|  |
| --- |
| **C2C Marketplace** |
| Architecture Design Document |
|  |
|  |
| **2025-09-25** |
| **Nikhil Gupta** |

This document is an Architecture Design Document for developing **C2C Marketplace**.

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Description |
| 0.1 | 2025-09-25 | Nikhil Gupta | Initial document creation |
| 0.2 | 2025-10-11 | Nikhil Gupta | Pre-final Report Submission |
| 0.3 | 2025-10-19 | Nikhil Gupta | Changes according to Feedback |
| 0.4 | 2025-10-26 | Nikhil Gupta | Final Report Submission |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

[1. Overview 3](#_Toc516321189)

[2. Requirements 7](#_Toc516321190)

[2.1. Functional Requirements 7](#_Toc516321191)

[2.2. Non-functional Requirements 12](#_Toc516321192)

[2.3. Quality Attributes 13](#_Toc516321193)

[3. Architecture 14](#_Toc516321194)

[4. Modules 15](#_Toc516321195)

[Appendix 16](#_Toc516321196)

[A. Domain Model 17](#_Toc516321197)

[B. Quality Scenarios 18](#_Toc516321198)

[C. Quality Scenario Analysis 19](#_Toc516321199)

[D. Candidate Architectures 20](#_Toc516321200)

[E. Candidate Architecture Evaluation 21](#_Toc516321201)

[F. Architecture Design 22](#_Toc516321202)

[G. Architecture Evaluation(ATAM) 23](#_Toc516321203)

# Overview

// A1. System Definition

// C1-1. Is the defined system boundary clear?

// C1-2. Is there sufficient explanation of System’s operation and business environment as business drivers?

* 1. **Introduction**

A Customer-to-Customer (C2C) Marketplace is a platform that allows users to **buy and sell** both used and new products. It provides a digital space where individuals can easily list their items and buyers can discover items they want using advanced search. The Marketplace should ensure safety and security for all the operations that happen in the system.

* 1. **System Definition**

The purpose of this project is to design a Customer-to-Customer (C2C) Marketplace.

Figure 1 below depicts the System Boundary and how C2C Marketplace will interact with the outside Components.

* System will provide interface for Actors like Buyer, Seller & Moderators.
* System will interact with an external Notification Service Interface & External LLM.

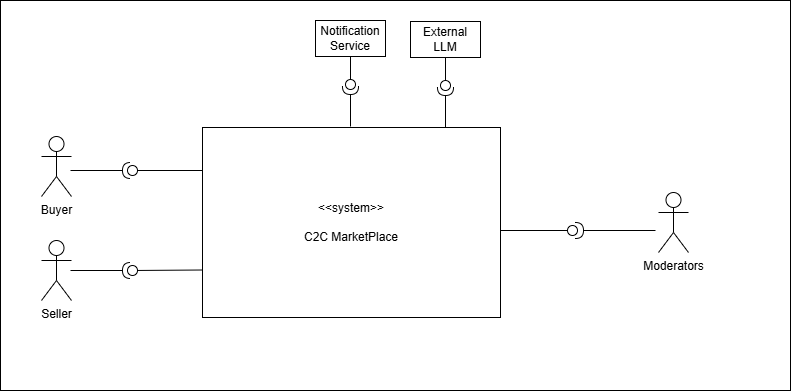


Figure 1: **System boundary for C2C Marketplace**

* 1. **Business Context & Drivers**

In a modern C2C marketplace, primary business challenge is to create a trusted, cohesive ecosystem where the user journey from discovery to final purchase is not only possible, but also intuitive and secure.

To achieve this, our architecture will be guided by these core principles:

* **Easy Discovery**: We will help buyers find what they want quickly. This will be done through a simple, effective search feature and by showing personalized recommendations on the home page.
* **Seamless Listing Experience**: We will make it effortless for sellers to list their items. The process will be smooth and guided, with helpful features like auto-saving drafts and a final preview before publishing.
* **~~Building Community Trust~~**~~: We will maintain a trusted environment in two key ways:~~
  + ~~First, use smart technology (AI) to proactively detect fraud by automatically checking new listings and chat messages for suspicious content before it becomes a problem.~~
  + ~~Second, we will empower our users to help keep the community safe by providing direct and easy-to-use tools for reporting suspicious content and appealing moderation decisions.~~

# Requirements

## Functional Requirements

// A2. Functional Requirement Specification

// C2-1. Is there sufficient functional requirement specification to affect System’s architecture?

// C2-2. Is the relationship between use cases clear?

// C2-3. Is the division of use cases explicit?

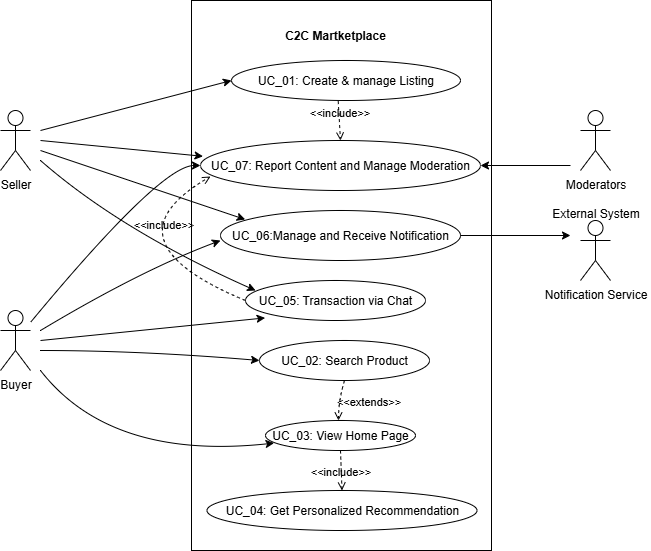


Figure 2: C2C Marketplace - Use Case Diagram

|  |  |
| --- | --- |
| **UC\_01** | Create & Manage Listing |
| Description | User can Create a new product listing & Update or Delete an existing product listing. |
| Actor | User (Seller) |
| Pre-condition | The User must be signed-in and have an authorized account to create a listing. |
| Post-condition | **On Success**: A new listing is created in the system.  **On Failure**: The listing is not created, and the Seller is notified of the reason for the failure. |
| Basic Flow | 1. User requests the system to create a new item listing.  2. User uploads images and enters common item details (like Product Category, Title, Description, Price, and Location. The user can also specify if it is a premium listing (Y/N). (AF1)  3. Based on the category the user selected, the system automatically shows additional, relevant fields for the user to complete.  4. System runs content moderation checks on all the filled-in data and confirms it passes the validation. (AF2)  5. System opens a preview page showing the user how their listing will appear to other users.  6. User reviews and validates the data on the preview page, then publishes the listing.  7. System confirms that the new listing has been successfully created. |
| Additional Flow | AF1. Auto Save Draft: While the User is entering details (during steps 2-3), the system automatically saves the progress as a draft. This allows the user to restore the last saved version if interrupted.  AF2: Content Moderation Failure:  1. If the system's content moderation check detects a policy violation, the listing process is paused.  2. System shows the specific issue to the Seller and offers potential solutions.  3. User can then either correct the details & resubmit the form, or they may be given an option to submit the listing for a manual review. |

|  |  |
| --- | --- |
| **UC\_02** | Search Product |
| Description | User can search for an Item by entering keywords & system returns the results by using semantic search on entered Keywords and location. |
| Actor | User (Buyer) |
| Pre-condition | 1. Listings are already created and are indexed  2. User location is available |
| Post-condition | Results returned with categorical breakdowns |
| Basic Flow | 1. User requests the system to search products (with keywords and filters (optional).  2. System executes search query (both keyword searching and semantic search) using all the inputs provided by user.  3. System also finds suitable ads that can be merged in the searched results.  4. System intelligently merges and re ranks the listings by relevance.  5. System displays a list of (paginated) product listings that match the criteria.. (AF1) |
| Additional Flow | AF1: No Results Found: This flow occurs at Step 5 if no listings match the Buyer's search criteria. The system displays a "No results found" message.  To be helpful, the system also suggests alternative actions, such as checking for spelling errors, removing filters, or it may display popular items from a similar category. |

|  |  |
| --- | --- |
| **UC\_03** | View Home Page |
| Description | This use case describes what happens when a user opens the homepage of the app or web interface, which is then loaded along with recommendations for the user |
| Actor | User |
| Pre-condition | Internet Connection should be present. |
| Post-condition | On success, the system will load the homepage along with the recommendation data. |
| Basic Flow | 1. User opens the homepage of the application or web interface.  2. System identifies the authenticated user and initializes their session.  3. System fetches personalized content feeds for the Buyer, such as "Recommended For You," "New Items from Sellers You Follow," and "Recently Viewed." (UC\_04) (AF1)  4. System also fetches general content like "Trending Near You" and featured categories.  5. System assembles and displays the home page, prioritizing personalized content and ensuring a search bar is prominently visible. |
| Additional Flow | AF1: New or Guest User, System has no personalization data. Instead of a personalized feed, it will return a generic feed consisting of trending items, location-based bestsellers, or recently listed Items. |

|  |  |
| --- | --- |
| **UC\_04** | Get Personalized Recommendations |
| Description | This use case describes how the system provides proactive, context-aware product recommendations to a user in various situations, such as when they are viewing the home page or a specific product. |
| Actor | User (Buyer) |
| Pre-condition | User is logged in. |
| Post-condition | System displays a set of relevant personalized items to the user. |
| Basic Flow | Scenario A: Recommendations on the Home Page  1. Trigger: User opens the home page.  2. System analyzes the Buyer's profile, recent activity (views, searches), and location to understand their intent.  3. System generates and displays several personalized feeds, such as "Recommended For You," "Because You Viewed X," and "New Items Nearby." (AF1)  3. System also considers relevant targeted ads in the system and creates a final curated list of recommended items.  4. System displays the recommendations to the user on the home page.  Scenario B: Product Detail Page Recommendations  1. Trigger: User views a specific product's detail page.  2. System analyzes the current product's attributes (category, brand, price).  3. System generates and displays contextual recommendations designed to increase conversion, such as "Similar Items" (alternatives at different price points) and "Complementary Items" (accessories or related products).  4. System shows these related Items to the user. |
| Additional Flow | AF1. Cold start: For new users, use popular listings of user location as a fallback. |

|  |  |
| --- | --- |
| **UC\_05** | Secure Chat Messenger |
| Description | This use case describes how a User (a Buyer and a Seller) communicate in real-time through a secure messaging feature. It allows the Buyer to ask questions about an item and for both parties to negotiate a price by making formal offers and counter-offers, leading to a sales agreement. |
| Actor | User (Buyer & Seller) |
| Pre-condition | 1. Users are logged in.  2. A product listing must be active and available in the system. |
| Post-condition | 1. On success, an offer is accepted, the system creates a formal order, and the listing is marked as sold.  2. On failure, the negotiation ends without an agreement.  3. The chat history is saved for future reference. |
| Basic Flow | 1. User navigates to a product listing and initiates a chat.  2. System opens a secure chat window for both users.  3. Users can exchange text messages to discuss the item details.  4. After discussions, user submits a formal offer with a specific price.  5. System presents this offer to the other user.  6. The other user accepts the offer. (AF1)  7. This flow (steps 2-4) continues until an offer is accepted or one of the users ends the conversation.  8. System confirms the agreement to both the users, and marks the items as sold. |
| Additional Flow | AF1: Counter Offer: At Step 6, instead of accepting, the user can decline the offer and submit a counter-offer with a different price. The flow then returns to Step 5, with the Buyer now having the option to accept or decline.  AF2: Automated Content Filtering: If a message contains content that violates policies, such as requests for off-platform payments or personal information, System automatically blocks the message to prevent frauds.  AF3: Spam Prevention: System monitors message frequency and will temporarily limit a user's ability to send messages if they are suspected of spam or abuse. |

|  |  |
| --- | --- |
| **UC\_06** | Manage and Receive Notifications |
| Description | This use case describes how a user customizes their notification settings to control how and when they receive alerts from the system. It also covers how a user interacts with a received notification to navigate directly to relevant content within the application |
| Actor | User (Buyer/Seller) |
| Pre-condition | User account exists with verified contact details (email & contact number) to receive notifications. |
| Post-condition | 1. The user's notification preferences are saved and applied to all future alerts.  2. Upon interacting with a notification, the user is successfully navigated to the correct screen. |
| Basic Flow | 1. User navigates to their "Notification Preferences" screen within the account settings.  2. System displays all notification categories along with the available channels (e.g., Push, In-App, Email, SMS) for each.  3. User adjusts the settings to their preference and saves the changes.  4. Sytem confirms that the preferences have been successfully updated and will be used in future |
| Additional Flow | AF1: Interacting with a Received Notification:  1. A user receives an alert (e.g., a push notification on their device) that an event has occurred in the system.  2. User taps on the notification.  3. System opens the application and navigates the user directly to the relevant screen (e.g., to the specific chat for a new message). |

## Non-functional Requirements

// A5. Quality Requirement Specification

// C5-2. Is the specification of quality requirements appropriate?

// C5-3. Is quality requirement measurable?

// C5-4. Is the allowance of non-functional requirement clear?

|  |  |  |
| --- | --- | --- |
| **NFR\_01** | **Performance** | **Search operation response time should be low** |
| Description | Searching of an item using user input along with filters and user location should be low | |
| Environment | During peak operational hours, with the system handling a high volume of concurrent users. | |
| Stimulus | A user submits a search query with optional filter | |
| Response | 1. System receives the GET /search request at the API Gateway. 2. The Search Service validates the query and filter parameters (e.g., location, category, price range). 3. Search Service executes the compiled query against the dedicated Search Index cluster. 4. Search Service receives the list of matching listing IDs from the index. 5. Search Service optionally re-ranks the results based on business logic (e.g., boost new sellers, seller rating). 6. System returns the final, paginated list of listings to the client. | |
| Measure | [Response Time] = [time when final ordered list is returned] – [time of user request] | |
| Allowance | [Response Time] <= N milliseconds | |

|  |  |  |
| --- | --- | --- |
| **NFR\_02** | **Performance** | **Home Screen Loading time** |
| Description | The maximum time allowed for the backend to aggregate and deliver all the data required to render the user's home screen, including personalized content | |
| Environment | Under normal operational conditions when a user opens the application. | |
| Stimulus | User opens the app/home route; client requests home payload. | |
| Response | 1. System receives the request, identifies the user, and initiates calls to various downstream services (e.g., recommendations, listings) to retrieve the necessary content feeds. 2. System receives the GET /home request at the API Gateway. 3. The Home Controller identifies the user and initiates *parallel* calls to downstream services (e.g., Recommendation Service, Listing Service for categories). 4. Home Controller waits for all critical downstream services to respond or time out. 5. System aggregates the content from all responses into a single JSON data payload. 6. System returns the complete data payload to the user's client. | |
| Measure | [Response Time] = [time when data payload is returned] – [time when request is initiated]. | |
| Allowance | [Response Time] <= N milliseconds | |

|  |  |  |
| --- | --- | --- |
| **NFR\_03** | **Performance** | **Service Scalability** |
| Description | System should automatically adjust its resource allocation to handle significant fluctuations in traffic, maintaining performance without manual intervention. | |
| Environment | During a period of rapidly increasing user traffic, such as a marketing event or flash sale. | |
| Stimulus | The incoming request rate for key services increases by N% over a 15-minute period. | |
| Response | 1. The cloud monitoring infrastructure component of the system detects that there is increase in key service metrics (e.g. request count) has increased by N% over a 15-minute period. 2. Cloud monitoring infrastructure detects that a key metric (e.g., CPU utilization) has breached the pre-defined auto-scaling threshold. 3. The orchestrator (e.g., Kubernetes HPA or an Auto-Scaling Group) triggers a scale-out event. 4. System provisions one or more new service instances (pods/containers). 5. The new instances start, pass their mandatory health checks, and register with the load balancer. 6. The load balancer begins routing live traffic to the new, healthy instances. | |
| Measure | [Time to Scale] = [time when new provisioned infra is able to accept traffic] - [time when system detects the increase by N% over a 15-minute period] | |
| Allowance | [Time to Scale] <= N seconds | |

|  |  |  |
| --- | --- | --- |
| **NFR\_04** | **Reliability** | System Recovery |
| Description | System should be able to automatically recover from a critical component failure and restore full service functionality without requiring manual intervention | |
| Environment | A critical, stateless service instance crashes, or a primary node of a stateful service (like a database) fails. | |
| Stimulus | A critical, stateless service instance (e.g., ListingService) crashes or fails its health check. | |
| Response | 1. The system's orchestrator (e.g., Kubernetes) detects the instance is unhealthy and terminates it.  2. The load balancer immediately stops routing traffic to the failing instance.  3. The orchestrator's replication controller detects the replica count is below the desired state.  4. The orchestrator immediately schedules a replacement instance.  5. A new, healthy instance is started, passes its health checks, and begins accepting traffic, restoring full service capacity. | |
| Measure | [Time to recover] = [time when a new instance begins accepting traffic] – [time when failure is detected] | |
| Allowance | [Time to recover] <= 'N' minutes | |

## Quality Attributes

// A5. Quality Requirement Specification

// C5-2. Is the specification of quality requirements appropriate?

// C5-3. Is quality requirement measurable?

|  |  |  |
| --- | --- | --- |
| **QA\_01** | **Usability** | Search Result Relevance |
| Description | This attribute measures the effectiveness of the search algorithm by quantifying how often the displayed results are relevant enough to warrant user engagement. | |
| Environment | In the live production environment where users are actively searching for items. | |
| Stimulus | A user enters a search query and system returns an ordered list of results. | |
| Response | The user reviews the list, finds the top results relevant to their intent, and clicks on a result to view the item's detail page.   1. System displays the paginated list of search results to the user. 2. The user visually scans the *first page* of results. 3. The user identifies a relevant item within the top N (e.g., 5) results. 4. The user clicks on that item to view its detail page. | |
| Measure | [Search Result Click-Through Rate] (CTR) = (Total Searches that Result in a Click) / (Total Number of Searches Performed) | |

|  |  |  |
| --- | --- | --- |
| **QA\_02** | **Usability** | Recommendation Usefulness |
| Description | This attribute measures the effectiveness of the recommendation system by quantifying user engagement with the items it suggests. | |
| Environment | In the live production environment with a diverse set of real users. | |
| Stimulus | A user is shown a carousel of recommended items on the home page or a product detail page. | |
| Response | The user finds the recommendations relevant and interesting, leading them to click on one or more of the suggested items to learn more.   1. System renders the "Recommended For You" carousel on the home page. 2. The user scrolls or visually scans the items in the carousel. 3. The user identifies a compelling item in the list. 4. The user clicks on the recommended item to view its detail page. | |
| Measure | [Recommendation Click-Through Rate] (CTR) = (Total Clicks on Recommended Items) / (Total Impressions of Recommended Items) | |

|  |  |  |
| --- | --- | --- |
| **QA\_03** | **Security** | Personal Information & Off-platform Solicitation Protection |
| Description | This security attribute ensures that users are proactively protected from frauds and privacy violations within the chat feature by automatically filtering prohibited content. | |
| Environment | During a real-time chat session between any two users. | |
| Stimulus | A user attempts to send a chat message containing sensitive information (e.g., a phone number) or a link to an external payment site. | |
| Response | The system's content moderation service analyzes the message in real-time, identifies the policy violation, blocks the message from being delivered, and notifies the sender. | |
| Measure | [Detection Accuracy] = (Correctly Identified Violations) / (Total Actual Violations)  [False Positive Rate] = (Incorrectly Blocked Legitimate Messages) / (Total Legitimate Messages Analyzed) | |

|  |  |  |
| --- | --- | --- |
| **QA\_04** | **Security** | Listing Moderation & Malicious Content Prevention |
| Description | This trust and safety attribute ensures that all new and updated listings are automatically scanned for prohibited or malicious content before they are made visible to buyers. | |
| Environment | During the listing creation or editing workflow. | |
| Stimulus | A seller submits a new listing that includes policy-violating text (e.g., prohibited items) or images (e.g., unsafe content). | |
| Response | The automated moderation service analyzes the listing content, detects the violation, rejects the submission, and provides specific feedback to the seller on how to correct the issue. | |
| Measure | [False Positive Rate] = (Incorrectly Blocked Legitimate Listings) / (Total Legitimate Listings Submitted) | |

|  |  |  |
| --- | --- | --- |
| **QA\_05** | **Performance** | Listing Visibility Latency |
| Description | This attribute defines the maximum acceptable delay from when a seller creates or updates a listing to when that change is reflected and visible in user-facing systems like search and recommendations. | |
| Environment | During normal system operations. | |
| Stimulus | A seller successfully publishes a new product listing or saves an update to an existing one. | |
| Response | The system persists the change to the primary database and successfully propagates the update to the search index and recommendation data stores, making the listing discoverable by buyers. | |
| Measure | [Latency] = [t\_visible\_in\_search] – [t\_published/t\_edited] | |

|  |  |  |
| --- | --- | --- |
| **QA\_06** | **Maintainability** | Adaptability for New Recommendation Algorithms |
| Description | This attribute defines the system's modifiability, ensuring that new recommendation or search algorithms can be integrated & deployed with minimal engineering effort and without requiring major architectural changes. | |
| Environment | During a planned development cycle in a staging/testing environment. | |
| Stimulus | The data science team provides a new, trained recommendation model that needs to be integrated into the production system. | |
| Response | An engineer integrates the new model by implementing a pre-defined service interface, allowing it to be deployed alongside the existing model for A/B testing without altering the core services that request recommendations. | |
| Measure | [Engineering Effort] = The total person-weeks required to integrate, test, and deploy a new recommendation algorithm. | |

|  |  |  |
| --- | --- | --- |
| **QA\_07** | **Security** | Real-time Seller Risk Scoring |
| Description | This fraud prevention attribute defines the system's ability to assess the risk level of a seller in real-time based on their actions, in order to flag potentially fraudulent activity early. | |
| Environment | A seller performs a significant action, such as creating an account or listing a high-value item. | |
| Stimulus | A new seller with no transaction history lists a high-demand electronic item at a price significantly below the market average. | |
| Response | The real-time risk engine processes signals associated with the seller and the listing, calculates a high-risk score, and automatically flags the listing for mandatory manual review before it is published. | |
| Measure | [Risk Scoring Latency] = [t\_score\_available] – [t\_trigger\_event] | |

|  |  |  |
| --- | --- | --- |
| **QA\_08** | **Response Time** | Listing Media Processing Time |
| Description |  | |
| Environment |  | |
| Stimulus |  | |
| Response |  | |
| Measure |  | |

|  |  |  |
| --- | --- | --- |
| **QA\_09** | **Performance** | Chat Message Delivery Time |
| Description | This attribute defines the end-to-end latency for a message sent between two users. A low latency is critical to ensure the chat feels conversational and real-time. | |
| Environment | During a chat session between two online users under normal network conditions. | |
| Stimulus | A user sends a message to another user. | |
| Response | The system receives the message from the sender, processes it, and delivers it to the recipient's client application. | |
| Measure | [Delivery Latency] = [t\_received] – [t\_sent]. | |

|  |  |  |
| --- | --- | --- |
| **QA\_10** | **Reliability** | Monitoring and Alerting |
| Description | This attribute ensures that key health metrics of all production micro services are actively monitored and that automated alerts are triggered when anomalies are detected. | |
| Environment | The system is running in the 24/7 production cloud environment. | |
| Stimulus | A critical backend service (e.g., Order Service) experiences a sustained spike in its server-side error rate (HTTP 5xx errors) exceeding a predefined threshold. | |
| Response | The monitoring system detects the threshold breach and automatically generates and sends a high-priority alert to the designated on-call engineering channel | |
| Measure | [Detection-to-Alert Latency] = [t\_alert\_sent] – [t\_threshold\_breached] | |

# Architecture

// A8. Architecture Documentation

// C8-1. Is allocation of processes, etc. appropriate? (deployment)

// C8-2. Is grouping appropriate in terms of components? (component & connector)

// C8-3. Is the description of System architecture appropriate?

# Modules

// A9. Module Specification

// C9-1. Is component specification sufficient to develop?

// C9-2. Is grouping appropriate in terms of module?

// C9-3. Is it appropriate to design dependencies between modules?

// C9-4. Is the work assignment appropriate?

Appendix

[A. Domain Model 17](#_Toc516321204)

[B. Quality Scenarios 18](#_Toc516321205)

[C. Quality Scenario Analysis 19](#_Toc516321206)

[D. Candidate Architectures 20](#_Toc516321207)

[E. Candidate Architecture Evaluation 21](#_Toc516321208)

[F. Architecture Design 22](#_Toc516321209)

[G. Architecture Evaluation(ATAM) 23](#_Toc516321210)

1. Domain Model

// A3. Domain Model Design

// C3-1. Is domain model sufficiently sub-divided?

// C3-2. Does domain model reflect architecture decisions?

**Entity-Control-Boundary** pattern is used to describe the conceptual model of C2C Market Place.

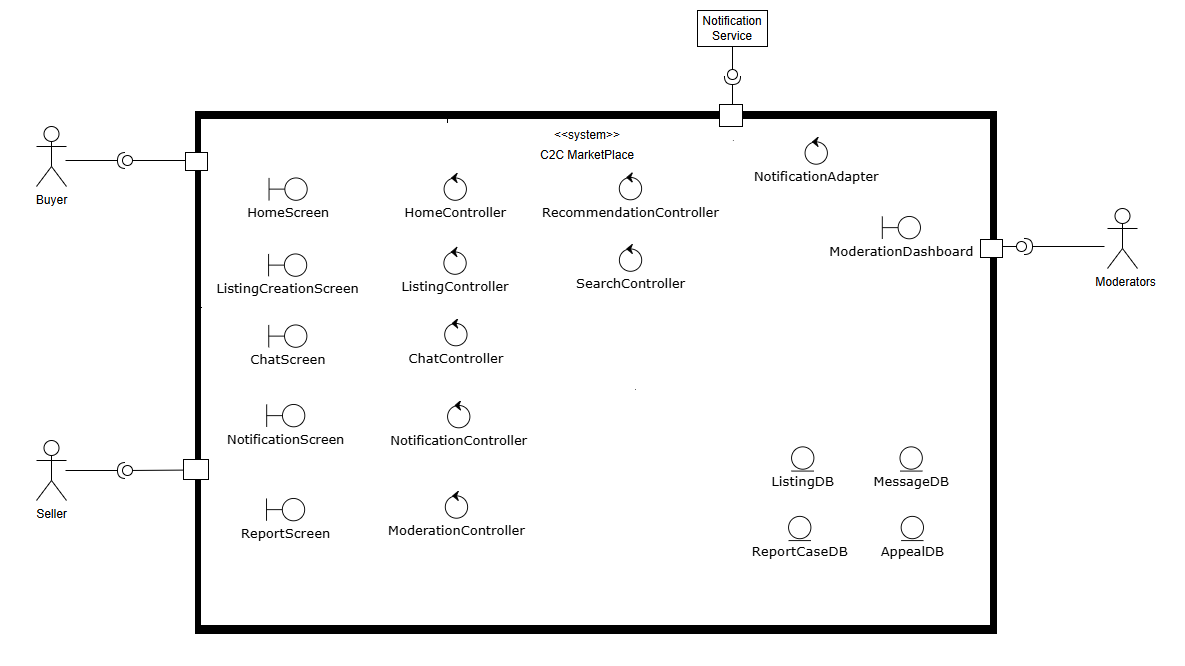


Figure 10: Domain Model for C2C Market Place

### Boundaries Components: These are the interfaces that users interact with directly.

* **HomeScreen**: The main landing screen that displays content and includes search and recommendation features.
* **ListingCreationScreen**: The UI where a Seller creates, previews, and manages their product listings.
* **ChatScreen**: The interface for sending and receiving messages and offers between users.
* **NotificationScreen**: The interface for managing notification preferences.
* **ReportScreen**: The UI used by any user to report suspicious content or other users.
* **ModerationDashboard**: A specialized UI for Moderators to review and act on reported cases and appeals.

### Controllers Components: These are the components containing the core business logic for each use case.

* **ListingController**: Manages the logic of creating, validating, and publishing a product listing.
* **SearchController**: Handles the logic of executing a search query and returning relevant results.
* **HomeController**: Orchestrates the assembly of the home page, including fetching recommendations.
* **RecommendationController**: Contains the logic for generating personalized recommendations for users.
* **ChatController**: Manages the business rules for messaging, offers, and finalizing agreements.
* **NotificationController**: Manages the logic for when and how to send notifications to users based on system events.
* **ModerationController**: Handles the workflow for automated and manual content moderation, including user reports and appeals.
* **NotificationAdapter**: A specialized controller that translates internal notification requests into the format required by external services.

### Entities: These are the core data objects that are persisted on the server.

* **ListingDB**: Represents a product or item that a Seller has put up for sale.
* **MessageDB**: Represents a single communication sent between a Buyer and a Seller.
* **ReportCaseDB**: An entity created to track a user-reported issue.
* **AppealDB**: Represents a user's request to have a moderation decision reviewed manually.

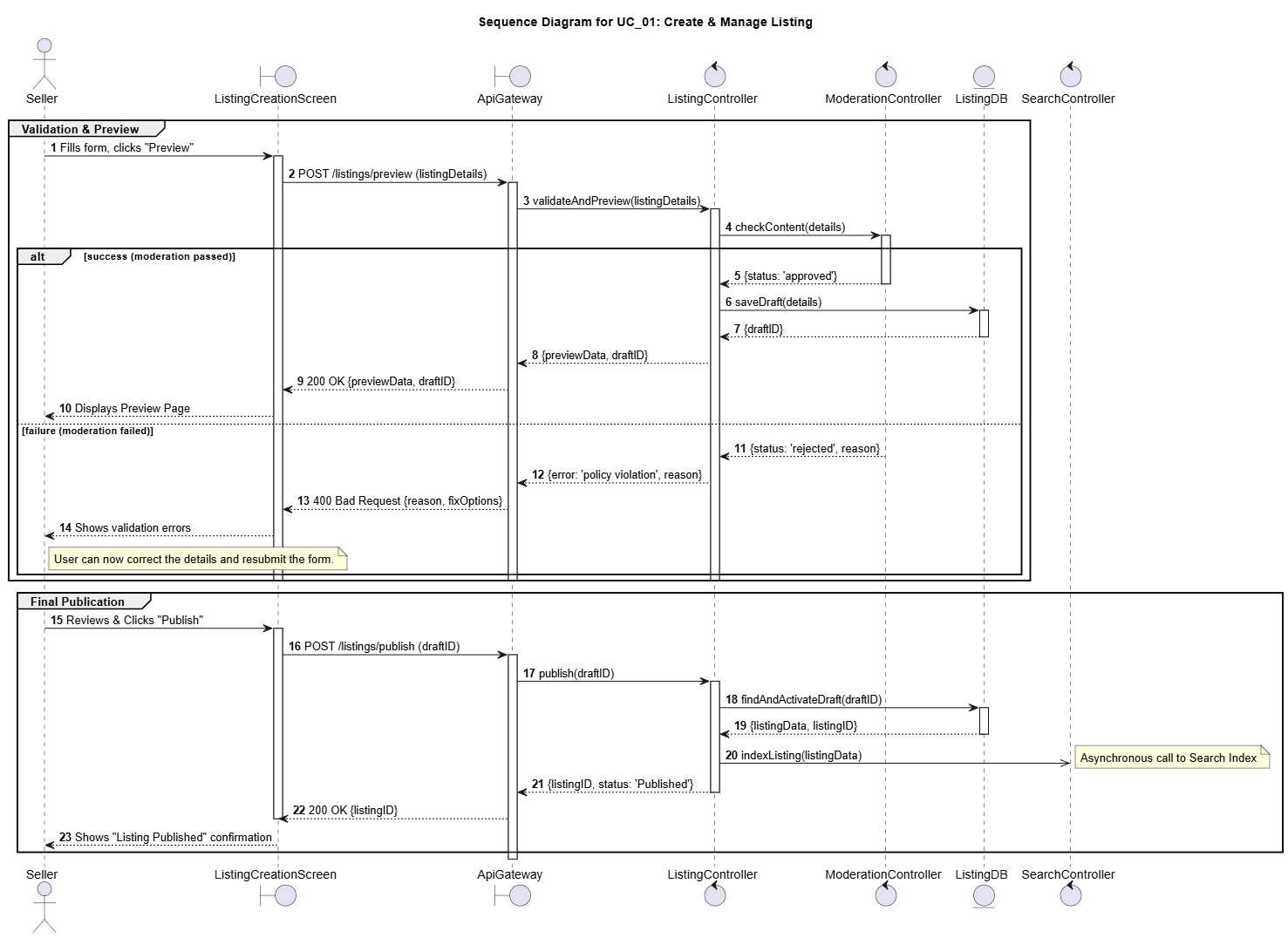


Figure 11: Sequence Diagram for UC01: Create & Manage Listing

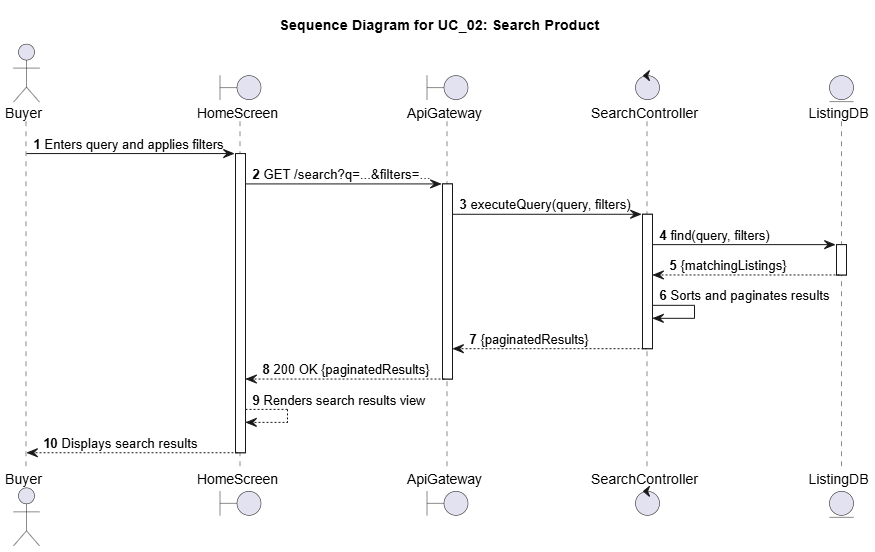


Figure 12: Sequence Diagram for UC02: Search Product

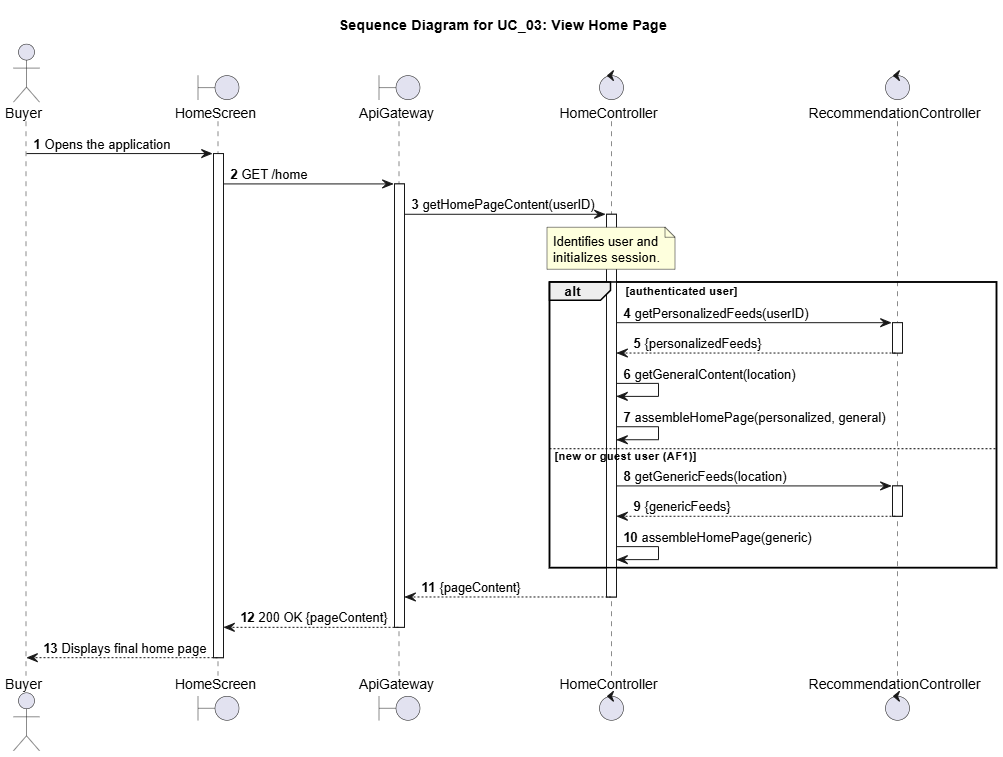


Figure 13: Sequence Diagram for UC03: View Home Page

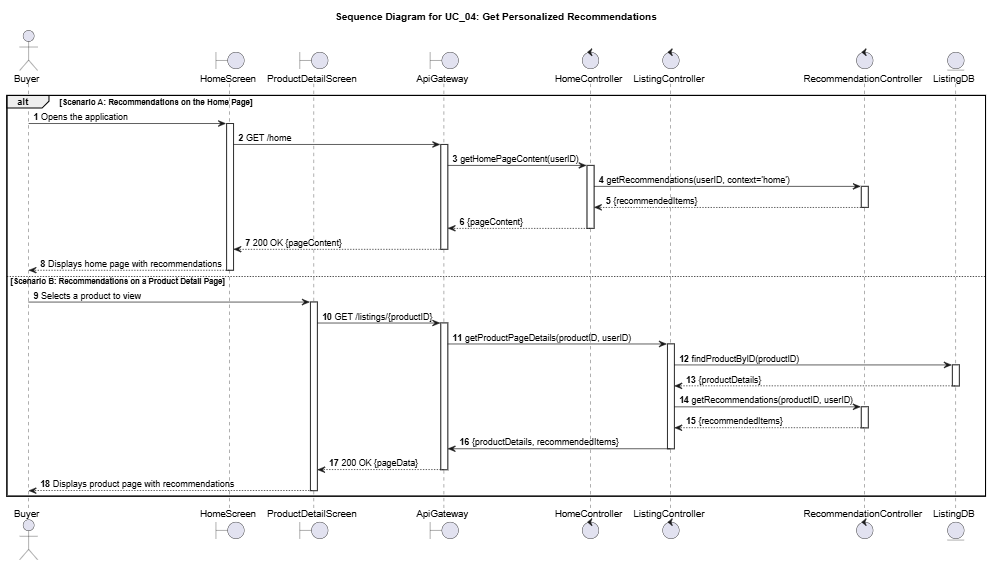


Figure 14: Sequence Diagram for UC04: Get Personalized Recommendations

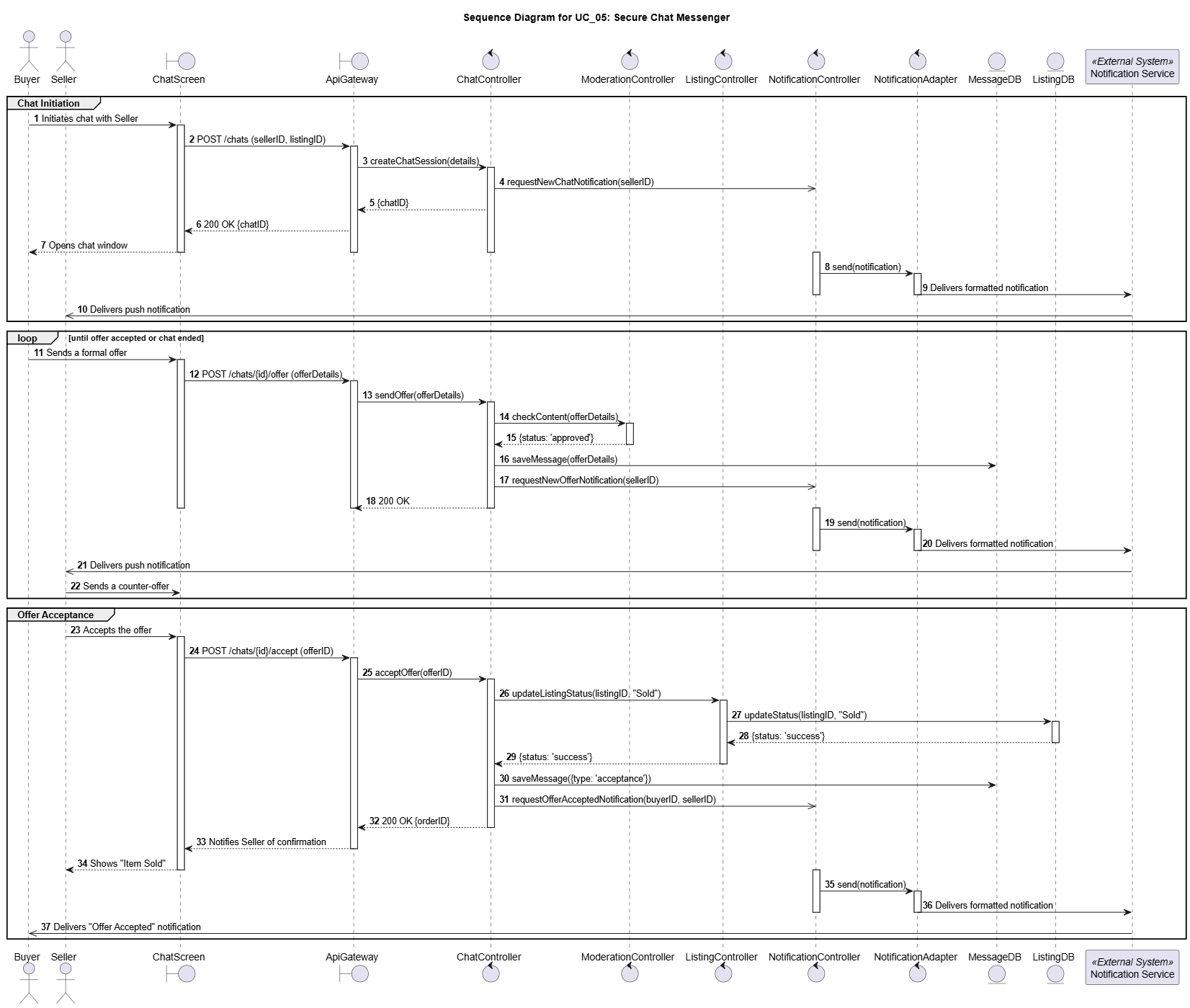


Figure 15: Sequence Diagram for UC05: Secure Chat Messenger

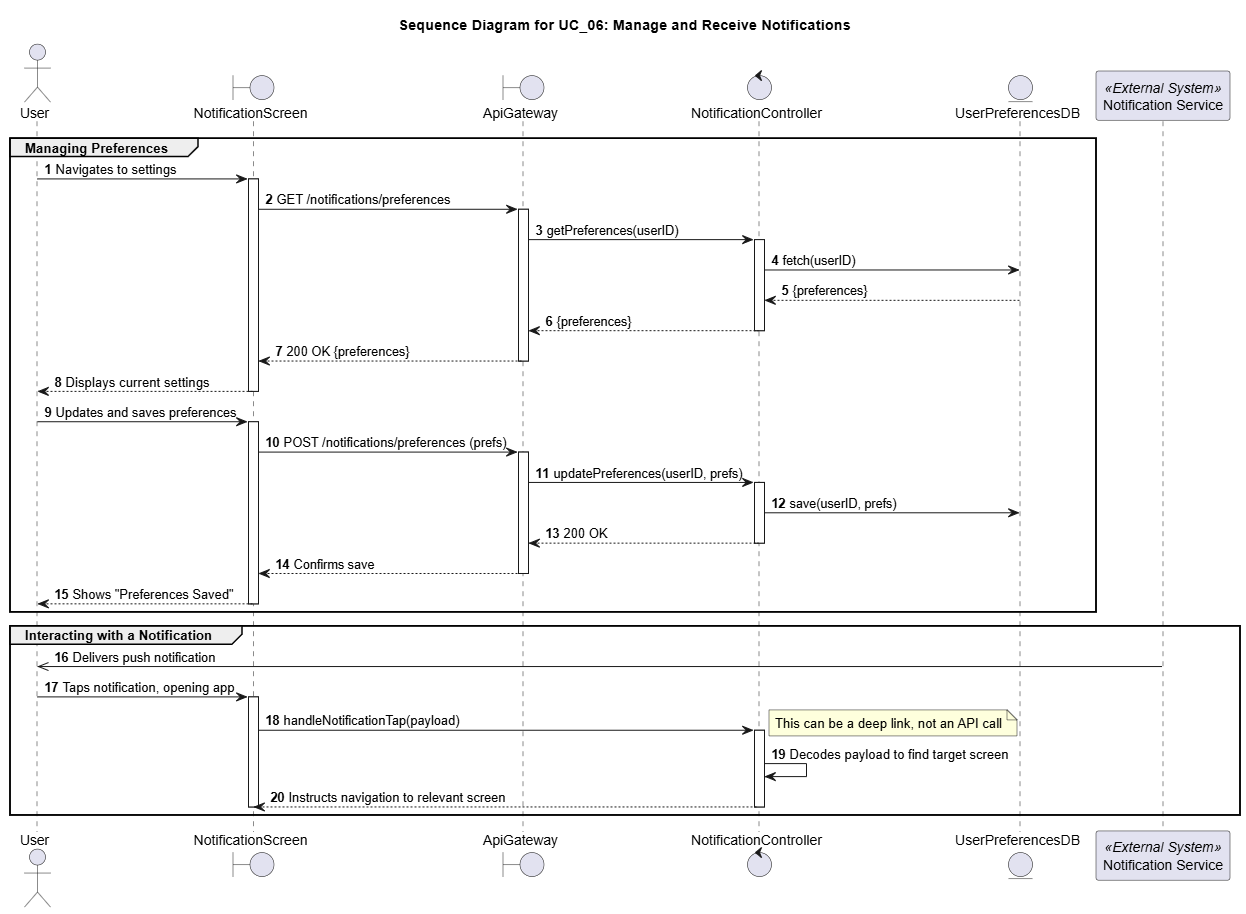


Figure 16: Sequence Diagram for UC06: Manage and Receive Notifications

1. Quality Scenarios

// A4. Quality Scenario Elicitation

// C4-1. Is there sufficient scenario elicitation affecting the architecture?

// C4-2. Is there sufficient review of the quality related to performance?

// C4-3. Is there sufficient review of the quality related to modifiability?

|  |  |  |
| --- | --- | --- |
| **Category** | **Sub Category** | **Quality Scenario** |
| **Performance** | **Response Time** | **QS-01: Search operation response time** |
| When a user searches for an item, the system must return results within a very short time.  Measure: [Response Time] = [t\_result\_returned] – [t\_query\_received] at API gateway. |
| **Response Time** | **QS-02: Time taken for a new (or updates in existing) listing to be visible to the user** |
| When a seller publishes a new listing or changes an existing one, it should be discoverable by search and recommendation systems almost instantly Measure: [Latency] = [t\_visible\_in\_search] – [t\_published/t\_edited]. |
| **Response Time** | **QS-03: Home Screen Loading time** |
| When a user opens the homescreen, the time taken to load all the data should be as low as possible Measure: [Recommendation Latency] = [t\_results\_returned] – [t\_request\_initiated]. |
| **Response Time** | **QS-04: Chat message delivery time** |
| Messages between users must be delivered with minimal delay to enable smooth communication and negotiation flows. Measure: Delivery Latency = [t\_received] – [t\_sent]. |
|  | **QS-05: Listing Media Processing Time** |
| add some description here Measure: [Media Ready Latency] = [t\_preview\_available] – [t\_upload\_initiated] |
| **Scalability** | **QS-06: Service should be able to scale up/down as per the load on service** |
| System is able to adjust itself as per the traffic trend there by ensuring that it does not affect the end user. Measure: [Average throughput] >= N thousand request per sec |
| **Reliability** | **Fault Tolerance** | **QS-07: Monitoring of Cloud Microservices** |
| System should be able to monitor key service and applications. Alarms should be generated in case of any abnormality Measure: [Error percentage of key service metric (like 5xx error rates) should be almost 0) |
| **Durability** | **QS-08: Messaging ordering & durability** |
| Chats must never be lost and message ordering should be guaranteed, even with offline/reconnect. Measure: [Message Loss Rate] |
| **Availability** | **QS-09: System should be able to recover from any error in system or Server Infrastructure** |
| System should be able to handle any error in system or infrastructure failure intelligently without affecting end user Measure: [Time to recover] <= 'N' minutes |
| **Maintainability** | **Modifiability** | **QS-10: System should be flexible to adopt new Recommendation algorithms** |
| System should be adaptable to easily change to new algorithms for better recommendations and search features Measure: [Development Effort in months to integrate new Algorithms] |
| **Modifiability** | **QS-11: System should be flexible to add new Category in item listing section** |
| Addition of new category in items listing section should be done quickly without impacting the existing data. Measure: [Development Effort in weeks for implementing new category] |
| **Cost Efficiency** | **QS-12: Cost efficiency & autoscale** |
| Autoscaling should minimize over provisioning of resources while sustaining performance. Measure: [Cost per Million requests] |
| **Security** | **Fraud Detection** | **QS-13: Personal Information & Off-platform Solicitation Protection** |
| Automatically detect and block sensitive information (phone numbers, emails) and off-platform payment solicitations (e.g., payment links) in chat to reduce fraud exposure and keep transactions secure. Measure: [Detection Accuracy] = correctly identified violations / total violations. [False Positive Rate] = incorrect blocks / total detections. |
| Expectation: Detection accuracy ≥ 98%, false positive rate < 2%, Solicitation detection rate ≥ 90% |
| **Content Moderation** | **QS-14: Listing Moderation & Malicious Content Prevention** |
| Automatically prevent prohibited, unsafe, or malicious content at listing creation using text/image moderation, ensuring policy compliance and protecting buyers while minimizing false positives for legitimate sellers. Measure: [False Positive Rate] = incorrect blocks / total detections. |
| **Authentication** | **QS-15: System should prevent unauthorised access to MicroService** |
| System should implement mechanism to prevent unauthorised access to APIs Measure: [Number of Violations]/[total Count] |
| **Fraud Detection** | **QS-16: System should be able to flag high-risk sellers in real-time to prevent abuse** |
| High-risk sellers must be flagged early to prevent abuse. Listings from low-risk sellers should be promoted Measure: [Review Latency] |
| **Usability** | **Appropriateness** | **QS-17: Recommendation should be useful to end users** |
| Recommendation provided by our system should be beneficial for end user. In order to quantify this, we should have an A/B testing mechanism around recommended listings for actual user acceptance. Measure : [Click-Through-Rate] |
| **Appropriateness** | **QS-18: Search Result should be relevant** |
| System should display search results in order of relevance Recommended, most viewed etc. Measure : [Search Click-through-Rate] |

1. Quality Scenario Analysis

// A5. Quality Requirement Specification

// C5-1. Is quality scenario analysis appropriate? (Evidence)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Category** | **Sub Category** | **Quality Scenario** | **Importance** | **Difficulty** | **Type** |
| **Performance** | **Response Time** | QS-01: Search operation response time should be low | H | H | NFR\_01 |
| **Response Time** | QS-02: Time taken for a new (or updates in existing) listing to be visible to the user should be low | H | M | QA\_05 |
| **Response Time** | QS-03: Home Screen Loading time should be low | H | M | NFR\_02 |
| **Response Time** | QS-04: Users should be able to get chat message quickly | M | M | QA\_09 |
| **Response Time** | QS-05: System should be able to process Media Content in new Listing quickly | M | M | QA\_08 |
| **Scalability** | QS-06: System should be able to scale the servers up/down as per the load on the service automatically | H | M | NFR\_03 |
| **Reliability** | **Fault Tolerance** | QS-07: System should have a mechanism of monitoring all the Micro services | M | M | QA\_10 |
| **Durability** | QS-08: System should ensure chat messaging ordering & durability | L | L | NA |
| **Availability** | QS-09: System should be able to recover from any error in system or Server Infrastructure | H | M | NFR\_04 |
| **Maintainability** | **Modifiability** | QS-10: System should be flexible to adopt new Recommendation algorithms | M | H | QA\_06 |
| **Modifiability** | QS-11: System should be flexible to add new Category in item listing section | L | L | NA |
| **Cost Efficiency** | QS-12: System should be Cost efficient | M | L | NA |
| **Security** | **Fraud Detection** | QS-13: System must block the sharing of contact information and any off the platform transaction. | H | M | QA\_03 |
| **Content Moderation** | QS-14: System should moderate all listings to prevent malicious or prohibited content from appearing on the platform | H | M | QA\_04 |
| **Authentication** | QS-15: System should prevent unauthorised access to MicroService | M | L | NA |
| **Fraud Detection** | QS-16: System should be able to flag high-risk sellers in real-time to prevent abuse | M | H | QA\_07 |
| **Usability** | **Appropriateness** | QS-17: Recommendation should be useful to users | H | M | QA\_02 |
| **Appropriateness** | QS-18: Search Result should be relevant | H | M | QA\_01 |

**QS-01: Search operation response time should be low**

This is the most critical performance metric as it directly supports the primary business driver of **Easy Discovery**. A slow search feature creates a frustrating user experience, undermining our goal of helping buyers find what they want quickly. Therefore, it is the **highest priority non-functional requirement** for the platform's success.

**QS-02: Time taken for a new (or updates in existing) listing to be visible to the user should be low**

This scenario is fundamental to the **Seamless Listing Experience**. Sellers expect their items to be discoverable immediately after publishing, and any delay breaks the "effortless" process and erodes seller trust. Minimizing this latency is a **high-priority architectural driver**.

**QS-03: Home Screen Loading time should be low**

The home screen is the main entry point for **Easy Discovery**, featuring personalized recommendations. A long loading time creates a poor first impression and can lead to user abandonment before they even begin browsing. Fast home screen performance is therefore a **non-negotiable, fundamental requirement** for the platform’s viability, hence making is **high-priority** NFR.

**QS-04: Users should be able to get chat message quickly**

Real-time communication is vital for negotiation and **Building Community Trust**. Delays in chat can make conversations feel disjointed and lead to frustration, but it follows the initial discovery and engagement phase. This makes ensuring low latency in chat an **important, medium-priority factor** for facilitating successful transactions.

**QS-05: System should be able to process Media Content in new Listing quickly**

This scenario is an important component of the **Seamless Listing Experience**. A slow or failing media upload process is a major friction point in the listing flow. Optimizing this is essential for a smooth seller journey, making it a **medium-priority quality attribute**.

**QS-06: System should be able to scale the servers up/down as per the load on the service automatically**

Scalability is a foundational requirement that underpins all business drivers. The system must maintain performance for all features during peak traffic, as a failure to scale would degrade the entire user experience. This is a **high-priority architectural concern** for long-term viability.

**QS-07: System should have a mechanism of monitoring all the Micro services**

Proactive monitoring is important for ensuring the platform remains reliable and trusted. However, it is an operational concern that supports the system rather than a direct, user-facing feature that drives the core architecture. It can be refined post-launch, which is why it is considered a **lower-priority quality attribute (QA\_11)**.

**QS-08: System should ensure chat messaging ordering & durability**

This scenario is critical for **Building Community Trust**. Lost or out-of-order messages during a negotiation can lead to significant confusion and broken deals, severely damaging user trust in the platform. Guaranteeing message integrity is therefore a **high-priority reliability requirement**.

**QS-09: System should be able to recover from any error in system or Server Infrastructure**

High availability is a fundamental pillar of a trustworthy platform. If the system cannot recover quickly from failures, users cannot discover items, list products, or communicate, eroding confidence across all business drivers. Therefore, automated and rapid system recovery is a **high-priority requirement**.

**QS-10: System should be flexible to adopt new Recommendation algorithms**

This attribute is key to the long-term success of the **Easy Discovery** driver. The ability to easily integrate new and improved recommendation models ensures the platform can adapt and improve the user experience over time. This makes it an **important, medium-priority attribute for future growth**.

**QS-11: System should be flexible to add new Category in item listing section**

While important for long-term business flexibility, adding new categories does not fundamentally influence the core real-time performance or reliability of the architecture. This functionality can be addressed with minor impact on the existing design. Therefore, **this scenario is considered a low-priority concern and can be dropped** as a primary architectural driver for now.

**QS-12: System should be Cost efficient**

Cost is a critical business metric, but optimizing it is a secondary operational goal compared to establishing core functionality and performance. The primary architecture must first ensure the system works reliably and scales effectively while cost can be tuned later. Therefore, **this scenario can be dropped** as a main architectural driver.

**QS-13: System must block the sharing of contact information and any off the platform transaction**

This is a cornerstone of **Building Community Trust**. The system must proactively protect users from fraud within chat to make the marketplace a safe environment for transactions. This is a **high-priority security requirement**.

**QS-14: System should moderate all listings to prevent malicious or prohibited content from appearing on the platform**

This scenario directly addresses the business driver of **Building Community Trust** by proactively detecting fraud. Automatically scanning new listings for policy violations is essential for protecting buyers and maintaining the platform's reputation, making this a **high-priority security feature**.

**QS-15: System should prevent unauthorized access to MicroService**

While preventing unauthorized inter-service communication is a fundamental security principle, its solution relies on standard industry patterns like API gateways and token-based authentication. This concern is addressed as a baseline implementation requirement rather than a unique architectural challenge that drives the overall design. Therefore, **this scenario is dropped as a primary architectural driver** from this analysis.

**QS-16: System should be able to flag high-risk sellers in real-time to prevent abuse**

This is a key preventative measure for **Building Community Trust** by flagging potential fraud before it impacts buyers. It is a complex feature that significantly enhances platform safety. It is therefore an **important, medium-priority architectural concern**.

**QS-17: Recommendation should be useful to end users**

This usability scenario is the ultimate measure of success for the **Easy Discovery** driver. A high click-through rate is direct evidence that the system is succeeding in helping users find items they want. This makes it a **vital, medium-priority metric** for guiding product development.

**QS-18: Search Result should be relevant**

This is the qualitative counterpart to search performance and is critical for **Easy Discovery**. A fast search that returns irrelevant results is useless. Ensuring the top results are highly relevant to the user's intent is **fundamental to the platform’s core value proposition**.

1. Candidate Architectures

// A6. Candidate Architecture Design

// C6-1. Are quality analysis and solution candidate appropriate?

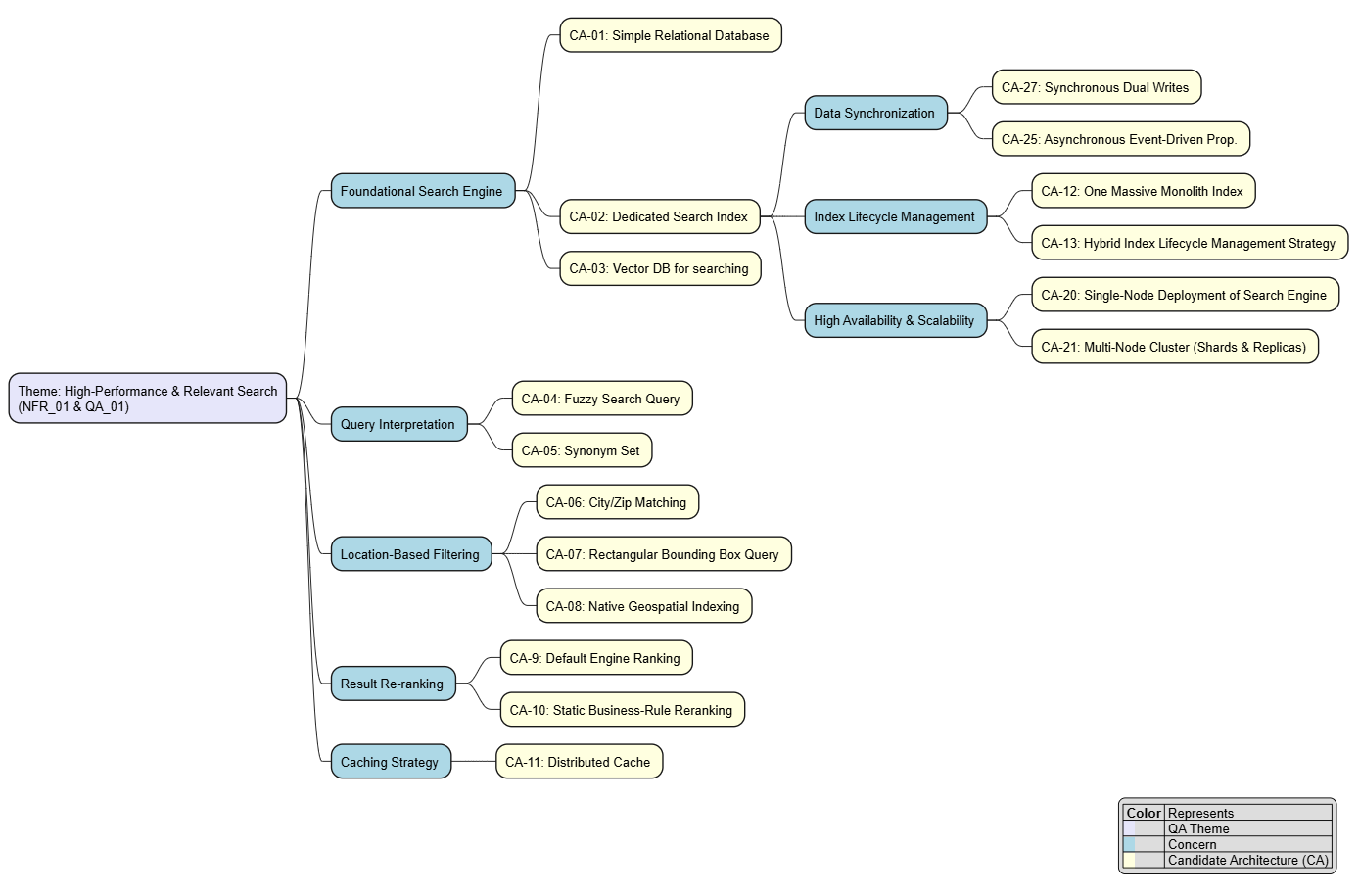
// C6-2. Are performance analysis and solution candidate appropriate?

// C6-3. Are modifiability analysis and solution candidate appropriate?

In this section, we will explore Candidate Architectures targeting each Quality Attribute and Non-Functional Requirement we described in Section 2.2 and 2.3

**Targeted QA/NFR:**

* NFR\_01 (**Performance**): Search operation response time. This requires that the time between the system receiving a query and returning the results be less than or equal to 'N' milliseconds.
* QA\_01 (**Usability**): Search Result should be relevant. This measures the effectiveness of the search algorithm by quantifying user engagement with the results, such as the click-through rate.



* 1. **Concern: Foundational Search Mechanism (Engine)**
  2. CA-01: Simple Relational Database

Enhances Search Response Time (NFR\_01)

* Direct DB query is the most basic approach of a search problem statement.
* We can design the SearchController to construct a SQL query with a LIKE '%keyword%' clause and run it directly against the primary ListingDB (likely a relational database like PostgreSQL or MySQL).
* We can enhance this simple LIKE search by using the **built-in Full-Text Search (FTS)** capabilities of the relational database (e.g., PostgreSQL's tsvector).
* Doing this provides a **significant relevance improvement** by supporting features like **language-specific stemming** and **basic ranking.**
  1. CA-02: Dedicated Search Index Service (e.g., Elasticsearch or OpenSearch)

Enhances Search Response Time (NFR\_01) & Search Result should be relevant (QA\_01)

* This candidate architecture is a pattern that introduces a specialized search engine (e.g., Elasticsearch or OpenSearch) which operates separately from the primary database. Primary database holds all the listings that were created from the start of the system where as search Engine DB only stores (after indexing) only those entries that are live right now.
* This design is implemented by creating two distinct data flows. First, the listing are saved to a primary database, and then **asynchronously replicated** to the search engine. Second, when the user makes a search, the SearchUI calls the Search Service, the application's SearchController directs all user search queries exclusively to this dedicated engine, bypassing the primary database for all search and retrieval operations.

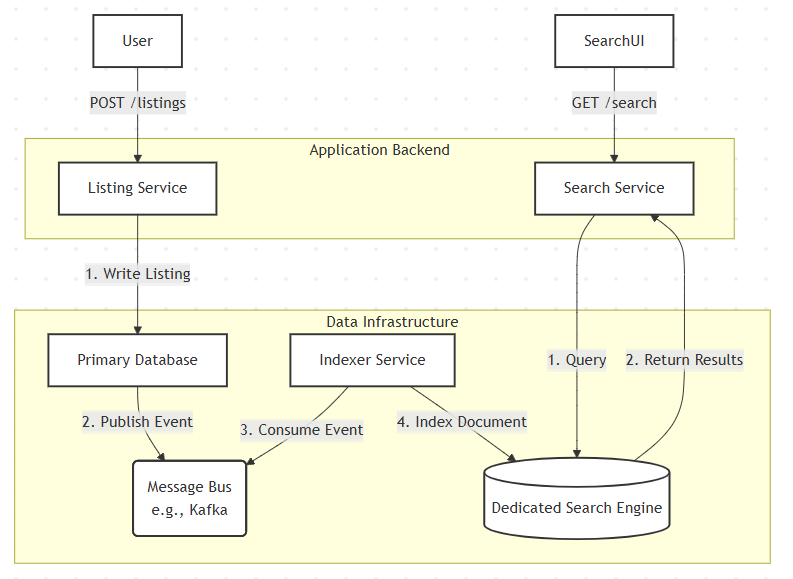


Figure: Search Index Service Data Flow

* 1. CA-03: Vector DB for searching

Enhances Search Response Time (NFR\_01) & Search Result should be relevant (QA\_01)

* We can use a **Vector Database** by converting product listings and user queries into **numerical vector embedding** through a dedicated **embedding model**.
* The database then finds the mathematically 'closest' items, enabling a powerful **semantic search** that matches by contextual meaning rather than exact keywords
* Doing this provides **state-of-the-art relevance** for conceptual queries (**QA\_01**) but is an **incomplete core solution**, as it cannot perform the **essential keyword searches** for specific models or handle the **complex filtering** required by the marketplace.

**2. Concern: Query Interpretation (Typos & Synonyms)**

* 1. CA-04: Fuzzy Search Query
* Adopting fuzzy search logic to propose auto correction in search input text.
* System will provide most suitable match for user input by using minimum edit distance by swapping or replacing char in string. (“mobyle” 🡪 “mobile”)

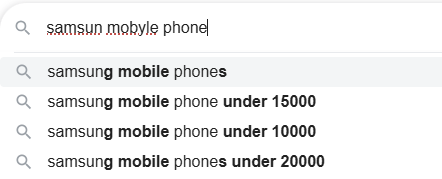


Figure: Fuzzy Search demonstration

* 1. CA-05: Synonym Set
* Users can use different terms to refer to same item (like “TV” vs “television” or “cupboard” vs “cabinet” vs “wardrobe” vs “almirah”)
* We can use Synonym Set so that system can expand the user's query to include a set of equivalent term using synonym Sets making the results more relevant.

1. **Concern: How to add location based Filtering?**
   1. CA-06: City/Zip Matching Only

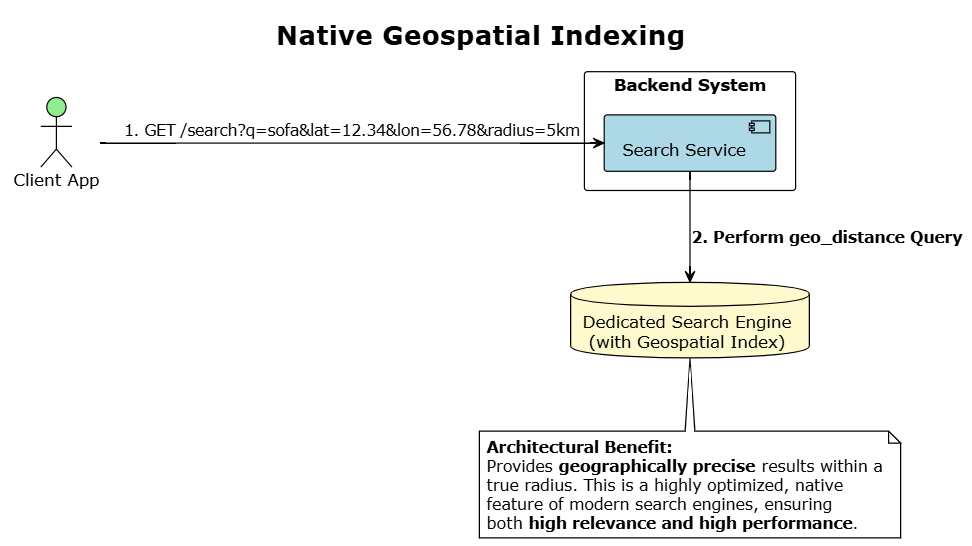
Enhances Search Result should be relevant (QA\_01)

* We can filter by city/ZIP which is provided by the user along with the search string.
* This approach will help return results from the same geographical location, making the results more relevant.
  1. CA-07: Geometric Proximity Filter

Enhances Search Result should be relevant (QA\_01)

We can implement a solution the system calculates a simple rectangular "bounding box" of latitude and longitude coordinates around the user's location. The database query then uses standard range filters (WHERE lat BETWEEN x AND y AND lon BETWEEN a AND b) to find items within this box.

* 1. CA-08: Native Geospatial Indexing
* We can leverage **Native Geospatial Indexing** by using a specialized **geo\_point data type** for storing location coordinates within our dedicated search engine (CA-02).
* The search is then performed using a highly optimized **geo\_distance query**, which accurately finds all items within a true circular radius. Doing this provides the **most accurate and high-performance solution** for 'near me' searches, best satisfying both **QA\_01** and **NFR\_01**.



1. **Concern: How to make search results more relevant (Re-rank)?**
   1. CA-09: Default Engine Ranking (Baseline)

* We can implement **Default Engine Ranking** by **relying exclusively on the default relevance score** provided by our selected search engine (CA-02). No further re-ordering or business logic is applied after the initial results are retrieved.
* While this is the **simplest approach** with no extra development, doing this means the ranking **lacks any business context**. It fails to prioritize listings based on critical factors like **seller rating or proximity**, providing a generic, one-size-fits-all experience.
  1. CA-10: Static Business-Rule Re-ranking
* We can implement **Static Business-Rule Re-ranking** by having a service apply a set of **pre-defined, static rules** to the search results *after* they are retrieved. This involves modifying the relevance score based on business logic, such as applying a **score boost** to listings from highly-rated sellers
* Doing this allows us to directly **inject critical business logic** into the search results, significantly improving relevance (**QA\_01**). The main trade-off is that these **rules are static** and may require **manual tuning** to adapt to changing user behavior.

1. **Concern: How to handle repeated requests?**
   1. CA-11: Distributed Cache for Search Results

Enhances Search Response Time (NFR\_01)

* We can cache the ListingId to the listing meta data object. Since this will be a very frequently used information, it will reduce load on the database.
* We can also cache the multiple data that is returned on homepage page loading like recommendation by user location, location related to user locations, user profile based recommendations (prepopulated and remain same for 24 hours).

1. **Concern: Search Data Retention Strategy**
   1. CA-12: One Massive Monolith Index

* We can implement a **Single, Ever-Growing Index** by keeping **all listings**—new, old, active, and sold—together in **one single, massive search index**. As new listings are created, they are simply added to this index, which grows larger over time.
* This is the **simplest approach** with no extra logic, but it forces every search to query an **increasingly large dataset** of mostly irrelevant historical data. Doing this will inevitably cause **performance to degrade over time**, creating a significant future risk to **NFR\_01**.
  1. CA-13: Hybrid Index Lifecycle Management Strategy
* We can implement a **Hybrid Lifecycle Strategy** that uses two distinct mechanisms for managing the index. For **immediate data correctness**, a ListingSold or ListingDeleted event will trigger a service to issue a **document-level delete command** to the active search index, removing the item from search results in near real-time.
* For **long-term performance and cost management**, we can use **Time-Based Indices with an Index Lifecycle Management Policy**. This policy will **automatically handle the bulk deletion** of entire old indices (e.g., deleting my-index-YYYY.MM.DD index after ‘N’ days), which is far more efficient than deleting millions of individual documents. Any Listing that was created will be live only for ‘N’ days, and if it is not sold within ‘N’ days, it will automatically be deleted by this rule. This will ensure our system is not loaded with irrelevant listings.

**Targeted QA/NFR:**

* NFR\_02 (**Performance**): Home Screen Loading time. This is the maximum time allowed for the backend to deliver all data required to render the home screen, which must be less than or equal to 'N' milliseconds.
* QA\_02 (**Usability**): Recommendation should be useful to end users. This measures the effectiveness of the recommendation system by quantifying user engagement (e.g., Click-Through Rate) with the suggested items.

The home page is the primary surface for passive discovery and must load almost instantly to retain user engagement. This is challenging as it requires aggregating multiple, personalized data feeds. The architecture must not only assemble this page quickly (NFR\_02) but also ensure the recommendations provided are compelling and useful enough to drive user clicks.

Below is the figure of the major components we need to populate the home screen.

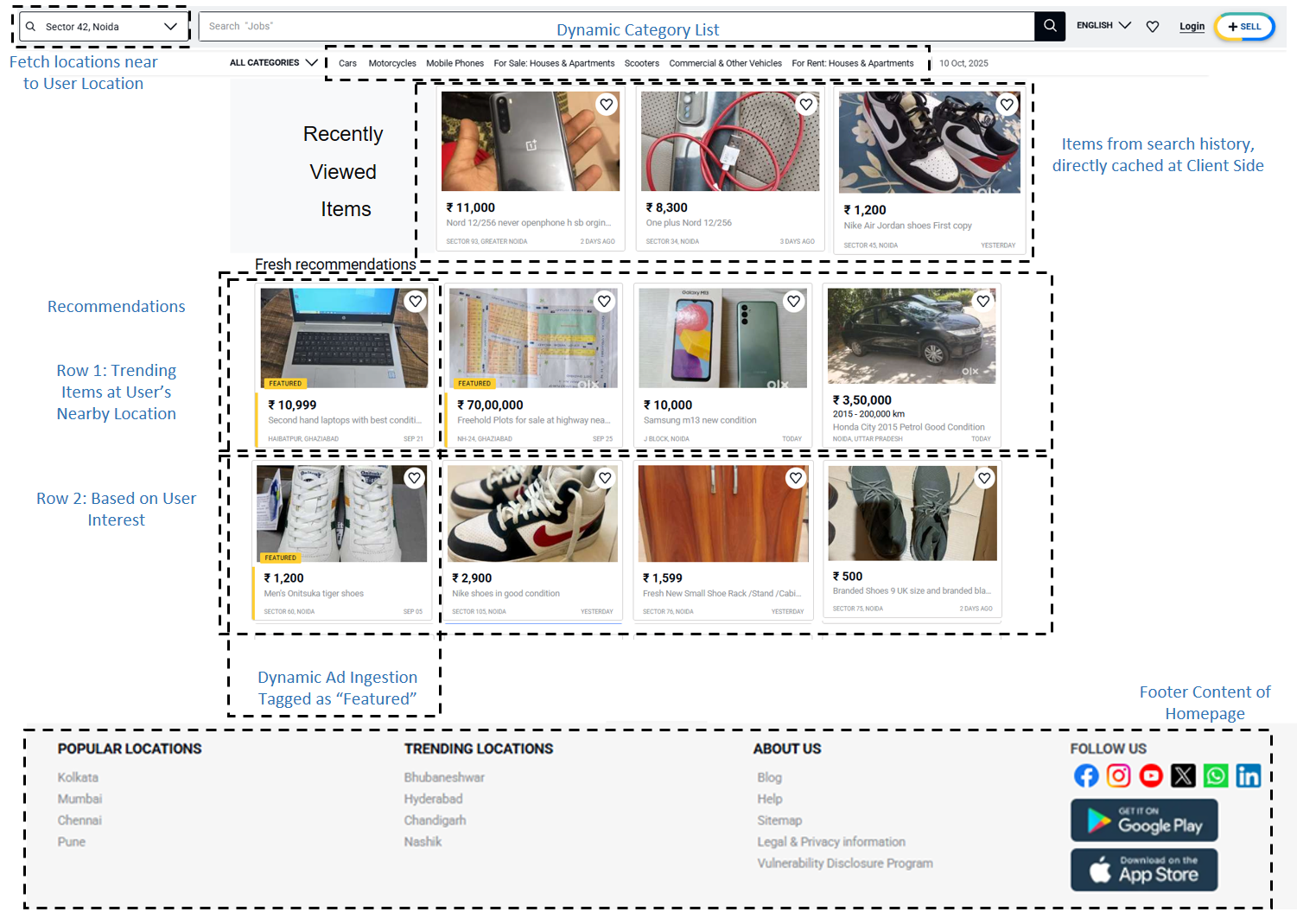
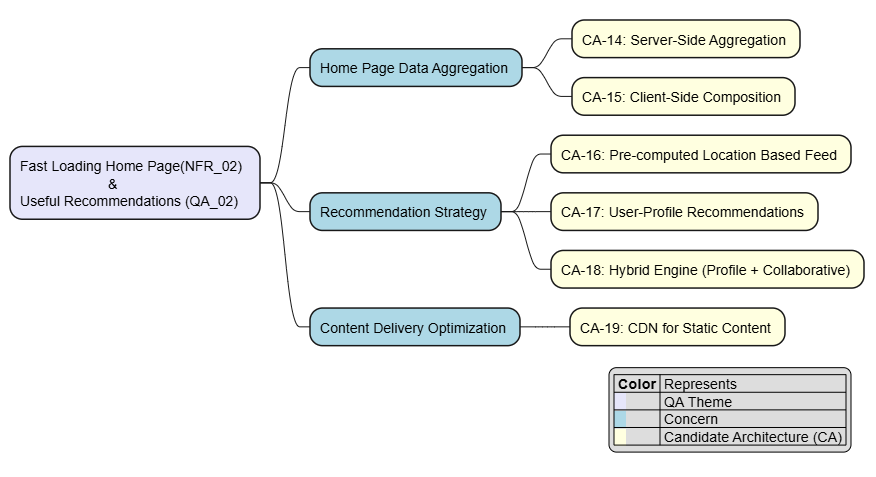
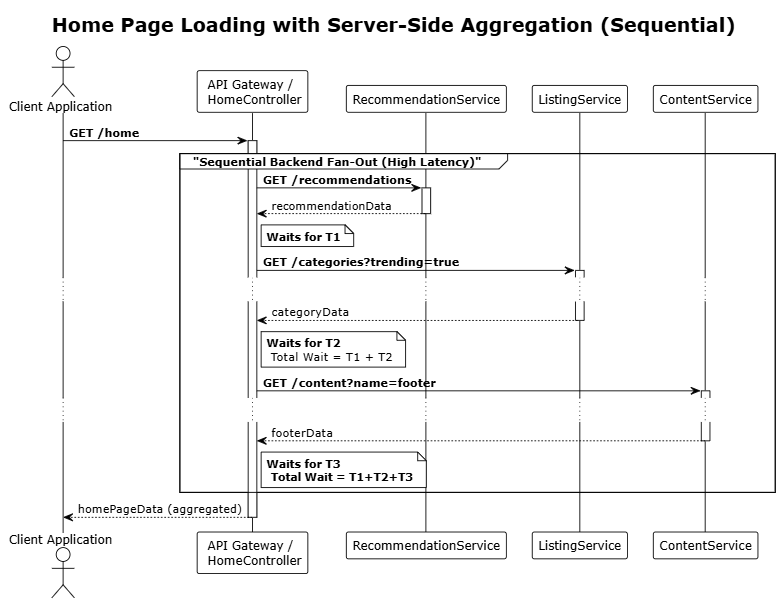


Figure: Home Screen Core Features



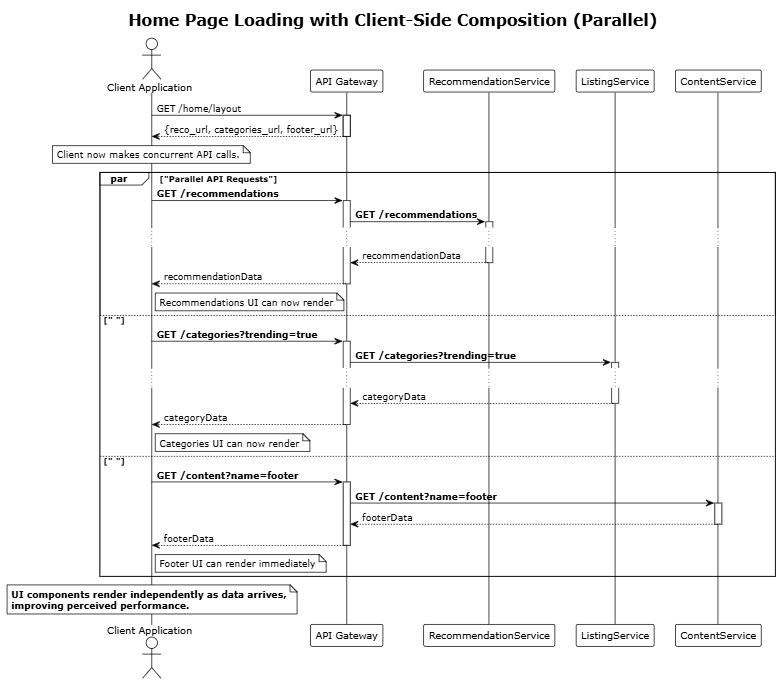
#### **7. Concern: Home Page Data Aggregation Pattern**

* 1. CA-14: Server-Side Aggregation Pattern
* The client makes a single request to an API Gateway. The client makes a single GET /home request. The HomeController then makes **blocking (sequential) calls to downstream micro services (**the RecommendationController, the LocationController, CategoryController (inside ListingService), FooterController (inside ContentService) etc.).
* It waits for all responses, aggregates them into one large JSON payload, and returns it.



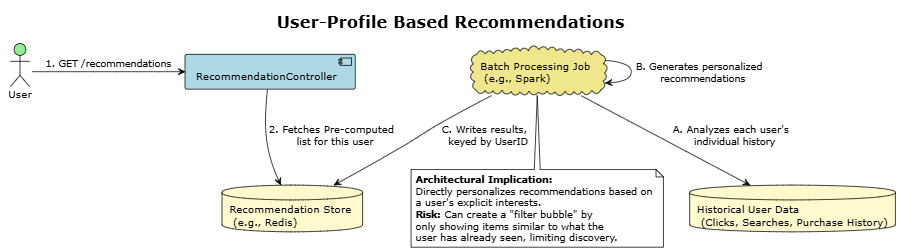
7.2 CA-15: Asynchronous Client-Side Composition

* The client application is responsible for orchestrating the home screen. It first fetches a minimal layout, then makes multiple, independent, and asynchronous calls to different endpoints (e.g., /recommendations, /categories). Each UI component populates itself as its data arrives.

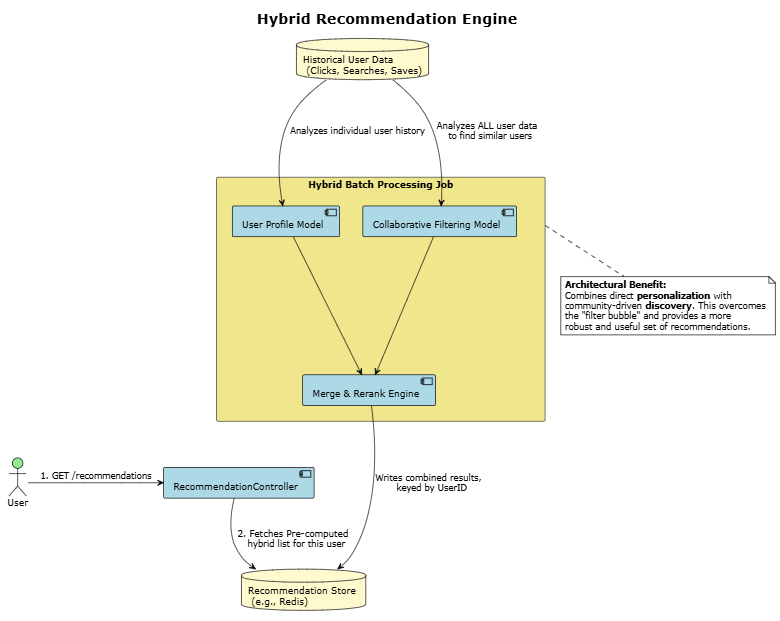


#### **Concern 8: Recommendation Generation Strategy**

* 1. CA-16: Pre-computed Feed Service based on Locations Only
* We can have a dedicated background service which pre-assembles home page feeds based on user location and stores the result in a fast key-value store (like Redis) for multiple locations. When the user opens the app, the HomeController makes a single, ultra-fast call to retrieve this pre-computed feed. This offers the lowest possible latency at the cost of the data being a few minutes out of date (eventual consistency).
  1. CA-17: User-Profile Based Recommendations
* We can implement **User-Profile Based Recommendations** by having an **offline batch job** analyze a **single user's historical interactions** (clicks, searches, saves). The output of this job is a **pre-computed list of recommended items** tailored specifically to that user's explicit interests
* Doing this **directly personalizes** the experience for each user, which is a strong driver for QA\_02 (Recommendation Usefulness). However, its primary architectural limitation is that it **struggles with discovery**, as it tends to recommend items very similar to what the user has already seen.



* 1. CA-18: Hybrid Recommendation Engine (User Profile + Collaborative)
* We can implement a **Hybrid Recommendation Engine** by **combining two distinct models** in an offline batch job. The first model generates a **personalized list** based on the individual user's profile and historical interactions (as in CA-17), while the second model uses **collaborative filtering** to generate a **discovery-oriented list** based on the behavior of similar users.
* Doing this **overcomes the limitations of each individual approach**. It augments the user's personal taste with **serendipitous discoveries** from the broader community, producing a **more robust and useful set of recommendations** that is designed to maximize QA\_02 (Recommendation Usefulness).



**9. Concern: Content Delivery Optimization to improve loading time**

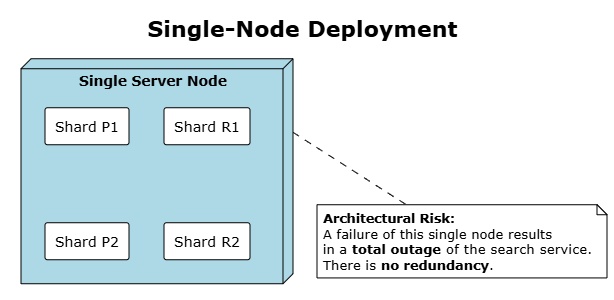
* 1. CA-19: Using CDN for reducing geo network latency
* We will utilize CDN server for keeping **website static content**
* CDN server replicates static content server to all **Edge location globally**
* User will be served with static content from nearest edge location, and will help in reducing geo network latency

**Targeted QA/NFR:**

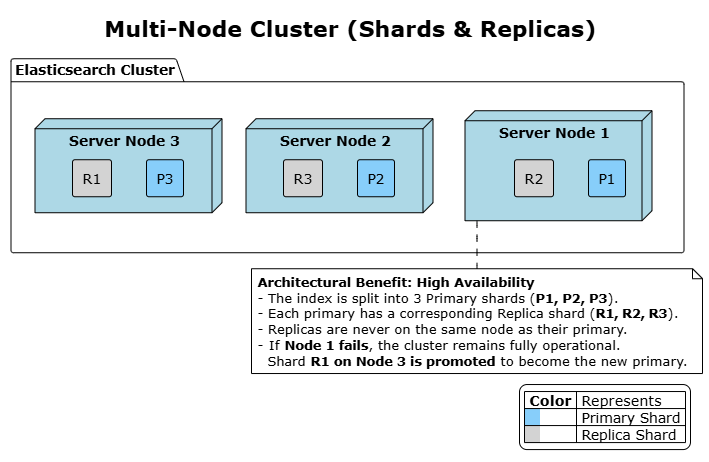
* NFR\_03 (Performance): Service should be able to scale if load increase by N%
* NFR\_04 (Reliability): System should be able to recover within ‘N’ minutes

**10. Concern: How to ensure High Scalability of System Components**

* 1. CA-20: Single-Node Deployment of Search Engine
* We can implement a **Single-Node Deployment** by running our entire **dedicated search engine on a single server**. All index shards (both primary and replica) would reside on this one machine.



* 1. CA-21: Multi-Node Cluster (Shards & Replicas)
* We can implement a **Multi-Node Cluster** by distributing our search engine across **multiple servers (nodes)**. The index is split into multiple **primary shards** to allow for parallel processing, and **replica shards** (copies) are created for redundancy.
* Doing this achieves **high availability and scalability**. **Sharding** allows the system to scale horizontally for performance (NFR\_03), while **replicas** ensure that if one node fails, the service remains fully operational by promoting a replica, satisfying **NFR\_04**



**11. Concern: Preventing Cascading Failures**

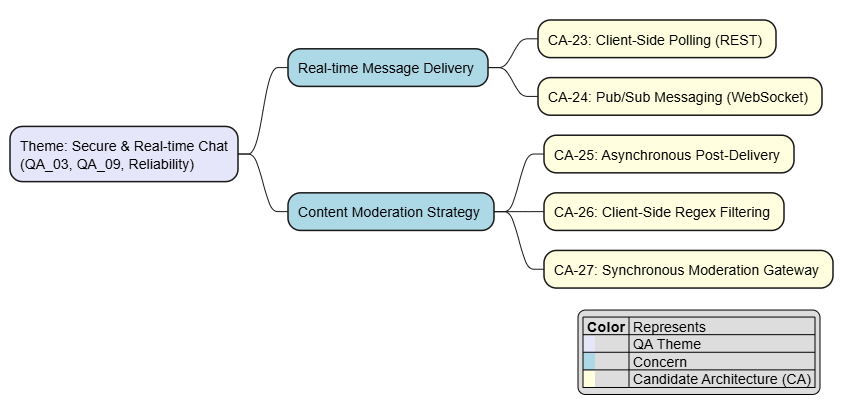
11.1 CA-22: Service Resilience via Circuit Breaker

* In a distributed micro services system, a failure in one downstream service (e.g., a recommendation engine) can cause a chain reaction.
* Upstream services that depend on it become blocked while waiting for responses, eventually exhausting their own resources and failing, which can lead to a system-wide outage known as a **cascading failure**.
* We can use **circuit breaker** that will help prevent this. It acts like an electrical circuit breaker by wrapping the calls to a downstream service. If it detects that the service is failing repeatedly, the circuit "trips" or "opens" and immediately fails any subsequent calls without even trying to contact the failing service.
* This prevents the upstream service from getting blocked, allowing it to handle the failure gracefully (e.g., by serving a default response) and giving the downstream service time to recover.

**Targeted QA/NFR:**

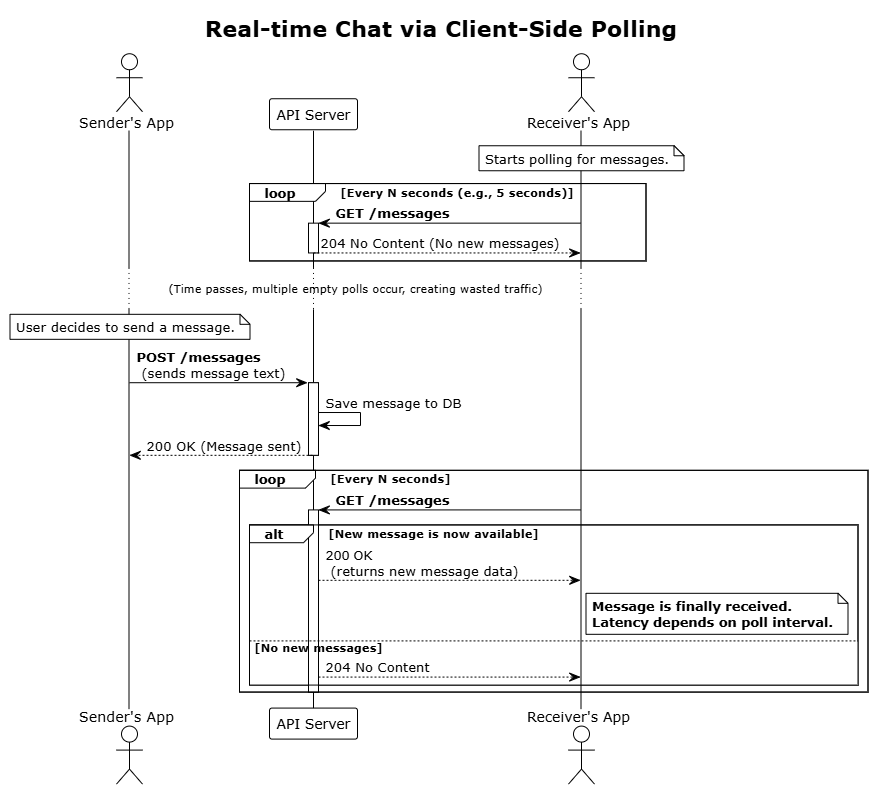
* QA\_03 (Security): System must block the sharing of contact information and any off the platform transaction
* QA\_09 (Performance): Users should be able to get chat message quickly

The chat feature is critical for negotiation and building trust. It must feel real-time, guarantee that messages are never lost and are always in the correct order, and proactively protect users from frauds or sharing personal information.

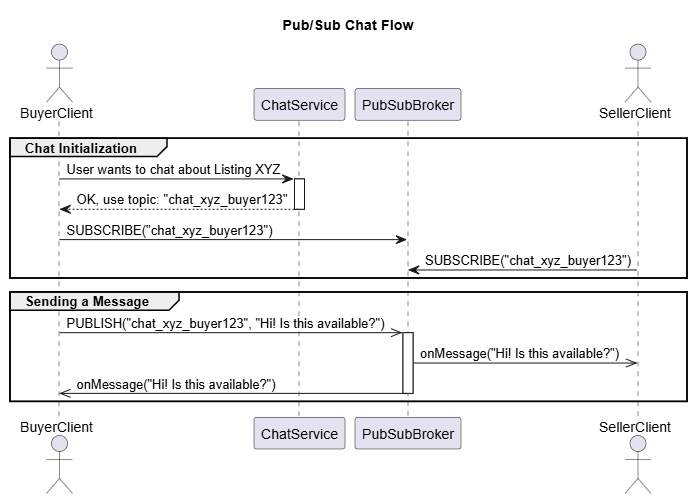


**12. Concern: Real-time Message Delivery Protocol**

* 1. CA-23: Client Side polling via REST API
* We can implement **Client-Side Polling** by having the **client application repeatedly send** HTTP GET requests to a /messages endpoint every few seconds. This is done to check for any new messages, while sending a message is handled by a separate **POST request**.

~~~~

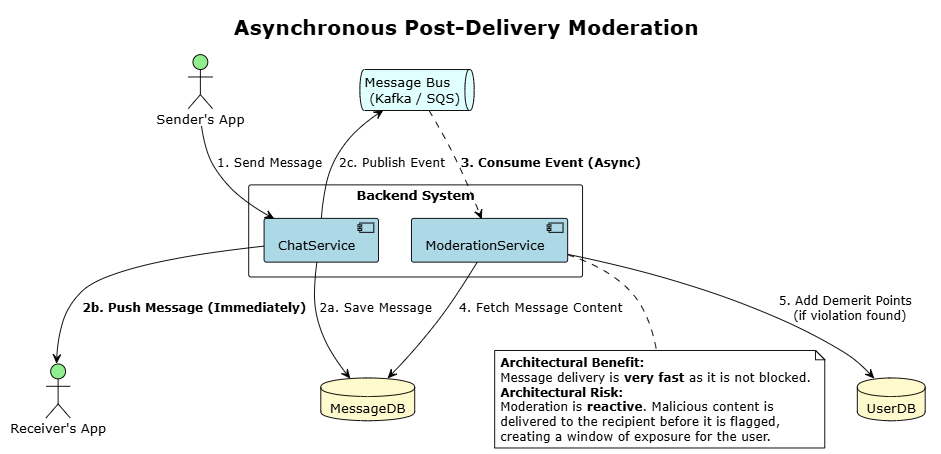
* 1. CA-24: Pub/Sub Messaging for Chat
* We can implement a **Pub/Sub (Publish/Subscribe) pattern** where each chat conversation is a unique 'topic'. When users join a chat, their client applications **subscribe** to that specific topic. To send a message, the application simply **publishes** it to the topic.
* A central message broker immediately pushes the published message to all subscribed users. This architecture enables true real-time, server-push communication, with the client's persistent connection typically handled by **WebSocket.**



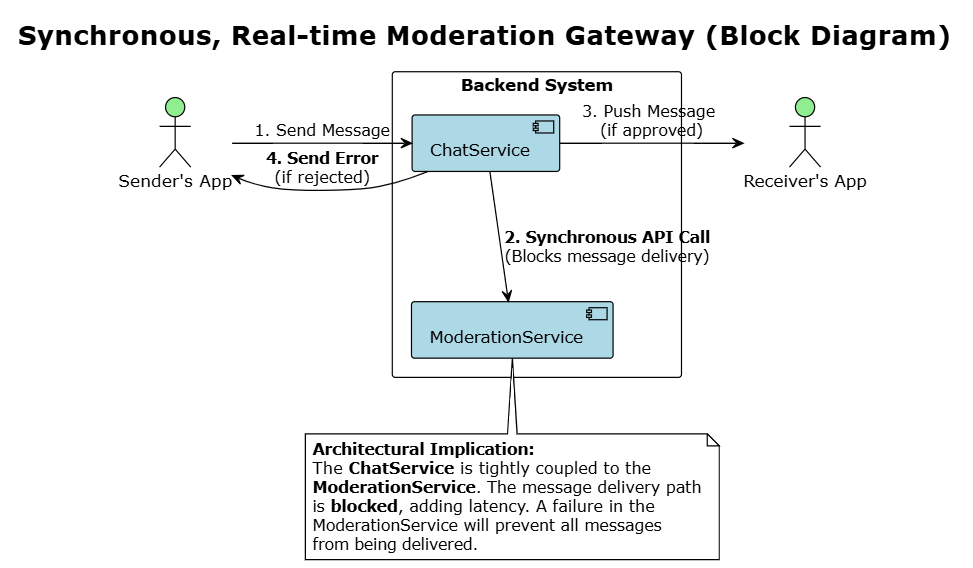
**13. Concern: Content Moderation in Chat**

This addresses how to scan messages for prohibited content before delivery.

* 1. CA-25: Asynchronous Post-Delivery Moderation
* Messages are delivered immediately to the recipient without real time moderation.
* A separate, asynchronous process scans the saved chat logs and flags messages for review after the fact. If a violation is found, demerit points added to user ratings and a moderator might take action later.



* 1. CA-26: Client Side Regex filtering
* We can implement **Client-Side Regex Filtering** by embedding a set of **regular expressions** directly into the client application. This logic would scan a user's chat message **before it is sent**, using patterns to detect PII such as **10-digit phone numbers**, email addresses containing **"@" and "."**, or external payment links containing **"http://"**.
  1. CA-27: Synchronous, Real-time Moderation Gateway
* We can implement a **Synchronous, Real-time Moderation Gateway** by having the ChatService, upon receiving a message, make a **blocking API call** to a dedicated ModerationService **before delivering the message**.
* This service then analyzes the message content in real-time using rules and ML models to scan for policy violations.
* The ChatService **waits for an 'approved' or 'rejected' response** before taking action. If approved the message is sent to receiver, else if rejected the sender is notified about why this message is not compliant with security guidelines.



Targeted QA/NFR:

* QA\_04 (High Priority): Listing Moderation & Malicious Content Prevention. Ensures all new and updated listings are automatically scanned for prohibited content before becoming visible.
* QA\_05 (High Priority): Listing Visibility Latency. This defines the maximum acceptable delay from when a seller publishes a listing to when it is discoverable in search.
* QA\_08 (Medium Priority): Listing Media Processing Time. This is the time taken by the system to process uploaded media, such as images.

When a seller submits a new listing, the system must perform a sequence of operations: moderate the text and images for safety (QA\_03), process the media files (QA\_08), and finally, make the listing discoverable in search and other services. The entire pipeline must be highly efficient to ensure low visibility latency for the seller (QA\_05).

**14. Concern: How to Moderate a Listing?**

14.1 CA-28: On Client (Portal or Mobile App) Rule-Based Engine

* We can implement a **Rule-Based Engine** by using a pre-defined set of **keyword blocklists and regular expressions** to scan the listing's title and description for obvious violations. This would include patterns for phone numbers, external URLs, and specific prohibited words.
* This architecture is **extremely fast and cheap to run**, providing an effective first line of defense. However, it is **brittle and easy for users to bypass** with simple misspellings, and its lack of contextual understanding can lead to high false positives, impacting the user experience.

14.2 CA-29: Public Pre-trained NLP Model (Server Side)

* We can implement a moderation system using a **public, pre-trained NLP model** (e.g., from Hugging Face). This approach leverages models that are already trained to identify common issues like **toxicity, hate speech, or PII** (using Named Entity Recognition).
* Doing this allows us to quickly implement a **context-aware moderation** system with **zero training effort**.

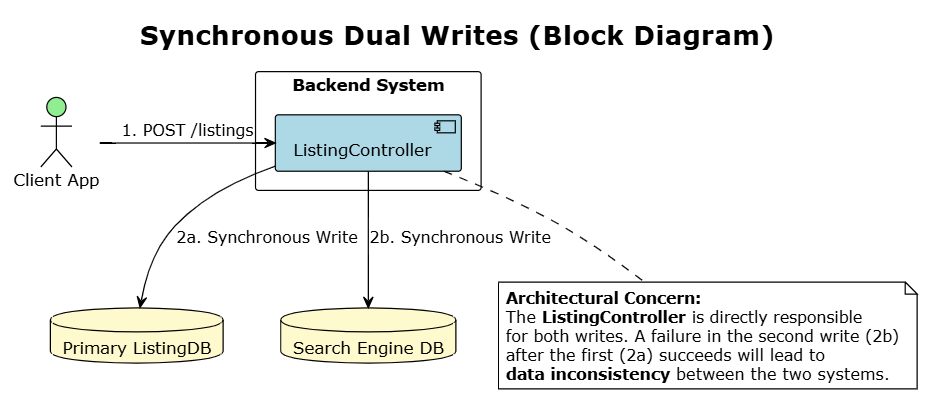
14.3 CA-30: On Server Validation checks: Large Language Models

* We can implement **LLM Validation** by sending the listing's title and description to a **general-purpose Large Language Model** via an API call. We would use a specific prompt asking the LLM to determine if the content violates any of our platform's policies.
* This architecture offers **superior understanding of nuance, context, and intent**, providing the highest level of accuracy for **QA\_04**. The primary architectural trade-offs are the **high cost per validation** and **high, variable latency** associated with LLM API calls.

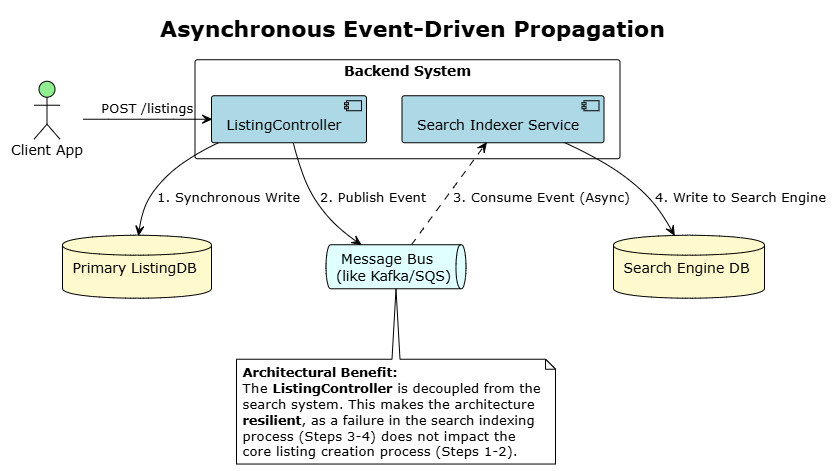
**15. Concern: Listing Visibility (Data Propagation)**

15.1 CA-31: Synchronous Dual Writes (both to primary master DB & Search engine DB)

* When the ListingController saves a new listing, it performs two database writes within the same request to:
  + 1. The primary Listing DB that will hold all the listings ever created in the system.
    2. Another search engine based DB (e.g., Elasticsearch), which will be indexed and made available for search.
* If any of the write operations fail, the whole operation is rolled back.

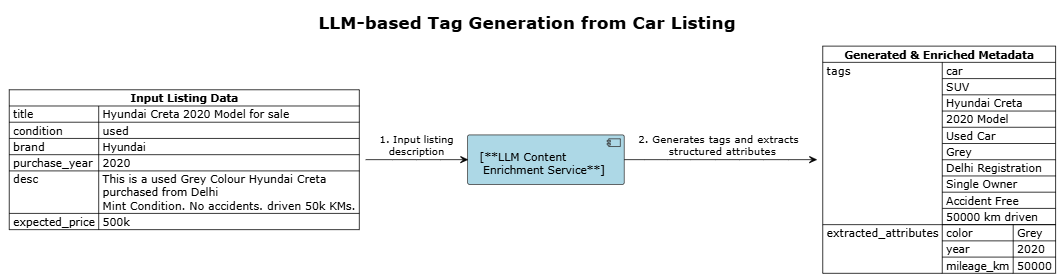


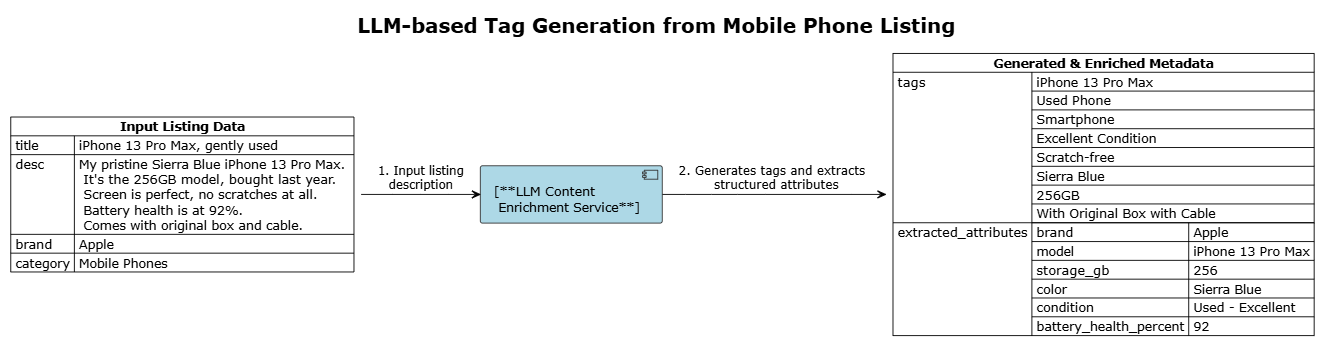
* 1. CA-32: Asynchronous Event-Driven Propagation
* We can implement **Asynchronous Event-Driven Propagation** by having the ListingController **write only to the primary database** and then immediately **publish an event** (e.g., ListingCreated) to a central **message bus** (like AWS SQS or Kafka).
* Doing this **decouples the core service** from the search system, as a separate and dedicated **'Search Indexer' service** independently subscribes to these events and updates the search engine. This makes the architecture **highly reliable and resilient** because a failure or slowdown in the search system will not block new listings from being created.



**16. Concern: Meta Data Enhancement during Listing Creation**

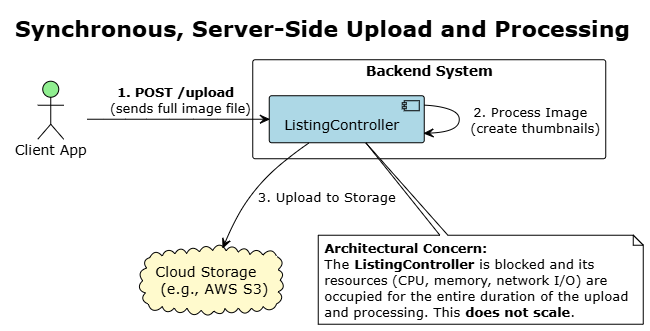
* 1. CA-33: Listing Meta Data Enhancement using LLM
* We can implement **Asynchronous Content Enrichment** by having a dedicated background service listen for ListingCreated events from a **message bus**. For each event, this service will take the listing's description and pass it to a **hosted Large Language Model (LLM)**.
* The LLM will **automatically generate relevant tags** and attributes (refer example below) which are then added as metadata to the search document. Doing this **significantly enhances search relevance (QA\_01)** by creating rich data for discovery, and because the process is **asynchronous**, it does not add any latency to the user's initial listing creation flow.



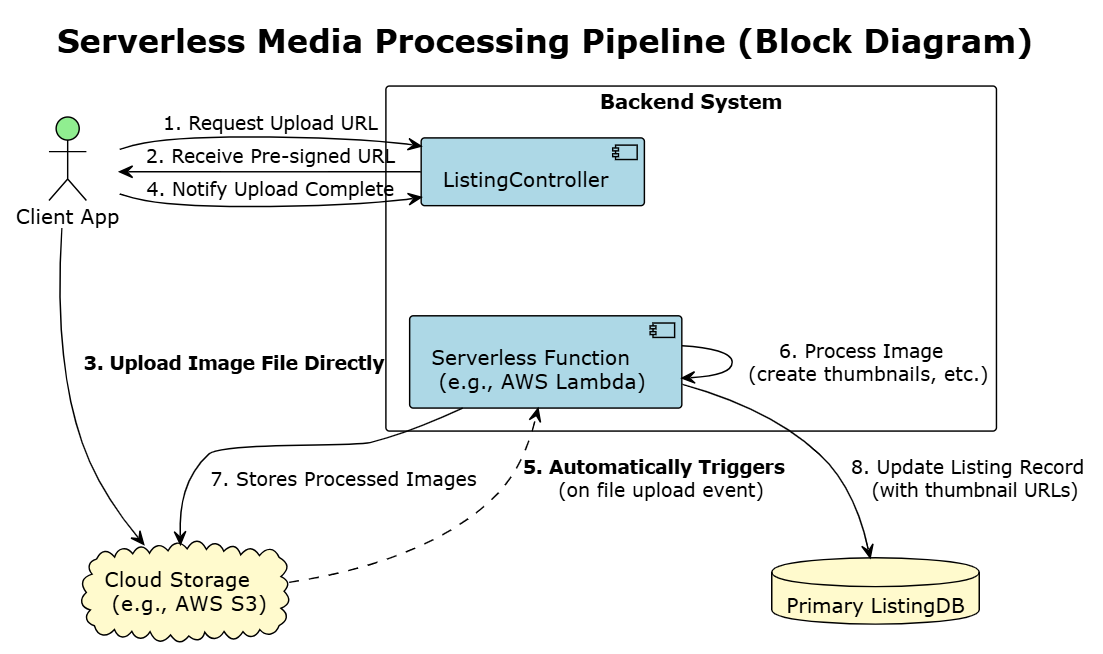


**17. Concern: Image Uploading & Processing while Listing Creation**

* 1. CA-34: Synchronous, Server-Side Upload and Processing
* We can implement a **Synchronous, Server-Side Pipeline** by having the client **upload the entire image file** directly to the ListingController's API endpoint. The controller is then responsible for both **processing the image** (e.g., creating thumbnails) and saving the results to cloud storage.
* This creates a performance **bottleneck**, as it ties up server resources for long-running operations and **underperforms** under concurrent user uploads.



* 1. CA-35: Asynchronous, Decoupled Media Pipeline
* We can implement an **Asynchronous, Decoupled Pipeline** by first having the client request a **secure pre-signed URL** from the backend. The client then uses this URL to **upload the image file directly** to cloud storage, completely bypassing our API server for the large data transfer
* The file upload to cloud storage then **automatically triggers a separate, serverless function** that runs in the background to handle all processing. Doing this **decouples media processing** from the core API and frees up server resources, making the architecture **highly scalable and resilient**.

.

**Targeted QA/NFR:**

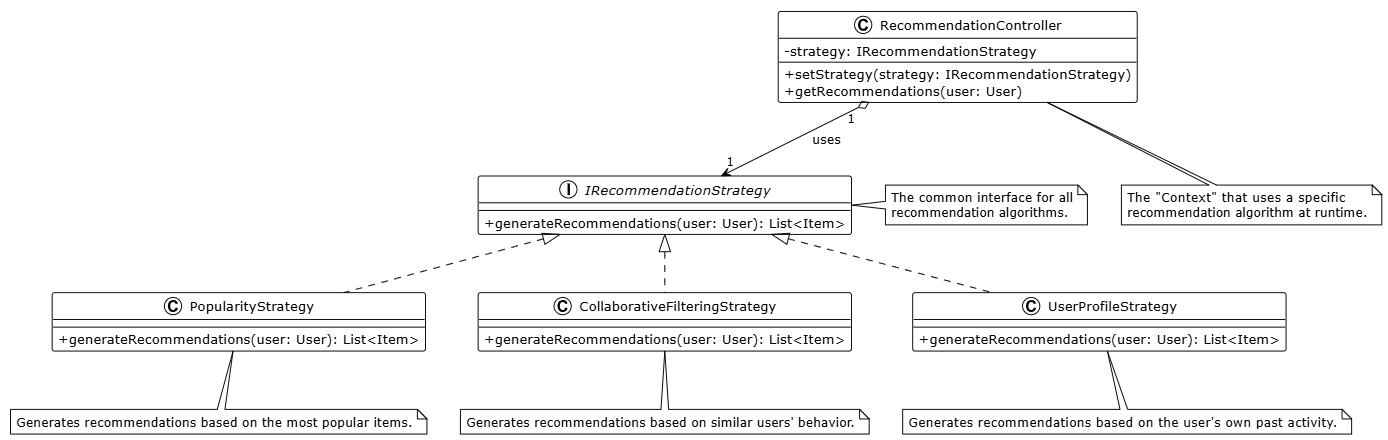
* QA\_06 (Maintainability): Adaptability for New Recommendation Algorithms.

**18. Concern: How to make a system where multiple algorithms can be running at same time & also how to smoothly transition between those?**

* 1. CA-36: Hardcoded Logic in Controller

The logic for generating recommendations is written directly inside the HomeController or RecommendationController. To change the algorithm, an engineer must modify and redeploy this core service.

* 1. CA-37: Strategy Design Pattern
* The controller delegates the task to a RecommendationStrategy interface. We can have multiple concrete implementations (e.g., PopularityStrategy, CollaborativeFilteringStrategy).
* To add a new algorithm, we simply create a new class that implements the interface. A configuration file or environment variable determines which strategy is active.



* 1. CA-38: A/B Testing

The controller calls a generic RecommendationService. This Service, in turn, queries an external A/B Testing Microservice to determine which algorithms to be used for this call. This allows a safe, gradual rollouts and data-driven comparison of different algorithms on live traffic.

**Targeted QA/NFR:**

* QA\_07 (Security): System should be able to flag high-risk sellers in real-time to prevent abuse.

**19. Concern: How can system detect risky sellers from user activity?**

* 1. CA-39: Rule-Based Engine (for seller Ratings)
* A dedicated RiskService uses a set of hardcoded rules to generate an instant, automated risk score
* An example rule is: IF new\_seller AND item\_is\_phone AND price < 50%\_market\_avg THEN score = HIGH\_RISK
* This provides an immediate score, but the rules can become complex and brittle, failing to adapt to new fraud patterns.

* 1. CA-40: Event driven ML Scoring
* An event stream of all seller activities (e.g., account creation, profile updates, listing posts) is fed into a real-time ML model.
* The model continuously updates a dynamic risk score for each seller based on their behavioral patterns
* This architecture allows the system to detect and respond to suspicious activity as it happens
  1. CA-41 User to User Rating
* This architecture introduces a reputation system where buyers can rate and review sellers after a transaction.
* A seller's aggregated rating and the volume of feedback directly influence their trust and risk score.
* This is a reactive measure based on past transaction outcomes, not a proactive, real-time analysis of a seller's current actions.

**Targeted QA/NFR:**

* QA\_10 (Reliability): Monitoring and Alerting of Cloud MicroServices.

**20. Concern: effective monitoring of Cloud Services**

* 1. CA-42: Basic Cloud Metrics and Logs
* This Architecture relies exclusively on default metrics provided by the cloud vendor (e.g., CPU Utilization, Memory Usage).
* Each MicroService logs to standard output, with no central aggregation.
* Diagnosing issues requires engineers to manually connect to individual servers to read log files, making it a slow and reactive process  
  1. CA-43: Centralized Logging & Metrics
* All micro services are configured to ship their logs and application-level metrics (e.g., request latency, error rates) to centralized platforms.
* Logs are sent to a dedicated system like the ELK Stack (Elasticsearch, Logstash, Kibana) for unified search and analysis.
* Metrics are sent to a time-series database like Prometheus, with dashboards and proactive alerting configured in a tool like Grafana.

1. CA-44: Distributed Tracing

* This architecture builds upon centralized logging and metrics.
* A tracing framework (like OpenTelemetry or Jaeger) is implemented, assigning a unique Trace ID to every initial user request.
* This Trace ID is passed along in the header of every subsequent downstream micro service call.
* This allows developers to visualize the entire end-to-end journey of a single request across multiple services, making it easy to pinpoint performance bottlenecks and errors.

1. Candidate Architecture Evaluation

// A7. Architecture Design

// C7-1. Is comparison analysis of colliding candidates appropriate? (evidence)

// C7-2. Is there sufficient complement of the selected candidate?

Summary of all Candidate Architectures

|  |  |  |
| --- | --- | --- |
| **S. No.** | **Title** | **Status** |
| CA-01 | Simple Relational Database | Rejected |
| CA-02 | Dedicated Search Index Service | Selected |
| CA-03 | Vector DB for searching | Rejected |
| CA-04 | Fuzzy Search Query | Selected |
| CA-05 | Synonym Set | Selected |
| CA-06 | City/Zip Matching Only | Rejected |
| CA-07 | Geometric Proximity Filter | Rejected |
| CA-08 | Native Geospatial Indexing | Selected |
| CA-09 | Default Engine Ranking (Baseline) | Rejected |
| CA-10 | Static Business-Rule Reranking | Selected |
| CA-11 | Distributed Cache for Search Results | Selected |
| CA-12 | One Massive Monolith Index | Rejected |
| CA-13 | Hybrid Lifecycle Management Strategy | Selected |
| CA-14 | Server-Side Aggregation Pattern | Rejected |
| CA-15 | Asynchronous Client-Side Composition | Selected |
| CA-16 | Pre-computed Feed Service (Location Only) | Selected |
| CA-17 | User-Profile Based Recommendations | Rejected |
| CA-18 | Hybrid Recommendation Engine | Selected |
| CA-19 | Using a CDN | Selected |
| CA-20 | Single-Node Deployment of Search Engine | Rejected |
| CA-21 | Multi-Node Cluster (Shards & Replicas) | Selected |
| CA-22 | Service Resilience via Circuit Breaker | Selected |
| CA-23 | Client Side polling via REST API | Rejected |
| CA-24 | Pub/Sub Messaging for Chat | Selected |
| CA-25 | Asynchronous Post-Delivery Moderation | Rejected |
| CA-26 | Client-Side Regex filtering | Selected |
| CA-27 | Synchronous, Real-time Moderation Gateway | Selected |
| CA-28 | Rule-Based Engine (for Listing Moderation) | Selected |
| CA-29 | Public Pre-trained NLP Model | Selected |
| CA-30 | Large Language Model (for Listing Moderation) | Rejected |
| CA-31 | Synchronous Dual Writes | Rejected |
| CA-32 | Asynchronous Event-Driven Propagation | Selected |
| CA-33 | Listing Meta Data Enhancement using LLM | Selected |
| CA-34 | Synchronous, Server-Side Upload and Processing | Rejected |
| CA-35 | Asynchronous, Decoupled Media Pipeline | Selected |
| CA-36 | Hardcoded Logic in Controller | Rejected |
| CA-37 | Strategy Design Pattern | Selected |
| CA-38 | A/B Testing Microservice | Selected |
| CA-39 | Rule-Based Engine | Rejected |
| CA-40 | Event driven ML Scoring | Selected |
| CA-41 | User to User Rating | Selected |
| CA-42 | Basic Cloud Metrics and Logs | Rejected |
| CA-43 | Centralized Logging & Metrics | Selected |
| CA-44 | Distributed Tracing | Selected |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** | | | |
| **Conflict** | | | |
| **CA-01: Simple Relational Database** | **CA-02: Dedicated Search Index** | **CA-03: Vector DB** |
| **Advantages** | Simplest to implement; no extra infrastructure; perfect data consistency. | High performance; feature-rich; horizontally scalable for search traffic. | State-of-the-art for semantic search; excellent for discovery features. |
| **Disadvantages** | Poor performance and relevance at scale; couples workloads. | Adds operational complexity like Cluster operations; introduces data synchronization latency. | Incomplete solution; cannot perform essential keyword search or complex filtering |
| **NFR\_01: Search Time** | (--) Fails to meet requirements under any significant load. | (++) Excellent. Purpose-built for low-latency queries. | (+) Good search time for *semantic* queries only but **not applicable for the primary keyword search requirement** |
| **QA\_01: Search Relevance** | (+) FTS provides a good relevance baseline, but overall control is limited. | (++) Excellent. Provides fine-grained relevance tuning capabilities. | (++) Excellent for *semantic* relevance , **but fails completely on *keyword* precision** |
| **NFR\_03: Service Scalability** | (--) Does not scale independently. Puts load on the monolithic database. | (++) Excellent. Designed for horizontal scaling of read traffic by adding more nodes. | (++) Excellent. Vector databases are designed to scale independently |
| **NFR\_04: System Recovery** | (-) Creates a single point of failure. A database failure impacts all functions. | (+) Good. Decouples search from core transactions, which isolates failures. | (+) Good. Also provides failure isolation for the semantic search component. |
| **QA\_05: Listing Visibility Latency** | (++) Excellent. Data is instantly consistent as it queries the source of truth. | (-) Introduces a small, managed latency due to the asynchronous indexing process | (-) Also introduces indexing latency for the vector data. |
| **Decision** |  | **SELECTED** because it is the only candidate that provides the necessary balance of high performance, advanced features, and scalability required to satisfy our highest-priority requirements (NFR\_01, QA\_01, NFR\_03). |  |

* **Risk:** The asynchronous Search Indexer service could fail to process an update event from the message bus, or it might process events out of order. This would cause the search index to become stale or inconsistent with the primary database (e.g., a sold item still appears as available in search results).
* **Solution:** Implement a **Dead-Letter Queue (DLQ)** for the message bus. If the indexer service fails to process a message after several automated retries, the message is moved to the DLQ. This triggers an immediate alert for an engineer to investigate the specific failure without losing the update event, ensuring data can be reconciled later.

|  |  |  |
| --- | --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** | |
| **CA-04: Fuzzy Search Query** | **CA-05: Synonym Set** |
| **Advantages** | Simple to implement; handles common typos effectively. | Bridges vocabulary gap between buyers and sellers; improves recall. |
| **Disadvantages** | Can sometimes return irrelevant results if too permissive. | Requires manual effort to maintain the synonym dictionary. |
| **QA\_01: Search Relevance** | (+) Good. Directly solves the common and frustrating problem of spelling mistakes. | (++) Excellent. Significantly improves relevance by finding items the user meant but described differently. |
| **NFR\_01: Search Time** | (++) No significant impact. This is a native, highly optimized feature of the selected CA-02. | (+) Minor impact if using a decoupled service, but negligible if integrated directly into the search engine. |
| **QA\_06: Adaptability** | (++) N/A. Standard feature. | (+) Good. The synonym dictionary can be easily updated without code changes. |
| **Decision** | **SELECTED** as it is an essential baseline feature. Adding it will enhance the system performance | **SELECTED** because it will improve the search relevance, and also this can be directly integrated to Search engines like Elastic Search |

* **Risk:** The primary risk is that poorly configured fuzzy matching or a stale/inaccurate synonym dictionary can lead to "over-matching," where the search returns irrelevant results, negatively impacting **QA\_01: Search Result Relevance**.
* **Solution:** The fuzzy search feature will be tuned with a low edit distance (e.g., 1) and monitored via A/B testing. A governance process for the synonym dictionary will be established with an internal tool for non-technical teams to manage updates.

|  |  |  |  |
| --- | --- | --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** | | |
| **Conflict** | | |
| **CA-06: City/Zip Matching** | **CA-07: Rectangular Bounding Box Query** | **CA-08: Native Geospatial Indexing** |
| **Advantages** | Simplest to implement; uses basic text filtering. | Fast on standard databases; a simple geometric approximation. | Highly accurate and high-performance; provides true radius-based filtering and sorting. |
| **Disadvantages** | Geographically imprecise; does not represent true proximity. | Inaccurate at the corners of the box; returns items outside the desired radius. | Dependent on having accurate latitude/longitude data for all listings. |
| **QA\_01: Search Relevance** | (-) Poor. A search in a large city may return results that are geographically very far from the user. | (+) Fair. An improvement over text matching, but the inaccuracy can still lead to irrelevant results. | (++) Excellent. Provides the most relevant results for "near me" searches, a critical use case. |
| **NFR\_01: Search Time** | (++) Very fast text match. | (++) Very fast on standard indexes. | (++) Excellent. This is a highly optimized, native feature within the selected CA-02. |
| **Decision** |  |  | **SELECTED**. It offers the best combination of geographic accuracy and performance and is a standard feature of the chosen foundational engine (CA-02). |

* **Risk:** The architecture is dependent on having accurate latitude/longitude data for all listings.
* **Solution:** Implement an address validation and geocoding service (e.g., using Google Maps API) during the listing creation process to ensure clean and accurate location data.

|  |  |  |
| --- | --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** | |
| **Conflict** | |
| **CA-09: Default Engine Ranking** | **CA-10: Static Business-Rule**  **Re-ranking** |
| **Advantages** | Simplest possible approach; no extra development effort. | Easy to implement; directly injects critical business logic into the search results. |
| **Disadvantages** | Lacks any business context; provides a generic, one-size-fits-all ranking. | Rules are static and can become brittle; requires manual tuning to adapt to new patterns. |
| **QA\_01: Search Relevance** | (+) Provides a good baseline relevance score from the search engine's text score. | (++) Excellent. Directly improves relevance by applying business context (e.g., boosting seller ratings). |
| **QA\_06: Adaptability** | (--) Poor. Changing the core ranking algorithm of the search engine is a complex, deep change. | (+) Good. Ranking rules can be easily modified in a configuration file or a simple service, allowing for quick adjustments to business strategy. |
| **Decision** |  | **SELECTED** as it offers the best balance of a significant relevance improvement for a low implementation cost, making it ideal for the initial product launch. |

* **Risk:** The most critical risk is **algorithmic bias creating an unfair marketplace**. Continuously boosting listings from highly-rated sellers creates a "rich-get-richer" feedback loop where established sellers dominate. New sellers are perpetually buried and unable to gain visibility, which stagnates inventory diversity and harms the platform's long-term health.
* **Solution:** Mitigate this by including rules to counteract the bias. The re-ranking logic will implement temporary **"new seller" or "first listing" boosts** to provide new participants with a fair chance at initial visibility. The system will also monitor marketplace fairness metrics to ensure the ecosystem remains healthy and competitive.

|  |  |
| --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** |
| **CA-11: Distributed Cache** |
| **Advantages** | Drastically reduces latency for common queries; reduces load on the search engine. |
| **Disadvantages** | Introduces a risk of data staleness; adds a new component to manage. |
| **NFR\_01: Search Time** | (++) Excellent. Makes popular and repeated searches feel instant by serving from memory. |
| **QA\_05: Listing Visibility Latency** | (-) Introduces a risk that a user might see an outdated price or a sold item in the cached results if the cache is not properly invalidated. |
| **Decision** | **SELECTED** as it is an essential component for meeting the strict performance requirements of NFR\_01 |

* **Risk**: Cached data can become stale, leading to a poor user experience.
* **Solution:** Implement an event-driven cache invalidation strategy. When a listing is updated, publish an event that triggers the proactive removal of relevant entries from the cache. Use a short Time-to-Live (TTL) on all cache entries as a fallback measure.

|  |  |  |
| --- | --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** | |
| **Conflict** | |
| **CA-12: One Massive Monolith Index** | **CA-13: Hybrid Lifecycle Management Strategy** |
| **Advantages** | Simplest to implement; requires no extra logic or configuration; data is never accidentally deleted by a policy. | Ensures both immediate data correctness (removes sold items) and long-term performance/cost efficiency (deletes old data in bulk). |
| **Disadvantages** | Performance degrades significantly over time; becomes operationally difficult to manage; high cost as all data is kept on expensive storage. | Higher initial implementation complexity; requires careful configuration of both event-driven logic and ILM policies. |
| **NFR\_01: Search Time** | (--) Very Negative. Fails to meet the NFR in the long term as the index bloats with irrelevant historical data. | (++) Excellent. Ensures sustained low latency by keeping the active search index small and fast. |
| **NFR\_03: Service Scalability** | (-) Poor. Scaling a single massive index is inefficient and operationally challenging. | (++) Excellent. This pattern is designed for long-term scalability and efficient resource management. |
| **QA\_05: Listing Visibility Latency** | (--) Very Negative. Sold items remain in the search index indefinitely, leading to a very poor user experience with stale results. | (++) Excellent. The event-driven part removes sold/deleted items in near real-time, ensuring high data correctness. |
| **Cost Efficiency** | (--) Very Negative. All data, including useless old data, is kept on expensive "hot" storage indefinitely. | (++) Excellent. ILM automatically moves old data to cheaper storage or deletes it, optimizing infrastructure costs over time. |
| **Decision** |  | **SELECTED** as it is the only robust, production-grade solution that addresses both the immediate need for data correctness (removing sold items) and the long-term need for performance and cost management |

* **Risk:** The primary risk is misconfiguration. An incorrect ILM policy could delete data too early, or a failure in the event-driven deletion process could lead to data inconsistencies.
* **Solution:** Mitigate this by implementing a two-part strategy:
  + All ILM policies must be thoroughly tested in a staging environment before production deployment. Specific monitoring and alerts will be created for the ILM process.
  + The event-driven deletion process will use a Dead-Letter Queue (DLQ) to capture any failed deletion events for manual investigation, ensuring no updates are lost.

|  |  |  |
| --- | --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** | |
| **Conflict** | |
| **CA-14: Server-Side Aggregation Pattern** | **CA-15: Asynchronous Client-Side Composition** |
| **Advantages** | Simplifies client-side logic; reduces the number of network round trips from the client to the backend. | Improves perceived performance as UI components can render independently as data arrives; provides greater resilience. |
| **Disadvantages** | Poor perceived performance as the client is blocked until all backend calls are complete; creates a single point of failure. | Increases client-side complexity as the client application is now responsible for orchestrating multiple API calls. |
| **NFR\_02: Home Screen Loading Time** | (--) Very Negative. The total response time is the sum of all sequential backend calls, which is very slow for the user. | (++) Excellent. The user sees content appearing on the screen much faster, even if the total time to load all data is the same. |
| **QA\_02: Recommendation Usefulness** | (NA) Neutral. This pattern can deliver useful recommendations, but a long loading time may cause users to abandon the page before seeing them. | (+) Good. By loading recommendation carousels quickly and independently, it increases the likelihood of user engagement. |
| **NFR\_04: System Recovery** | (-) Poor. A single slow or failing downstream service (e.g., the RecommendationService) can cause the entire home page to fail. | (+) Good. If the RecommendationService fails, the rest of the home page (categories, etc.) can still load successfully, gracefully degrading the experience. |
| **Decision** |  | **SELECTED** as it provides a superior user experience by prioritizing perceived performance and offers a more resilient architecture. |

* **Risk:** The primary risk is the increased complexity on the client application, which now has to manage the state of multiple concurrent data requests.
* **Solution:** Mitigate this by using a well-established client-side state management library (e.g., Redux, MobX) and a robust API client that can handle concurrent requests, loading states, and error handling gracefully for each UI component.

|  |  |  |  |
| --- | --- | --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** | | |
| **CA-16: Pre-computed Feed (Location Only)** | **CA-17: User-Profile Based Recommendations** | **CA-18: Hybrid Recommendation Engine (User Profile + Collaborative)** |
| **Advantages** | Lowest possible latency for home page load; highly scalable serving layer; works for new users (cold start). | Directly personalizes the experience for each user based on their explicit interests. | Best of both worlds: combines deep personalization with community-driven serendipitous discovery. |
| **Disadvantages** | Not personalized for logged-in users; data can be stale (eventually consistent). | Struggles with discovery and can create a "filter bubble," limiting user engagement with new types of items. | Highest architectural complexity; requires managing and combining multiple models and data sources. |
| **NFR\_02: Home Screen Loading Time** | (+++) Superior. The entire feed is retrieved in a single, ultra-fast cache lookup, and will most likely be served from cache as it same for all users in a certain location. | (++) Excellent. Serving a pre-calculated list from a simple database is also very fast. | (++) Excellent. The serving performance is identical to CA-17, as the heavy computation is precomputed. |
| **QA\_02: Recommendation Usefulness** | (-) Poor. For a logged-in user, location-only recommendations are not highly relevant or useful.  (+) Good(for Cold Start[User not logged in]). Helps generate at least some results | (+) Good. It is personalized, but its inability to help users discover new items limits its potential. | (+++) Superior. The ability to augment a user's known tastes with discoveries from similar users provides the most engaging experience. |
| **QA\_06: Adaptability** | (-) Poor. This is a very simple, static logic that is not designed to be adaptable for new personalization algorithms. | (+) Fair. The single-model approach means replacing the entire algorithm is a significant change. | (++) Excellent. The hybrid structure is inherently modular, allowing the user-profile or collaborative models to be updated or replaced independently. |
| **Decision** | **SELECTED** as it is the most effective pattern for delivering the results of the complex CA-18 computation with the lowest possible latency, best satisfying NFR\_02. Also, it is helpful in case of a cold start when we do not have any user data. |  | **SELECTED** as it is the best option that provides both deep personalization and serendipitous discovery, best satisfying QA\_02. |

Further Analysis on why we are choosing both CA-16 & CA-18:

Given our selection of **CA-13: Asynchronous Client-Side Composition**, which allows the user interface to render in pieces, we can adopt a more sophisticated loading strategy for our recommendations. In this approach:

* Initially, the client will immediately request the fast, pre-computed, location-only feed from **CA-16**. This allows recommendation carousels to be populated almost instantly, satisfying **NFR\_02 (Home Screen Loading time)** by providing immediate visual content.
* In parallel, a second request will fetch the slower, deeply personalized results from the **CA-18: Hybrid Recommendation Engine**. When this richer data arrives, the client will seamlessly add this new data over the initial location-based content. This hybrid approach ensures the user perceives an ultra-fast load while ultimately receiving the most relevant recommendations, maximizing **QA\_02 (Recommendation Usefulness)**.

* **Risk:** Poor Recommendation Quality due to Data Sparsity. This will cause the model to produce irrelevant or nonsensical recommendations, which damages user trust in the feature's usefulness (QA\_02).
* **Solution**: Mitigate this by designing the recommendation engine to detect when the collaborative model returns low-confidence results. In such cases, the system must automatically fall back to the simpler, more predictable User-Profile Based model (CA-17) or even a basic "Popularity-Based" model. This ensures the user always receives a reasonable and relevant recommendation, even when the advanced model lacks sufficient data to perform well.

|  |  |
| --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** |
| **CA-19: Using a CDN** |
| **Advantages** | Drastically reduces latency for global users; offloads traffic from origin servers; improves availability. |
| **Disadvantages** | Adds a component to configure; introduces a risk of serving **stale content** if the cache is not managed properly. |
| **NFR\_02: Home Screen Loading Time** | (++) Excellent. Serves static assets from a **geographically close edge location**, minimizing network latency and speeding up page rendering. |
| **NFR\_03: Service Scalability** | (++) Excellent. **Offloads a significant portion of traffic**, allowing backend services to scale more efficiently for their core API tasks. |
| **NFR\_04: System Recovery** | (+) Good. A CDN can improve availability by **serving cached assets** even if the origin server is temporarily unavailable. |
| **Cost Efficiency** | (++) Positive. CDN egress bandwidth is typically **cheaper than origin egress**, leading to significant cost savings at scale. |
| **Decision** | **SELECTED** asit is a foundational architectural pattern that provides significant, measurable improvements to performance, scalability, and availability. |

* **Risk:** The most critical risk is cache invalidation. If a seller updates a product image, users might continue to see the old, cached version from the CDN, leading to confusion and a poor user experience.
* **Solution:** Mitigate this by implementing a robust cache invalidation strategy. When an asset is updated, the system will programmatically issue an invalidation request to the CDN API to purge the old version from edge caches. Additionally, use versioned URLs (e.g., image-v2.jpg) as a fallback to ensure clients always request the latest version.

|  |  |  |
| --- | --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** | |
| **Conflict** | |
| **CA-20: Single-Node Deployment** | **CA-21: Multi-Node Cluster (Shards & Replicas)** |
| **Advantages** | Simplest and cheapest setup for development or very early-stage use. | Provides **high availability** and fault tolerance; allows for **horizontal scaling** of search throughput. |
| **Disadvantages** | Creates a **single point of failure** with no redundancy; cannot handle high traffic. | Higher operational complexity and infrastructure cost compared to a single node. |
| **NFR\_03: Service Scalability** | (--) Very Negative. This architecture **cannot be scaled** to handle any significant increase in concurrent user traffic. | (++) Excellent. **Search throughput can be increased** by adding more nodes or replicas to the cluster. |
| **NFR\_04: System Recovery** | (--) Very Negative. A single node failure results in a **total service outage** requiring manual recovery. | (++) Excellent. The cluster **automatically fails over** to a replica shard in case of a node failure, ensuring zero downtime. |
| **NFR\_01: Search Time** | (+) Good, but only under very low load. Performance degrades quickly as traffic increases. | (++) Excellent. Sharding allows for **parallel query execution**, which maintains low latency even with large data volumes. |
| **Decision** |  | **SELECTED** as it is the viable option for a production-grade, resilient, and scalable search service that can support the project's long-term growth. |

* **Risk:** A multi-node cluster has a higher operational complexity. Misconfiguration of sharding or replica allocation can lead to "hot spots" (unbalanced load) or inefficient resource usage.
* **Solution:** Mitigate this by using a **managed cloud service** (e.g., Amazon OpenSearch Service), which abstracts away much of the underlying cluster management complexity. Additionally, implement robust monitoring (QA\_11) specifically for shard health and resource distribution to proactively identify and resolve any imbalances.

|  |  |
| --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** |
| **CA-22: Service Resilience via Circuit Breaker** |
| **Advantages** | Prevents cascading failures by isolating failing services, thereby improving overall system resilience. Allows downstream services time to recover without being overwhelmed by continuous requests. Provides a graceful degradation of service (e.g., home page loads without recommendations) instead of a total failure. |
| **Disadvantages** | Adds configuration complexity; requires careful tuning of thresholds (e.g., error rate, latency) for when the circuit should 'open'. Can slightly increase latency in the 'half-open' state as it probes the failing service. |
| **NFR\_04: System Recovery** | (+++) Superior. This is a foundational pattern for building fault-tolerant systems. It directly addresses the requirement by preventing a single component failure from spreading, thus enabling faster and more isolated recovery. |
| **NFR\_03: Service Scalability** | (+) Good. By preventing resource exhaustion from failing downstream calls, it ensures that services remain stable and can scale to handle their primary traffic load, even when dependencies are degraded. |
| **User Experience** | (+) Good. Instead of seeing a complete error page, the user experiences a gracefully degraded service. For example, the home page might load successfully but without the 'Recommended for You' section, which is a far better experience than a total outage. |
| **Decision** | **SELECTED** as it is a must-have for any system built with many small services. It provides strong protection against the disastrous domino effect where one service failing brings down others in a chain reaction. |

* **Risk:** The primary risk is **misconfiguration**. If the thresholds are too sensitive (e.g., the circuit opens after only one or two failed requests), the system will be overly fragile, and users will experience degraded service unnecessarily. If the thresholds are too lenient, the breaker may not trip in time to prevent a cascading failure.
* **Solution:** This risk can be mitigated by implementing **dynamic, adaptive thresholds** that adjust based on real-time service performance rather than static numbers. We will use a mature, well-tested library (e.g., Resilience4j) and establish a baseline performance profile for each service in a staging environment. Furthermore, the state of all circuit breakers will be a key metric in our monitoring dashboards (QA\_10), with alerts configured to fire whenever a critical service's circuit trips, ensuring immediate operational awareness.

|  |  |  |
| --- | --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** | |
| **Conflict** | |
| **CA-23: Client Side polling via REST API** | **CA-24: Pub/Sub Messaging for Chat** |
| **Advantages** | Simple to implement; uses standard, stateless HTTP requests which are easy to scale. | Enables true real-time, low-latency communication; highly efficient as the server pushes data, eliminating wasteful network traffic. |
| **Disadvantages** | Not real-time; introduces significant delays, making conversations feel slow; inefficient and resource-intensive for both the client (battery) and server (network traffic). | Higher initial implementation complexity; requires managing stateful, persistent connections at scale. |
| **QA\_09: Chat Message Delivery Time** | (--) Very Negative. The latency is fundamentally high, making the chat feel unresponsive and not conversational. It directly fails to meet the core requirement of a modern chat system. | (++) Excellent. It is purpose-built for low-latency, bi-directional communication, providing the best possible user experience for real-time chat. |
| **NFR\_03: Service Scalability** | (-) Poor. The constant polling from thousands of clients generates enormous, unnecessary load on the network and backend services, making it inefficient and expensive to scale. | (+) Good. While stateful connections add complexity, modern message brokers and WebSocket gateways are designed to scale horizontally to millions of concurrent users. |
| **Reliability (Message Ordering & Durability)** | (-) Poor. Prone to message ordering issues and requires complex client-side logic to manage state and resynchronize, especially during network interruptions. | (++) Excellent. Message brokers used in Pub/Sub patterns are designed to guarantee message ordering and durability, providing a highly reliable foundation. |
| **Decision** |  | **SELECTED** as it is good scalable solution that can help achieve our requirement and also ensure low latency. It can also integrate well with CA-26 that we have selected for moderation part. |

* **Risk:** The primary risk is managing a large number of stateful, persistent WebSocket connections. If a server node handling these connections fails, all users connected to it will be abruptly disconnected, potentially leading to a poor user experience and lost messages if not handled gracefully.
* **Solution:** This risk will be mitigated by implementing a **resilient WebSocket gateway layer** with robust client-side reconnection logic. Clients will be designed to automatically re-establish their connection to a new healthy node upon disconnection and resynchronize their chat history to ensure no messages are lost and the user experience is seamless.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** | | | |
|  | **Conflict** | | |
| **CA-26: Client-Side Regex filtering** | **CA-25: Asynchronous Post-Delivery Moderation** | **CA-27: Synchronous, Real-time Moderation Gateway** |
| **Advantages** | Provides instantaneous user feedback without any network latency; simple to implement. | Ensures fast message delivery as it does not block the real-time chat path; server-side logic can be complex and is easily updatable. | Proactively blocks malicious content before it reaches the recipient, providing the strongest user protection. |
| **Disadvantages** | Architecturally insecure and unreliable as client-side validation can be easily bypassed; updating rules requires a new application release. | Reactive, not proactive; malicious content is delivered to the recipient before it is flagged, creating a window of exposure for the user. | Adds latency to every message, making the chat feel slower; creates a hard dependency on the moderation service. |
| **QA\_03: Personal Information & Off-platform Solicitation Protection** | (--) Very Negative. It is trivial for a malicious user to bypass client-side checks, making it completely ineffective for providing real security. | (-) Poor. It fails to proactively protect users, as the prohibited content is delivered first and action is only taken later. | (++) Excellent. It is the only option that fully satisfies the requirement to proactively block prohibited content before delivery. |
| **QA\_09: Chat Message Delivery Time** | (++) Excellent. The check happens on the device before the message is sent, so there is no added network latency. | (++) Excellent. Because the check is asynchronous, there is zero impact on the message delivery time, ensuring a real-time feel. | (-) Negative. It directly adds latency to every single message sent, which can negatively impact the conversational feel of the chat. |
| **QA\_06: Adaptability** | (-) Negative. Any change to the filtering rules requires a new client application release and for users to update their app, making it very slow to adapt. | (++) Excellent. The server-side moderation rules can be updated and deployed at any time without requiring a client-side change. | (++) Excellent. The server-side rules and ML models can be updated and deployed independently at any time. |
| **Decision** | **SELECTED** because it provides an immediate filtering. It may be basic, but even if it can handle 10-15% of the total positive cases, it can actually enhance the system by reducing the average moderation time and also by reducing server traffic. |  | **SELECTED** because it effectively meets the criteria of actually moderating the chat in real sense. The negative point on performance is overlooked considering it is a low prioirty QA. Also, since chat is not the primary feature of the app, slight performance hit on this feature is acceptable. |

* **Risk:** The most critical risk is that the added latency from the synchronous check could make the chat experience feel sluggish, violating QA\_09.
* **Solution:** Mitigate this by engineering the ModerationService for **extremely low latency (p99 < 200ms)**. This will be achieved by using a tiered-check system: fast, cached regex/keyword checks will handle the majority of cases, and only messages that pass this initial filter will be sent to slower, more complex ML models

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** | | | |
|  | **Conflict** | | |
| **CA-28: Rule-Based Engine (SLM)** | **CA-29: Public Pre-trained NLP Model** | **CA-30: Large Language Model (LLM)** |
| **Advantages** | Extremely fast and low-cost; effective for obvious, clear-cut violations. | Good balance of speed and accuracy for general violations; requires no training effort. | Highest accuracy for detecting nuanced, contextual, and cleverly disguised violations. |
| **Disadvantages** | Brittle and easy to bypass; high rate of false positives due to lack of context. | Lacks domain-specific knowledge of the platform's unique business rules. | High cost per API call and high, variable latency, making it risky for a synchronous workflow. |
| **QA\_04: Listing Moderation & Malicious Content Prevention** | (-) Poor. Its inability to understand context makes it ineffective against anything but the most basic attempts to post malicious content. | (+) Good. Effectively handles common problems like general toxicity and PII, but will miss violations specific to the marketplace's rules. | (++)Excellent. Provides the most comprehensive and nuanced validation, offering the best protection for users. |
| **QA\_05 Time taken for a new listing to be visible to the user should be low.** | (++) Excellent. Regex and keyword matching are extremely fast operations with negligible performance impact. | (+) Good. Running a local, optimized NLP model adds a very small, predictable amount of latency. | (--) Very Negative. API calls to a large, external LLM are slow and have variable latency, which would severely impact the listing creation time if used synchronously. |
| **QA\_06: Adaptability** | (-) Poor. Rules are hardcoded. Adding or modifying rules requires code changes and redeployments. | (+) Good. Can adapt to new general threats by swapping in an updated public model, but cannot adapt to new business rules. | (+) Good. Can be adapted to new business rules instantly by simply changing the prompt sent to the LLM, requiring no model retraining. |
| **Decision** | **SELECTED** as the mandatory first tier Moderation. It will be used to instantly filter out the majority of obvious violations at a very low cost. | **SELECTED** Since we are having Synchronous Moderation (mentioned in Use case), where the seller needs to be shown the moderation results, before he is able to submit. We need to have a fast Moderation mechanism |  |

|  |  |  |
| --- | --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** | |
| **Conflict** | |
| **CA-31: Synchronous Dual Writes** | **CA-32: Asynchronous Event-Driven Propagation** |
| **Advantages** | Guarantees immediate data consistency; simple to reason about as the operation is atomic (succeeds or fails completely). | Decouples services, making the system more resilient and scalable; provides a faster initial response to the user. |
| **Disadvantages** | Tightly couples the listing service to the search service; creates a single point of failure; slower response time for the user. | Introduces eventual consistency, meaning there is a small delay before a new listing is searchable; adds complexity. |
| **QA\_05: Listing Visibility Latency** | (++) Excellent. The data is instantly consistent, so the latency is effectively zero. The listing is searchable the moment the API call succeeds. | (--) Negative. This architecture inherently introduces a delay, as the search index is updated in the background. The latency is a key metric to be monitored. |
| **NFR\_04: System Recovery** | (--) Very Negative. A failure or slowdown in the search engine will cause the entire listing creation feature to fail, violating the principle of fault isolation. | (++) Excellent. A failure in the search indexing process does not impact the core ability to create listings. The system is highly resilient. |
| **Impact on user while creating a listing (Performance)** | (-) Poor. The user must wait for two separate network writes to complete, resulting in a slower API response time. | (++) Excellent. The user only waits for a single, fast write to the primary database, resulting in a very fast API response. |
| **QA\_06: Adaptability** | (-) Poor. If a new system (e.g., a recommendation engine) also needs to be notified of a new listing, the ListingController code must be modified. | (++) Excellent. New services can simply subscribe to the existing ListingCreated event without requiring any changes to the ListingController. |
| **Decision** |  | **SELECTED as it provides a superior architecture that is resilient, scalable, and adaptable, which are critical for a distributed microservices system. The negative aspect of QA\_05 is acceptable as it is a medium priority QA.** |

* **Risk:** The most critical risk is a **failure in the data synchronization pipeline**, causing the search index to become stale or inconsistent with the primary database. A user might see a sold item in search, or not see a newly created item for a long time.
* **Solution:** Mitigate this by implementing a **Dead-Letter Queue (DLQ)** for the message bus. If the Search Indexer service fails to process a message after several automated retries, the message is moved to the DLQ. This triggers an immediate alert (QA\_11) for an engineer to investigate the specific failure without losing the update event, ensuring data can be reconciled later.

|  |  |
| --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** |
| **CA-33: Listing Meta Data Enhancement using LLM** |
| **Advantages** | Significantly enhances search relevance (QA\_01) by creating rich, contextual metadata that improves discovery. The process is asynchronous, adding no latency to the seller's listing creation flow. |
| **Disadvantages** | Introduces high operational costs due to per-API call pricing for LLMs. There is a risk of generating irrelevant or inaccurate tags, and it adds complexity with a new background service. |
| **QA\_01: Search Relevance** | (+++) Superior. Creates rich data that allows for powerful semantic and faceted search, directly improving a user's ability to find what they want. |
| **QA\_05: Listing Visibility Latency** | (-) Negative. The process may take some time to complete, hence causing a delay in it to be visible. |
| **Cost Efficiency** | (--) Very Negative. LLM API calls are expensive, and running this process for every new listing represents a significant and ongoing operational cost. |
| **QA\_06: Adaptability** | (++) Excellent. The logic can be easily modified by changing the prompt sent to the LLM, allowing for rapid adaptation to new business needs without code changes. |
| **Decision** | **SELECTED** as this improves the search relevancy a lot. Since this operation happens only once at the time of listing, cost of this step and also the latency introduced by this step can be ignored as compared to the relevancy it brings. |

* **Risk:** The most significant risk is the high, potentially unpredictable **operational cost** from making an LLM API call for every new listing. A secondary risk is the generation of inaccurate or irrelevant tags, which could pollute search results and negatively impact user trust.
* **Solution:** To keep costs down, we're not going to use the big, expensive AI on every single item. Instead, since our use case is not very tough, we will have a smaller, cheaper AI alternates. Also, to make sure the tags it creates are actually any good, we'll check how "sure" the AI is about its suggestions. We will also keep an eye on things and run A/B tests to see if these generated tags actually get more people to click on the listings.

|  |  |  |
| --- | --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** | |
| **Conflict** | |
| **CA-34: Synchronous, Server-Side Upload and Processing** | **CA-35: Asynchronous, Decoupled Media Pipeline** |
| **Advantages** | Simple to implement as it follows a traditional, linear request-response model. | Highly scalable and resilient; decouples the core API from heavy media processing, freeing up server resources; enables parallel uploads for a better user experience. |
| **Disadvantages** | Creates a performance bottleneck; ties up server resources for long-running I/O operations; scales poorly under concurrent uploads; single point of failure. | Higher initial implementation complexity, requiring management of pre-signed URLs and a separate, event-triggered processing service. |
| **QA\_08: Listing Media Processing Time** | (--) Very Negative. The user is blocked and must wait for the entire upload and processing pipeline to complete, resulting in a slow and frustrating experience. | (++) Excellent. The user experiences a very fast initial response as the large file upload bypasses the main server. The processing happens in the background. |
| **NFR\_03: Service Scalability** | (--) Very Negative. The API server becomes a bottleneck as it is responsible for handling large file transfers, which does not scale effectively. | (+++) Superior. The use of direct-to-cloud storage uploads and serverless functions for processing creates a highly scalable and cost-effective architecture. |
| **Decision** |  | **SELECTED** because it provides a superior architecture that is resilient, scalable, and adaptable, which are critical for a distributed microservices system. The negative aspect of QA\_05 is acceptable as it is a medium priority QA. |

* **Risk:** The primary risk is a failure in the asynchronous pipeline. A user could successfully upload an image, but the background serverless function might fail to process it, leaving the listing with a missing or broken image and no feedback to the user.
* **Solution:** This risk will be mitigated by implementing a robust monitoring and retry mechanism for the serverless function. If processing fails after several automatic retries, the event will be moved to a **Dead-Letter Queue (DLQ)**. This will trigger an alert for the engineering team to investigate while also updating the listing's status to "Media Processing Failed," which can be communicated back to the seller on their "My Listings" page.

|  |  |  |  |
| --- | --- | --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** | | |
| **Conflict** | |  |
| **CA-36: Hardcoded Logic in Controller** | **CA-37: Strategy Design Pattern** | **CA-38: A/B Testing Microservice** |
| **Advantages** | Simplest to implement for the very first algorithm; no initial design overhead. | Decouples algorithm logic from the client, making code cleaner and more testable. New algorithms can be added without modifying existing code. | Allows for safe, gradual rollouts and data-driven comparison of algorithms on live traffic, minimizing risk and enabling data-driven decisions. |
| **Disadvantages** | Extremely rigid and brittle; every change requires modifying and redeploying a core service, which is high-risk and high-effort. | Does not provide a mechanism to test or compare algorithms on live users; switching strategies typically requires a redeployment. | Highest architectural complexity; requires building and maintaining a separate service for user bucketing and metric collection. |
| **QA\_06: Adaptability** | (--) Very Negative. This approach completely fails the requirement, as it is the definition of an unadaptable system. | (++) Excellent. This is a classic object-oriented solution. Adding a new algorithm is a low-effort task of creating a new class that implements a stable interface. | (+++) Superior. It provides a framework for not just adding new algorithms, but for safely validating their real-world performance before full deployment. |
| **QA\_02: Recommendation Usefulness** | (NA) Neutral. Its rigidity makes it very difficult to improve usefulness over time. | (+) Good. It facilitates experimentation by making it easy to swap algorithms, which is a prerequisite for improving usefulness. | (+++) Superior. This is the only approach that enables the scientific measurement and optimization of recommendation usefulness (e.g., via Click-Through Rate) in a controlled manner. |
| **Risk of Deploying a New Algorithm** | (High) A bug in a new algorithm is deployed to 100% of users and can break the entire feature. | (Medium) A new algorithm is still rolled out to all users at once. If it provides poor recommendations, the experience is degraded for everyone until it is rolled back. | (Low) The impact of a poorly performing algorithm is contained to a small, controlled percentage of users, minimizing negative business impact. |
| **Decision** |  | **SELECTED** as adding a new algorithms by having Strategy pattern is a good design pattern to have. | **SELECTED** as through this design we can parallelly run 2 algorithms, and then smoothly transition from old to new |

* **Risk:** The primary risk is the potential for high architectural complexity. As the system becomes more distributed with the addition of a new microservice for A/B testing, it can be difficult for engineers to understand the complete end-to-end flow. This makes debugging challenging and creates a steep learning curve for new team members joining the project.
* **Solution:** To mitigate this risk, the A/B testing system will be treated as a first-class internal platform. This approach includes establishing strong governance and creating comprehensive documentation. We will implement a robust CI/CD pipeline to automatically validate any new experiment configurations before deployment. Furthermore, we will invest in observability tools to trace requests throughout the system and build real-time dashboards to monitor experiment health, allowing us to quickly halt any experiment that shows a negative impact.

|  |  |  |  |
| --- | --- | --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** | | |
| **CA-39: Rule-Based Engine (for seller Ratings)** | **CA-40: Event-driven ML Scoring** | **CA-41: User-to-User Rating System** |
| **Advantages** | Instant, predictable, and automated risk scoring. Simple to implement for clear-cut signals. | Learns and adapts to new fraud patterns automatically. Provides a dynamic, context-aware risk score. | Builds long-term community trust. Provides a strong reputation signal based on actual user experiences. |
| **Disadvantages** | Rules are brittle and easy to bypass; becomes complex to maintain. | Higher initial development complexity; risk of "cold start" problem for new users. | Reactive, not proactive. A new fraudulent seller will have no rating. Cannot prevent the first few instances of fraud. |
| **QA\_07: Real-time Seller Risk Scoring** | (+) Good. Provides an instant, automated score for known patterns, satisfying the real-time requirement for basic cases. | (++) Excellent. It is designed to analyze behavioral signals and update risk scores in near real-time as events happen. | (-) Poor. This is a historical, not real-time, indicator. It provides no signal for a new seller's first actions. |
| **QA\_06: Adaptability** | (-) Poor. Adapting to new fraud patterns requires engineers to write and deploy new code for the rules. | (++) Excellent. The ML model can be retrained on new data to automatically learn and adapt to emerging fraud patterns. | (+) Fair. The system adapts based on user feedback but does not proactively identify new types of fraud schemes. |
| **NFR\_03: Service Scalability** | (++) Excellent. A stateless rule engine is simple and can be scaled horizontally with ease. | (++) Excellent. The architecture is based on processing an event stream and is designed for high-throughput, scalable analysis. | (++) Excellent. Aggregating user ratings is a simple database operation that scales easily. |
| **Decision** |  | **SELECTED** as the primary strategy. It is the only solution that provides a dynamic, adaptive, and scalable approach to real-time fraud detection, fully satisfying QA\_07. | **SELECTED** as a complementary feature. It is essential for long-term trust but does not address the real-time requirement. |

* **Risk**: The primary risk of a pure **ML Scoring (CA-40)** approach is the **"cold start" problem**. A new, legitimate seller will have no behavioural history, making it difficult for the model to distinguish them from a new fraudster. This could lead to high false positives, frustrating good users.
* **Solution**: In case of Cold Start, the **User-to-User Rating (CA-41)** score will serve as a critical input feature for both the rule-based engine and the ML model, providing a long-term reputation signal.

|  |  |  |  |
| --- | --- | --- | --- |
| **Quality Requirement** | **Effect of Candidate Architecture** | | |
| **CA-42: Basic Cloud Metrics and Logs** | **CA-43: Centralized Logging & Metrics** | **CA-44: Distributed Tracing** |
| **Advantages** | Zero initial setup cost or complexity. | Provides a "single pane of glass" for all logs and metrics. Enables powerful querying, dashboarding, and proactive alerting. | Pinpoints the exact source of latency or errors in a complex, multi-service request. Drastically reduces time to diagnose issues. |
| **Disadvantages** | Extremely slow and inefficient for debugging; not scalable for a microservices architecture; purely reactive. | Adds operational overhead to manage the monitoring stack; can't easily show the causal relationship between events in different services. | Higher implementation complexity, as it requires code instrumentation in every service; can add minor performance overhead. |
| **QA\_10: Monitoring and Alerting** | (--) Very Negative. Fails to meet the requirement as it provides no mechanism for proactive, application-level alerting. Mean Time To Detect (MTTD) is extremely high. | (++) Excellent. Directly provides the foundation for monitoring key health metrics and triggering automated alerts when thresholds are breached. | (+++) Superior. Enhances alerting with deep context. Instead of just knowing a service is slow, you know which downstream call is making it slow, enabling more intelligent and actionable alerts. |
| **NFR\_04: System Recovery** | (-) Negative. The long time required to diagnose a problem directly increases the Mean Time To Recover (MTTR), leading to longer outages. | (+) Good. By drastically reducing MTTD, it enables a much faster start to the recovery process, thus improving MTTR. | (++) Excellent. Provides the fastest possible root cause analysis, minimizing diagnosis time and leading to the lowest possible MTTR. |
| **Decision** |  | **SELECTED as the essential foundation. Centralized data is a non-negotiable requirement for observability.** | **SELECTED as a critical layer on top of CA-43. For a distributed system, tracing is the only effective way to debug and understand system performance.** |

* **Risk**: The primary risk is the **operational complexity and cost** of managing a full observability stack (logging, metrics, tracing). This can become a significant engineering effort and can incur high data ingestion and storage costs at scale. A secondary risk is the performance overhead from instrumentation.
* **Solution:**

1. **Use Managed Services**: Mitigate operational complexity by leveraging managed cloud services (e.g., AWS CloudWatch, AWS X-Ray) or unified third-party observability platforms (e.g., Datadog, New Relic). This abstracts away the complexity of scaling and managing the underlying infrastructure.
2. **Implement Intelligent Sampling**: Mitigate cost and performance overhead by implementing sampling for distributed traces. Instead of tracing every request, the system will trace a representative percentage (e.g., 10% of all successful requests) and 100% of requests that result in an error. This provides the necessary diagnostic data while significantly reducing the volume of data generated and its associated costs.
3. Final Architecture

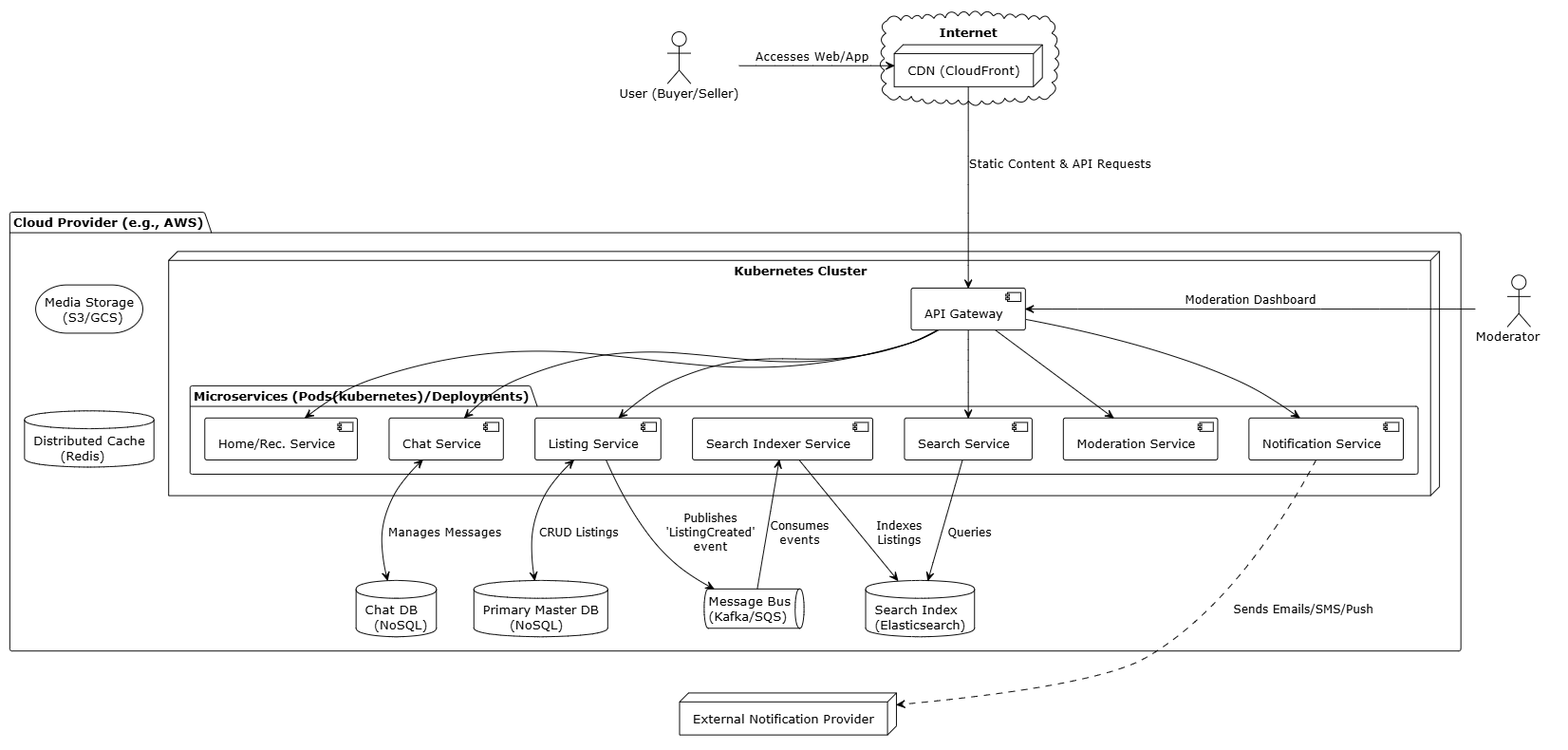
// A7. Architecture Design

// C7-3. Is there right integration into the final architecture?

// C7-4. Is there appropriate risk management of the final architecture?

**F1.1 Deployment View**

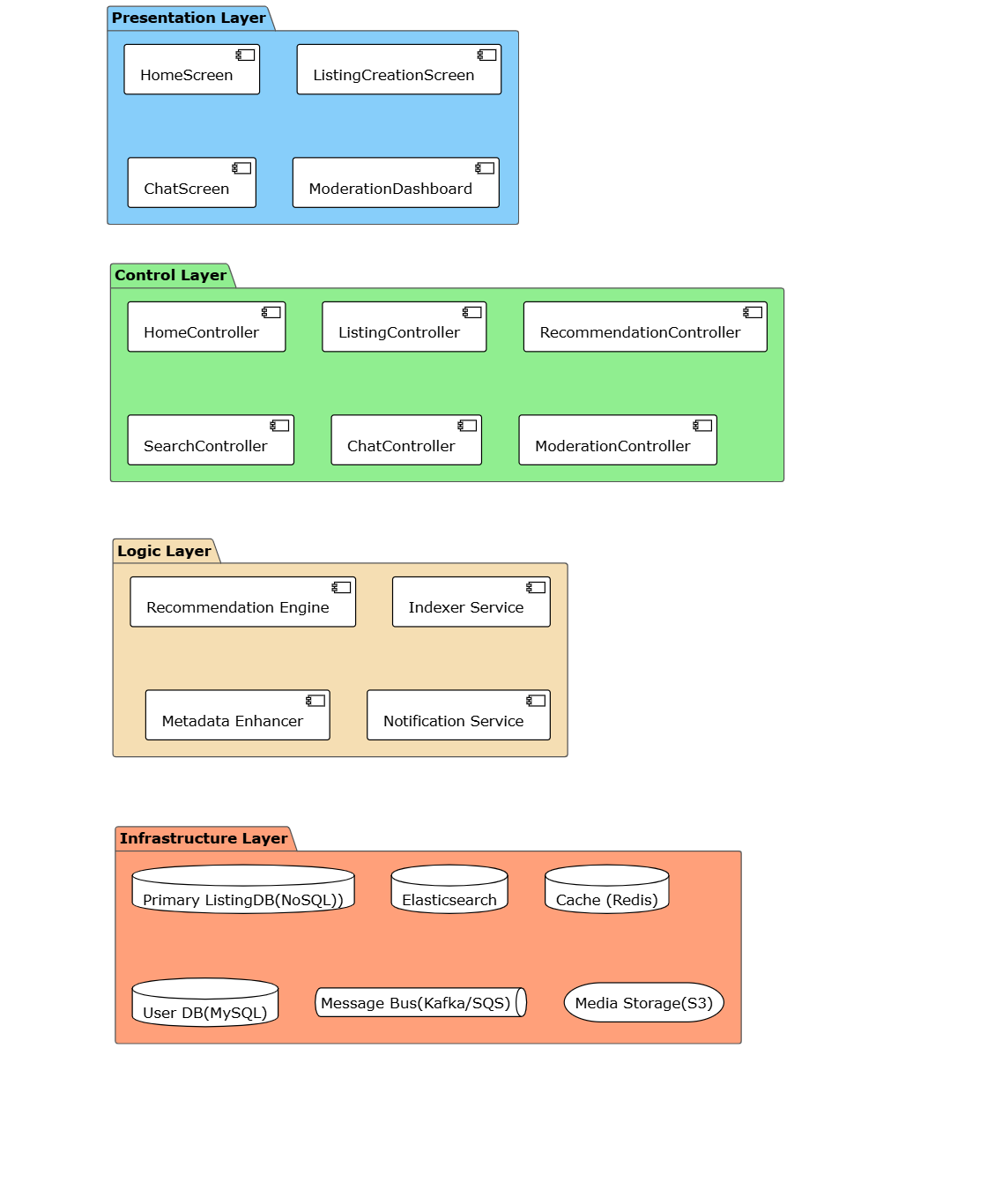
**Overview.** The marketplace is deployed as a cloud-native, Kubernetes-orchestrated system with an edge tier (DNS/CDN/WAF/API-Gateway), a services tier (microservices in K8s), and managed data plane (RDS/PSQL, Redis, Kafka/SNS+SQS, S3/GCS, OpenSearch/ES, Warehouse). Services scale horizontally and communicate synchronously via REST/WebSockets and asynchronously via an event bus for decoupling and resilience.



**Figure: Deployment View for C2C MarketPlace**

**F1.2 Module View**

The system's modules are organized into a four-layer architecture to ensure a clear separation of concerns and promote modifiability. The **Presentation Layer** contains all UI components, while the **Control Layer** handles incoming user requests. The **Logic Layer** executes core business processes, and the **Infrastructure Layer** manages data persistence and external services.



**Figure: Module View for C2C MarketPlace**

**F1.3 Risks**

**1. Data Inconsistency due to Asynchronous Processing**

* **Risk**: Our extensive use of an **event-driven architecture** for decoupling services introduces a primary risk of data inconsistency. A failure in the asynchronous pipeline could lead to the search index becoming stale (e.g., a sold item still appears in search results), product images failing to process after upload, or cached data becoming outdated.
* **Mitigation Plan**: This risk will be managed by implementing a **Dead-Letter Queue (DLQ)** for the message bus. If a service fails to process an event after several retries, the message will be moved to the DLQ, triggering an immediate alert for investigation without losing the update. For caching, we will supplement this with an event-driven cache invalidation strategy and short Time-to-Live (TTL) policies.

**2. Cascading Failures in Microservice Dependencies**

* **Risk**: In a distributed system, a single slow or failing downstream service can block upstream callers, leading to resource exhaustion and a chain reaction of failures that can bring down the entire platform.
* **Mitigation Plan**: We will implement the **Circuit Breaker pattern** to isolate failures. The primary risk of this solution is misconfiguration (thresholds being too sensitive or too lenient). This will be mitigated by using dynamic, adaptive thresholds based on real-time service performance and by making the state of all circuit breakers a key metric in our monitoring dashboards.

**3. Algorithmic Bias Creating an Unfair Marketplace**

* **Risk**: The decision to use **Static Business-Rule Re-ranking** to boost listings from highly-rated sellers creates a significant risk of algorithmic bias. This can lead to a "rich-get-richer" feedback loop where new sellers are perpetually buried, stagnating inventory diversity and harming the platform's long-term health.
* **Mitigation Plan**: To counteract this bias, the re-ranking logic will include rules that provide temporary **"new seller" or "first listing" boosts**. This gives new participants a fair chance at gaining initial visibility. We will also actively monitor marketplace fairness metrics to ensure the ecosystem remains healthy and competitive.

**4. Real-time Chat Unreliability and Disconnections**

* **Risk**: Using a **Pub/Sub pattern with persistent WebSocket connections** introduces the risk of managing a large number of stateful connections. If a server node handling these connections fails, all connected users will be abruptly disconnected, leading to a poor user experience and potentially lost messages.
* **Mitigation Plan**: This will be mitigated by building a **resilient WebSocket gateway layer** coupled with robust client-side reconnection logic. Client applications will be designed to automatically re-establish their connection to a new healthy node and resynchronize their chat history, ensuring the experience is seamless and no data is lost.

**5. High Latency in Chat due to Synchronous Moderation**

* **Risk**: The decision to use a **Synchronous, Real-time Moderation Gateway** for chat messages introduces a critical risk of adding latency to every message sent. This could make the chat experience feel sluggish and unresponsive, violating a key quality attribute for real-time communication (QA\_09).
* **Mitigation Plan**: We will mitigate this by engineering the ModerationService for extremely low latency. This will be achieved using a **tiered-check system**, where fast, cached keyword and regex checks handle most cases, and only ambiguous messages are passed to slower, more complex machine learning models.

**6. High Operational Costs from Advanced AI Features**

* **Risk**: Implementing **Listing Metadata Enhancement using an LLM** introduces a significant and potentially unpredictable operational cost due to the per-API call pricing of large language models. A secondary risk is the generation of inaccurate or irrelevant tags that could pollute search results.
* **Mitigation Plan**: To manage costs, a **tiered enrichment strategy** will be implemented. A smaller, cheaper model will handle initial analysis, and only high-value or ambiguous listings will be escalated to the more expensive LLM. To ensure quality, generated tags will be validated against a confidence score and their impact will be measured via A/B testing.

1. Architecture Evaluation(ATAM)

// A10. Architecture Evaluation

// C10-1. Are there sufficient quality scenarios evaluating architecture?

// C10-2. Are there sufficient architectural decisions identified?

// C10-3. Is the analysis of design decisions appropriate? (evidence)

// C10-4. Are the mitigation plans to the risk factors appropriate?