RPO: 1 min
Database Corruption	Restore from cross-region snapshots (automated daily).	RTO: 15 mins, RPO: 24h

3. Potential Availability Risks & Mitigations

3.1 Scheduled Maintenance

Activity	Impact	Mitigation
Data Batch Loading	10-min latency spike during ETL.	Throttle jobs to use ≤50% DB CPU; run during off-peak hours (12 AM–4 AM UTC).
Housekeeping	Brief API timeouts.	Serve stale cache data during cleanup (e.g., Redis cache pruning).

3.2 Vendor API Dependencies

Risk	Mitigation
Amazon API rate limits exceeded.	Exponential backoff + serve cached data from last successful fetch.
eBay API downtime.	Automatic traffic shift to alternate vendors (e.g., Walmart API).

3.3 Database Schema Updates

- **Risk**: Schema locks during migrations.
- **Mitigation**: Use **online schema migration tools** (e.g., AWS DMS) to avoid table locks.

4. Monitoring & Proactive Measures

4.1 Key Metrics

Metric	Threshold	Alert Action
Service uptime	<99.9% (monthly)	Trigger incident review + SLA credit process.

Metric	Threshold	Alert Action
Failed health checks	>5% for 5 mins	Auto-restart pods/K8s nodes.

4.2 Synthetic Transactions

- **Implementation**: Simulate user searches every 5 mins via AWS CloudWatch Synthetics.
- **Action**: If 3 consecutive failures → auto-failover to backup region.

8.2 Security and Authorization

1. Business Requirements

The system must enforce:

- **Role-Based Access Control (RBAC)** for admin vs. end-users.
- **Data Isolation**: Users can only access their own search/purchase history.
- **Vendor API Security**: Secure storage/rotation of third-party API keys.
- **GDPR/CCPA Compliance**: Anonymize PII after 6 months of inactivity.

2. Authorization Framework

2.1 Role Definitions

Role	Permissions
End-User	Search products, view personal history, initiate purchases.
Admin	Manage vendor API keys, audit logs, and user bans.
Analyst	Access aggregated analytics (no PII).

2.2 Access Control Implementation

Component	Technology	Details
Authentication	OAuth 2.0 + JWT (Auth0/Cognito)	JWT claims include `role` and `userId`.
API Authorization	Spring Security `@PreAuthorize` annotations (e.g., `@PreAuthorize("hasRole('ADMIN')")`).	
Data Filtering	Hibernate Filters (auto-applies `WHERE userId = :currentUser` to DB queries).	

Example Policy:

java
Copy

```
// Ensure users only access their own data
@PreAuthorize("#userId == authentication.principal.id")
public PurchaseHistory getHistory(String userId) { ... }
```

3. Application-Specific Security Design

3.1 Vendor API Key Management

Requirement	Implementation
Secure Storage	Encrypted in AWS Secrets Manager (auto-rotated monthly).
Access Control	Only Admin role can add/rotate keys (audit logs track changes).

3.2 GDPR/CCPA Compliance

Feature	Design
Right to Erasure	Anonymize user data via scheduled AWS Lambda (retain purchase logs without PII).
Data Export	UserService provides JSON dumps of personal data upon request.

3.3 Rate Limiting

- **End-Users**: 100 requests/minute (prevent scraping).
- **Admins**: 1,000 requests/minute (enforced via API Gateway).

4. User Access Management

4.1 Admin Interface

Features:

- **User Role Assignment**: Dropdown to assign roles (Admin/Analyst).
- **Access Reviews**: Monthly audit reports of admin actions.
- **Self-Service**: Users can delete accounts via UI (triggers anonymization).

Tech Stack:

- **Frontend**: React + Material UI.
- **Backend**: Protected by @PreAuthorize("hasRole('ADMIN')").

4.2 Workflow for Access Changes

1. **Request**: User submits access request via UI.
2. **Approval**: Admin reviews in dashboard.
3. **Provisioning**: Cognito user pool updated (event logged in DynamoDB).

5. Monitoring & Auditing

5.1 Key Security Metrics

Metric	Tool	Alert Threshold
Failed logins	AWS GuardDuty	>5 attempts in 5 mins.
Unauthorized API calls	CloudTrail	Any 403 response.
JWT tampering	Auth0	Automatic token revocation.

5.2 Audit Logs

- **Stored**: AWS S3 + Glacier (immutable logs).
- **Fields**: timestamp, userId, action, IP address.

6. Conclusion

Key Strengths

✓ **Least Privilege**: Fine-grained RBAC.
✓ **Compliance**: Automated PII handling for GDPR/CCPA.
✓ **Auditability**: Immutable logs for all admin actions.

Trade-offs

- **Latency**: JWT validation adds ~50ms to API calls.
- **Complexity**: Requires training for admins on Cognito/Secrets Manager.

Next Steps:

- Conduct **penetration testing** (OWASP ZAP).
- Implement **just-in-time (JIT) access** for admins (e.g., AWS IAM Roles Anywhere).

This design ensures **secure, compliant, and manageable** access control aligned with business needs.

8.3 Load and Performance Implications

1. Business Transaction Load Projections

Based on market analysis, the system must handle:

Scenario	Peak Load	Performance Target
Product Searches	10,000 RPM	≤500ms latency (p95)
Purchase Redirections	1,000 RPM	≤300ms latency (p95)
User Recommendations	5,000 RPM	≤200ms latency (p95)

2. Implications on Design Components

2.1 Search & Aggregation Module

Key Components Affected:

- `ProductSearchService`
- `VendorIntegration` (Amazon/eBay APIs)
- `ProductCache` (Redis)

Design Adjustments:

Requirement	Implication	Solution
10,000 RPM searches	Vendor APIs may throttle (>50 calls/sec to Amazon).	**Aggressive Caching**: Cache results for 5 mins (Redis). Misses trigger real-time API calls.
500ms latency	Sequential vendor API calls are too slow.	**Parallel Fetching**: Use `CompletableFuture` to call vendors concurrently.
High-cardinality queries	80% of searches target 20% of products (e.g., "iPhone 15").	**Hotkey Optimization**: Pre-cache trending products.

Database Growth:

- **Projection**: 100M product records/year (2TB storage).
- **Indexing**: Elasticsearch for text search; RDS PostgreSQL for transactional data.

2.2 Vendor Integration Module

Key Components Affected:

- `VendorAPIFactory`
- `APIRateLimiter`

Design Adjustments:

Requirement	Implication	Solution
Rate limits	Amazon allows 1 req/sec/token; eBay allows 5,000 calls/day.	**Token Pooling**: Rotate 10+ API tokens (Secrets Manager).

Requirement	Implication	Solution
API Latency	Vendor responses vary (200ms–2s).	**Timeout Handling**: Fail after 1s → serve stale cache.

Monitoring:

- **CloudWatch Alerts**: Vendor API errors >1% for 5 mins trigger fallback mode.

2.3 Database Layer

Key Tables:

Table	Growth Rate	Partitioning Strategy
`products`	50GB/month	Shard by `vendor_id` (UUID).
`user_searches`	20GB/month	Time-based (monthly partitions).

Performance Optimizations:

- **Read Replicas**: 2 replicas for analytics queries.
- **Connection Pooling**: HikariCP with 100 max connections.

3. Infrastructure Scaling

3.1 Compute Resources

Service	Baseline	Scaling Trigger
SearchService	10 K8s pods	CPU >70% → scale to 50 pods.
Redis Cache	3 nodes (r6g.large)	Cache hit ratio <80% → add nodes.

3.2 Network Throughput

- **Egress**: 1 Gbps (upgrade to 5 Gbps if cross-region traffic >50%).
- **API Gateway**: 10,000 requests/sec (AWS ALB auto-scaling).

4. Load Testing Plan

4.1 Test Scenarios

Test Case	Simulated Load	Success Criteria
Peak Search Traffic	12,000 RPM	Latency ≤500ms, error rate <0.1%.

Test Case	Simulated Load	Success Criteria
Vendor API Failure	100% eBay downtime	Fallback to cached data in ≤1s.

4.2 Tools

- **Load Generation**: Locust (Python) for realistic user flows.
- **Monitoring**: Prometheus + Grafana (track latency, errors, throughput).

5. Failure Mode Analysis

Risk	Mitigation
Redis outage	Serve stale DB data (5% latency increase).
RDS CPU saturation	Throttle non-critical writes (e.g., analytics logs).
Vendor API quota exhaustion	Prioritize high-value vendors (e.g., Amazon over Walmart).

8.4 Monitoring and Control

1. Controllable Processes

1.1 Message Handlers

Process	Purpose	Control Mechanism
SearchQueryHandler	Processes real-time product search requests from the queue (Kafka/SQS).	**Auto-scaling**: Spins up pods when queue depth >100.
PurchaseEventLogger	Asynchronously logs purchase events for analytics.	**Retry Policy**: 3 attempts → DLQ on failure.
CacheWarmerDaemon	Preloads trending products into Redis every 30 mins.	**Manual Trigger**: Admin API endpoint (`/cache/warm`).

1.2 Scheduled Daemons

Daemon	Schedule	Function
VendorAPIHealthChecker	Every 5 mins	Tests connectivity to vendor APIs; switches to backup vendors if errors >5%.
UserDataAnonymizer	Daily at 2 AM	GDPR compliance: Anonymizes inactive user data (>6 months).
SearchLogArchiver	Weekly	Moves old search logs from PostgreSQL to S3 Glacier.

2. Measurable Values for Monitoring

2.1 Key Metrics Published

Metric	Source	Alert Threshold	Monitoring Tool
API Latency (p95)	API Gateway	>500ms for 5 mins	Prometheus + Grafana
Cache Hit Ratio	Redis	<80%	AWS CloudWatch
Vendor API Error Rate	VendorIntegration	>1% for 10 mins	Datadog
Kafka Consumer Lag	SearchQueryHandler	>1,000 messages	Confluent Control Center
DB Replication Lag (ms)	RDS PostgreSQL	>1,000ms	AWS RDS Monitoring

2.2 Custom Application Logs

Log Type	Fields	Use Case
SearchAuditLog	`timestamp`, `userId`, `query`, `vendorCount`, `responseTimeMs`	Fraud detection & SLA tracking.
PurchaseRedirection	`productId`, `vendor`, `ipAddress`, `isFallback`	Conversion rate analysis.

3. Control Interfaces

3.1 Admin API Endpoints

Endpoint	Method	Parameters	Purpose
`/admin/cache/purge`	POST	`vendor` (optional)	Force-clears Redis cache.
`/admin/scale-handlers`	PUT	`desiredReplicas`	Manually adjust Kafka consumer pods.
`/admin/vendor/disable`	POST	`vendorId`, `reason`	Temporarily block a vendor.

3.2 Dashboard Controls (Grafana)

- **Manual Overrides**:
  - Toggle caching on/off for debugging.
  - Adjust rate limits in real-time.

4. Alerting & Auto-Remediation

4.1 Critical Alerts

Condition	Action
Database CPU >90% for 15 mins	Auto-scale read replicas + throttle non-critical writes.
Redis memory >95%	Trigger LRU eviction + alert admins.
5xx Errors >5%	Roll back last deployment (Canary analysis).

4.2 Notification Channels

- **PagerDuty**: For Sev1/Sev2 incidents (e.g., DB outage).
- **Slack**: For warnings (e.g., cache hit ratio drop).

---

5. Example Workflow: Handling Vendor API Failure

1. **Detection**: `VendorAPIHealthChecker` logs errors >5%.
2. **Alert**: Slack notification + PagerDuty if Amazon API is down.
3. **Fallback**: `ProductSearchService` switches to cached data.
4. **Recovery**: Auto-retry every 1 min; resume normal ops when healthy.