|  |
| --- |
| **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?

## 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.

## System Definition

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

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

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

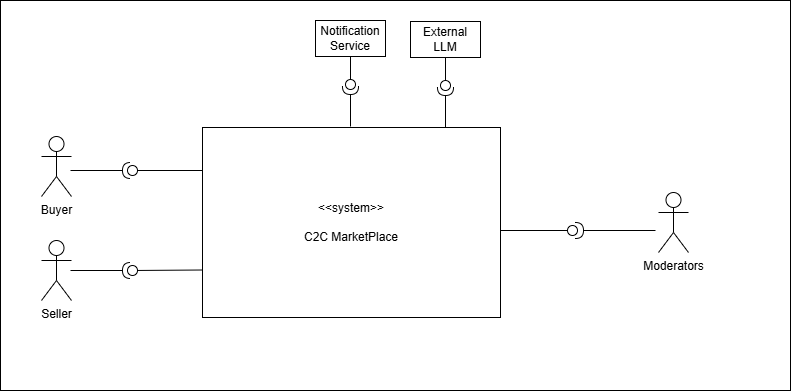


Figure 1: **System boundary for C2C Marketplace**

## Business Context & Drivers

This section outlines the current market environment and specifies the core business objectives that mandate the architectural design choices for the new Customer-to-Customer (C2C) marketplace platform.

### 1.3.1. Business Context: Optimizing the Core User Journey

In a highly competitive re-commerce landscape, our primary business challenge is to create an ecosystem where the user is able to find the most relevant listings according to their needs, and their journey from discovery to final purchase is not only possible, but also intuitive and secure. Success depends on maximizing the efficiency of both **product discovery for buyers** and **listing creation for sellers**.

### 1.3.2. Business Drivers

The following drivers are the non-negotiable drivers guiding the platform's service architecture, focused exclusively on the core experience components: listing, search, and recommendations.

* **Superior Product Discovery and Personalization** to help buyers find what they want quickly and effortlessly. This will be achieved by using **advanced search capabilities** and delivering **personalized, relevant recommendations** directly on the home page and category feeds to maximize click-through and purchase conversion rates.
* **Seamless Listing Experience and Quality** to increase seller on boarding rates and listing quality by making the creation and publishing process effortless. The listing journey must be smooth, guided and feature helpful automations to reduce seller’s effort and the time-to-publish.

## 1.4 Constraints

This project

# 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?

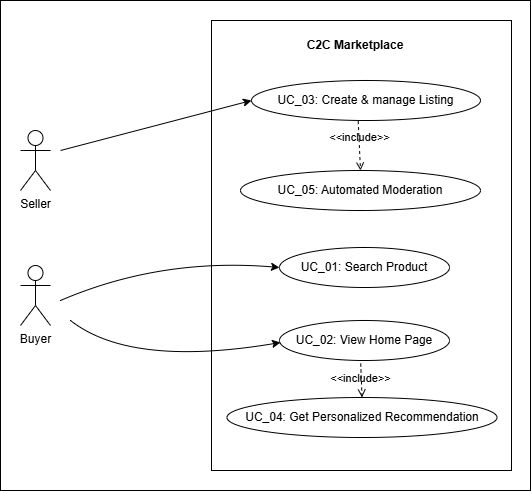


Figure 2: C2C Marketplace - Use Case Diagram

|  |  |
| --- | --- |
| **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. System **dynamically determines and presents** additional fields relevant to the selected product category, utilizing a **configurable attribute schema** to support diverse item types and future category additions. For example:   * If user selects the **"Cars"** category, the system presents fields for Make, Model and Year of manufacture. * If the user selects the **"Mobile Phones"** category, the system instead presents fields for Brand, Storage Capacity, and Battery Health.   4. User then completes these category-specific fields.  5. System runs content moderation checks on all the filled-in data and confirms it passes the validation. (AF2)  6. System presents a preview of the listing showing the user how their listing will appear to other users.  6. User reviews and validates the data in the preview, 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 indicates the specific issue to the Seller and offers potential solutions.  3. User can then either correct the details & resubmit the listing information, 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 submits a search request to the system with keywords and optional filters.  2. System validates the request and executes a query against the dedicated search index, using keyword matching, semantic search, and filters to retrieve initial listings ranked by a core relevance score.  3. System retrieves supplementary data (e.g., listing date, seller rating) for the initial results and applies defined business rules to adjust the ranking based on factors like:   * Geographic Proximity * Listing Recency * Seller Reputation   4. System identifies relevant advertisements and merges them into the re-ranked listing results according to placement strategies.  5. System prepares and returns the final, merged, and ranked list (including pagination information) to the user. (AF1) |
| Additional Flow | AF1: No Results Found: This flow occurs at Step 5 if no listings match the Buyer's search criteria. The system communicates, "No results found".  To be helpful, the system also suggests alternative actions, such as checking for spelling errors, removing filters, or it may present popular items from a similar category. |

|  |  |
| --- | --- |
| **UC\_03** | View Home Page |
| Description | This use case describes what happens when a user initiates a session with the system’s main 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 assemble and provide the main interface along with the recommendation data. |
| Basic Flow | 1. User initiates a session with the main 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 presents the main interface content, prioritizing personalized content and ensuring search feature is readily accessible. |
| 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 accessing the main interface or viewing details of a specific product. |
| Actor | User (Buyer) |
| Pre-condition | User is logged in. |
| Post-condition | System presents a set of relevant personalized items to the user. |
| Basic Flow | Scenario A: Recommendations on the Home Page  1. Trigger: User initiates a session with the main interface.  2. System analyzes the Buyer's profile, recent activity (views, searches), and location to understand their intent.  3. System generates and presents several personalized feeds, such as "Recommended For You," "Because You Viewed X," and "New Items Nearby." (AF1)  4. System also considers relevant targeted ads in the system and creates a final curated list of recommended items.  5. System displays the recommendations to the user.  Scenario B: Product Detail Page Recommendations  1. Trigger: User requests details for a specific product.  2. System analyzes the current product's attributes (category, brand, price).  3. System generates and presents contextual recommendations designed to increase conversion, such as "Similar Items" (alternatives at different price points) and "Complementary Items" (accessories or related products).  4. System presents these related Items to the user. |
| Additional Flow | AF1. Cold start: For new users, as a fallback, system uses popular listings of user’s location. |

|  |  |
| --- | --- |
| **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 establishes a secure communication channel between the 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 directed to the correct screen. |
| Basic Flow | 1. User accesses the notification settings feature within the account settings.  2. System presents 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. System 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 directs the user to the relevant section. (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 should be low** |
| 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 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 | 1. System displays the paginated list of search results to the user.  2. User visually scans the first page of results.  3. User identifies a relevant item within the top N (e.g., 5) results.  4. 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** | Home Page Relevancy |
| 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 | 1. System renders the "Recommended For You" carousel on the home page.  2. User scrolls or visually scans the items in the carousel.  3. User identifies a compelling item in the list.  4. 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** | **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\_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\_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)

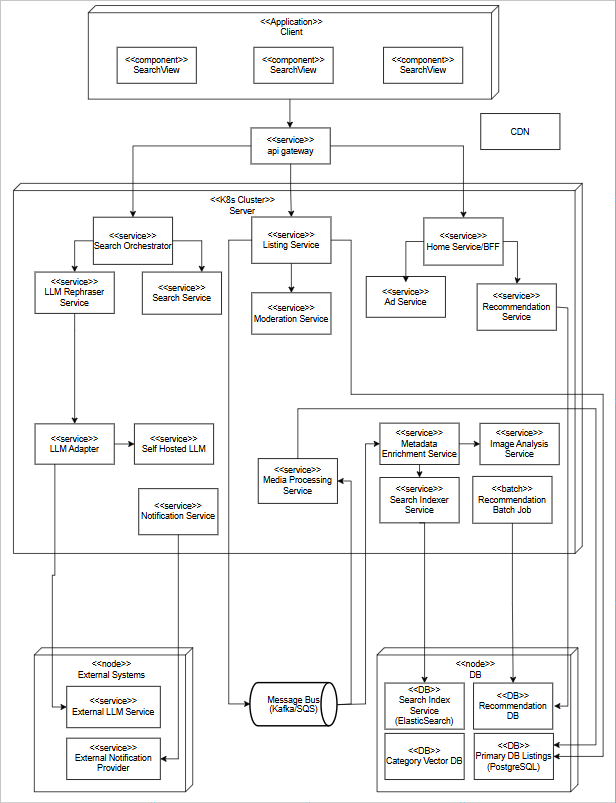
// C8-3. Is the description of System architecture appropriate?

This section details the physical and logical architecture of the C2C Marketplace system, including the deployment of services and the high-level strategies for component interaction.

## 3.1. Deployment View

The Deployment View represents the hardware and software structure of the C2C Marketplace, which is designed based on a Kubernetes (K8s) cluster and a Microservice Architecture (MSA) Style.

The system is organized into logical nodes, each containing services and resources grouped by their core function, such as user-facing features, listing/search logic, and data persistence.



# 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?

## Design Overview

This system is designed with a 4-Layer-based hierarchical module structure (Layered Architecture Style) to clearly separate complex listing creation, adaptive search, and home page loading with recommendation flows and to independently manage each responsibility unit.

The entire module structure is dependent from the higher layer to the lower layer, with a unidirectional flow, enabling flexibility in changes, separation of development responsibilities, and reusability.

Each layer has the following responsibilities:

* **Presentation Layer**: Includes UI components (e.g., HomeScreen, ListingCreationScreen) that directly interact with users and is responsible for forwarding user requests to the Control Layer.
* **Control & Coordination Layer**: This layer coordinates user request flows and directs the execution of Use Cases. Components such as HomeController, SearchOrchestrator, and ListingController act as orchestrators, calling domain logic components and mediating communication with other services.
* **Domain Logic Layer**: This layer runs the core business logic of the system (e.g., recommendation algorithms, search re-ranking, moderation rules, and data enrichment logic). It includes components like the Recommendation Engine, LTR Re-ranking Service, and LLM Rephraser Tool, focusing on high-level abstraction and domain-specific processing logic.
* **Infrastructure Layer**: This layer provides the technical foundation for system operations. It contains the application-side components (such as repositories, clients, and adapters) responsible for integrating with external systems, data storage, and cache management. It handles data persistence by communicating with the Primary Listing DB, asynchronous messaging via the Message Bus, and caching through the Distributed Cache.

This layer separation aims to achieve quality attributes defined at the design stage, such as **performance (NFR\_01), adaptability (QA\_06), reliability (NFR\_04), and maintainability**. It also clearly defines dependencies between layers to serve as a basis for module-level maintenance and assignment of development responsibilities.

<<layered View Diagram >>

## Work Assignment

To promote efficient development, the C2C Marketplace system's modules are grouped according to functional responsibilities, with each group assigned to a distinct organizational unit. This project anticipates involving five development teams, each taking ownership of specific modules arranged within a defined hierarchical structure that follows the primary functional flows of the application.

The following table outlines the designated roles for each team, their corresponding module responsibilities, and how these technical areas connect to the system's business functions. Organizing the work this way supports parallel development efforts, defines clear points of collaboration and integration between teams, and establishes distinct ownership for ongoing maintenance and future feature expansion.

|  |  |  |
| --- | --- | --- |
| **Organization Name** | **Responsible Module Group** | **Responsibilities and Assignment Reason** |
| **UI Development Team** | Presentation Layer Module Group | - Responsible for the entry UI (View) that directly interacts with users, ensuring that user requests are properly delivered to the Control Layer. - Assigned independent responsibility for screen composition to optimize UX. |
| **Core Development Team** | Control & Coordination Layer (Partial), Listing Module Group (Partial) | - Plays a central role in coordinating core use-case flows such as listing creation and home page aggregation.  - Manages foundational services like Listing Service, Home Service, connecting domain logic and infrastructure. |
| **AI Intelligence Team** | Search Module Group, Recommendation Module Core Logic, Enrichment Module Group | - Implements the core "Easy Discovery" business driver. - Responsible for the complex adaptive search logic (Search Orchestrator), all enhancement tools (LTR Service, LLM Rephraser), and the recommendation engine. |
| **Database & Infrastructure Team** | Infrastructure Layer Module Group | - Responsible for managing and maintaining all data infrastructure (Primary DB, Search Index, Cache, Message Bus). - Manages all the infra & Security activities across the project. |

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.

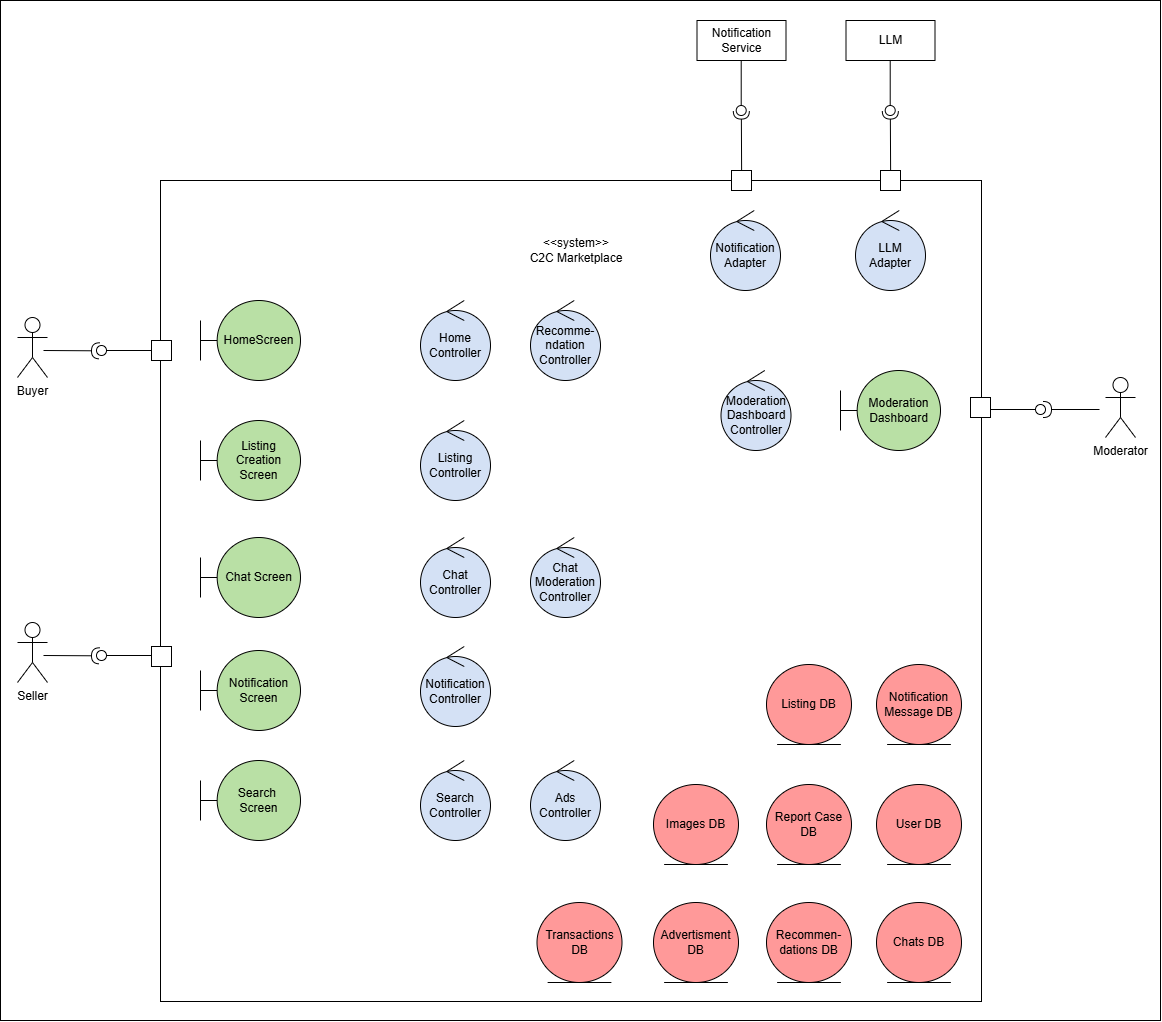


Figure 10: Domain Model for C2C Market Place

### 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. Include search feature and recommendation
* **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.
* **SearchScreen**: The UI used by a user to search for what they are looking for.
* **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.
* **ModerationDashboardController**:
* **ChatModerationController**:
* **AdsController**:
* **NotificationAdapter**: A specialized controller that translates internal notification requests into the format required by external services.
* **LLMAdapter**

### 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.
* **RecommendationsDB:**
* **MessageDB**: Represents a single communication sent between a Buyer and a Seller.
* **ReportCaseDB**: An entity created to track a user-reported issue.
* **ChatsDB**:
* **NotificationMessageDB:**
* **ImagesDB:**
* **UserDB:**
* **TransactionsDB:**

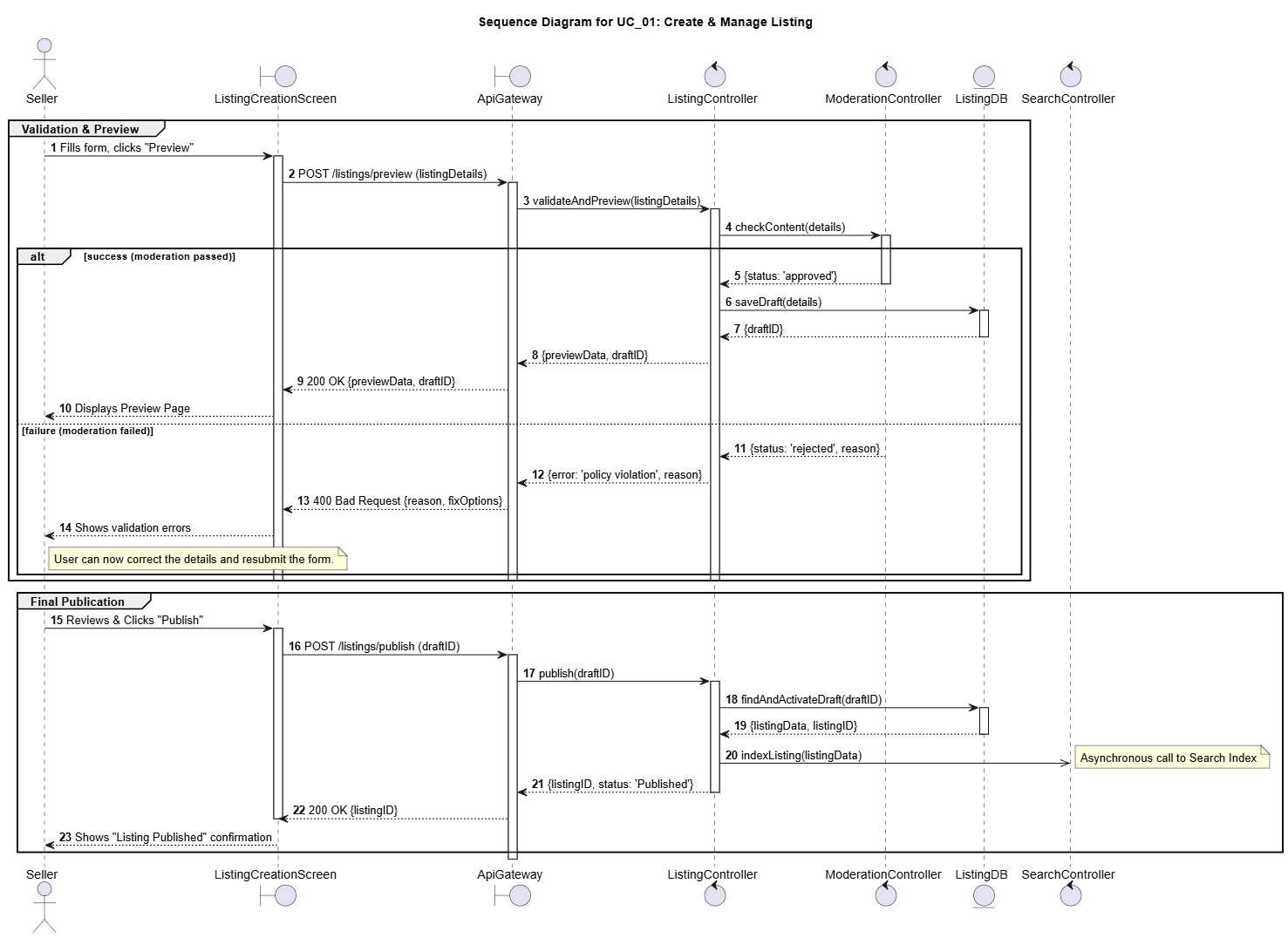


Figure 11: Sequence Diagram for UC01: Create & Manage Listing

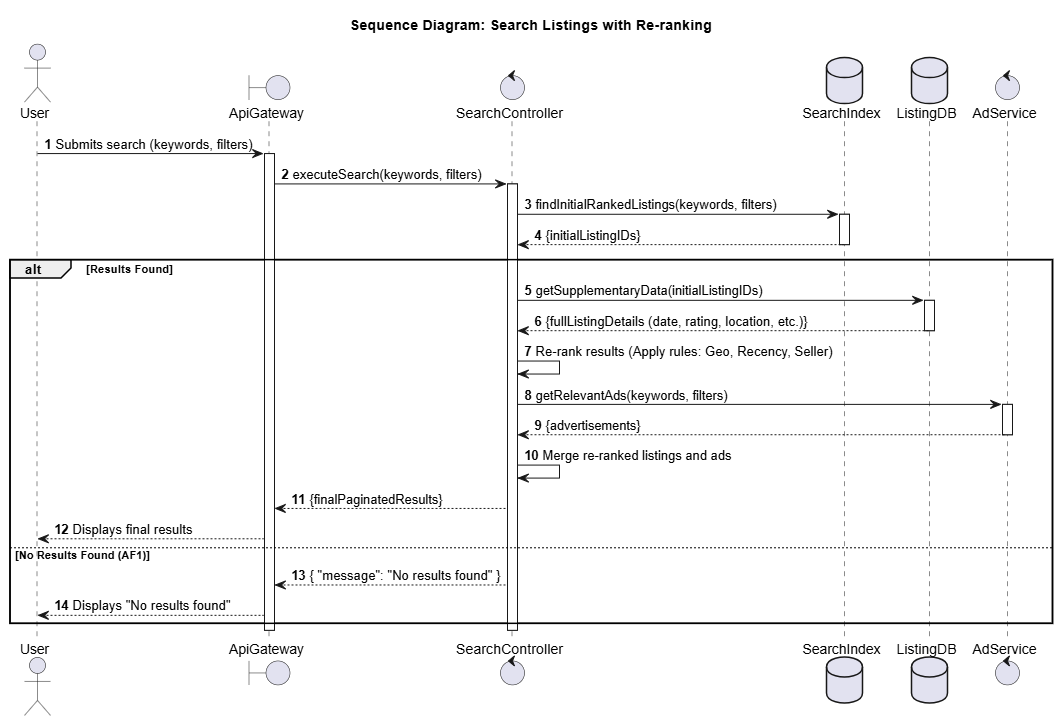


Figure 12: Sequence Diagram for UC02: Search Product

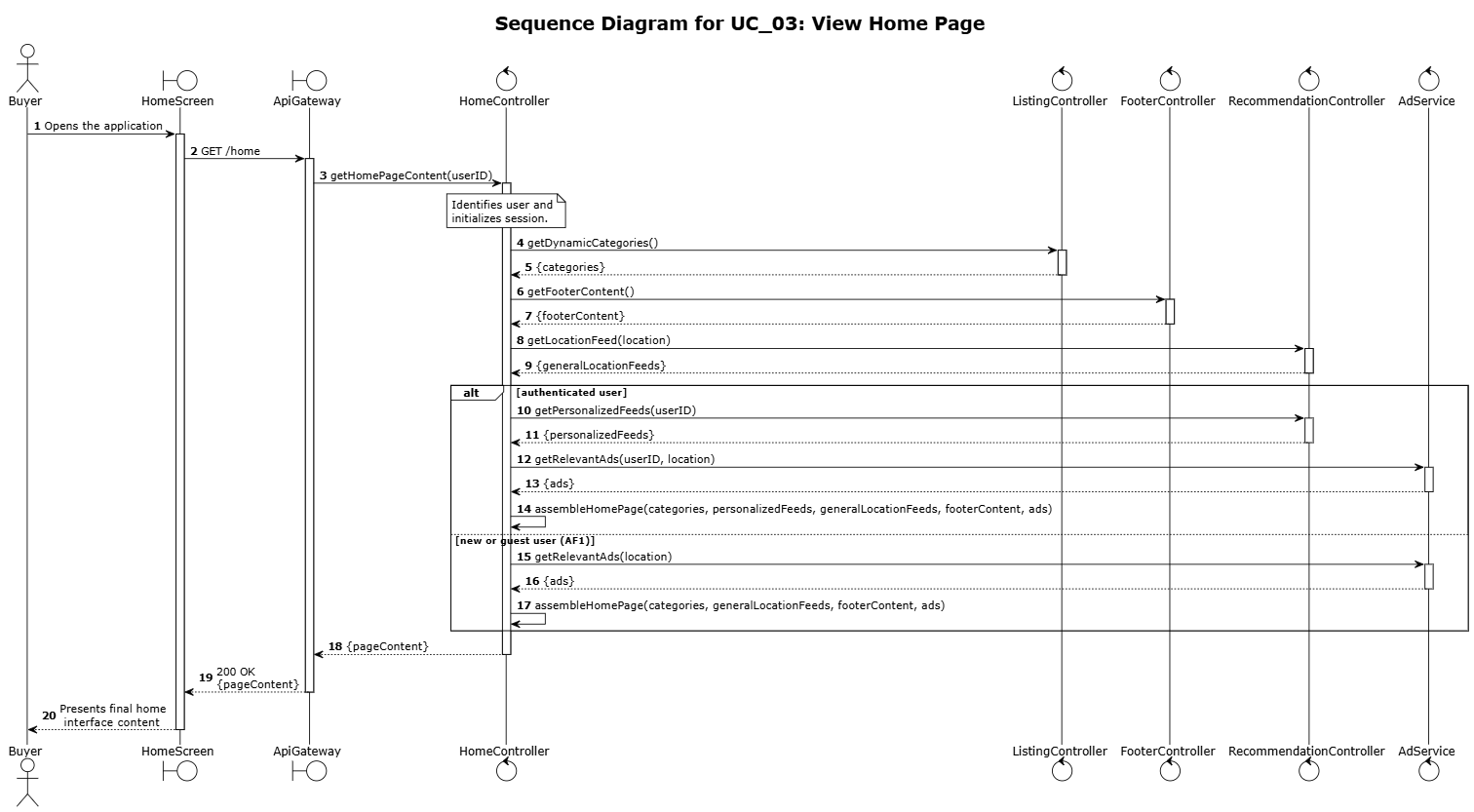


Figure 13: Sequence Diagram for UC03: View Home Page

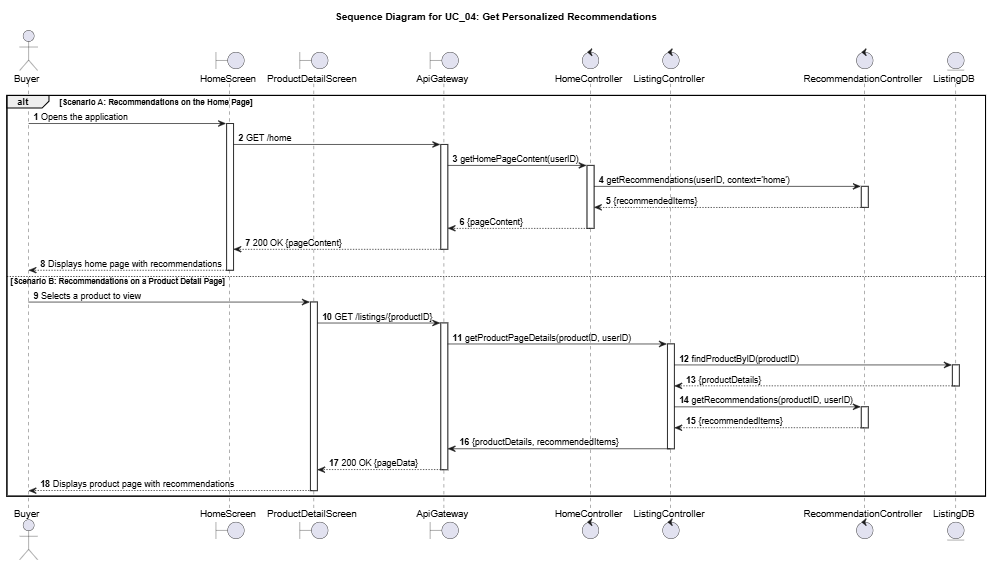


Figure 14: Sequence Diagram for UC04: Get Personalized Recommendations

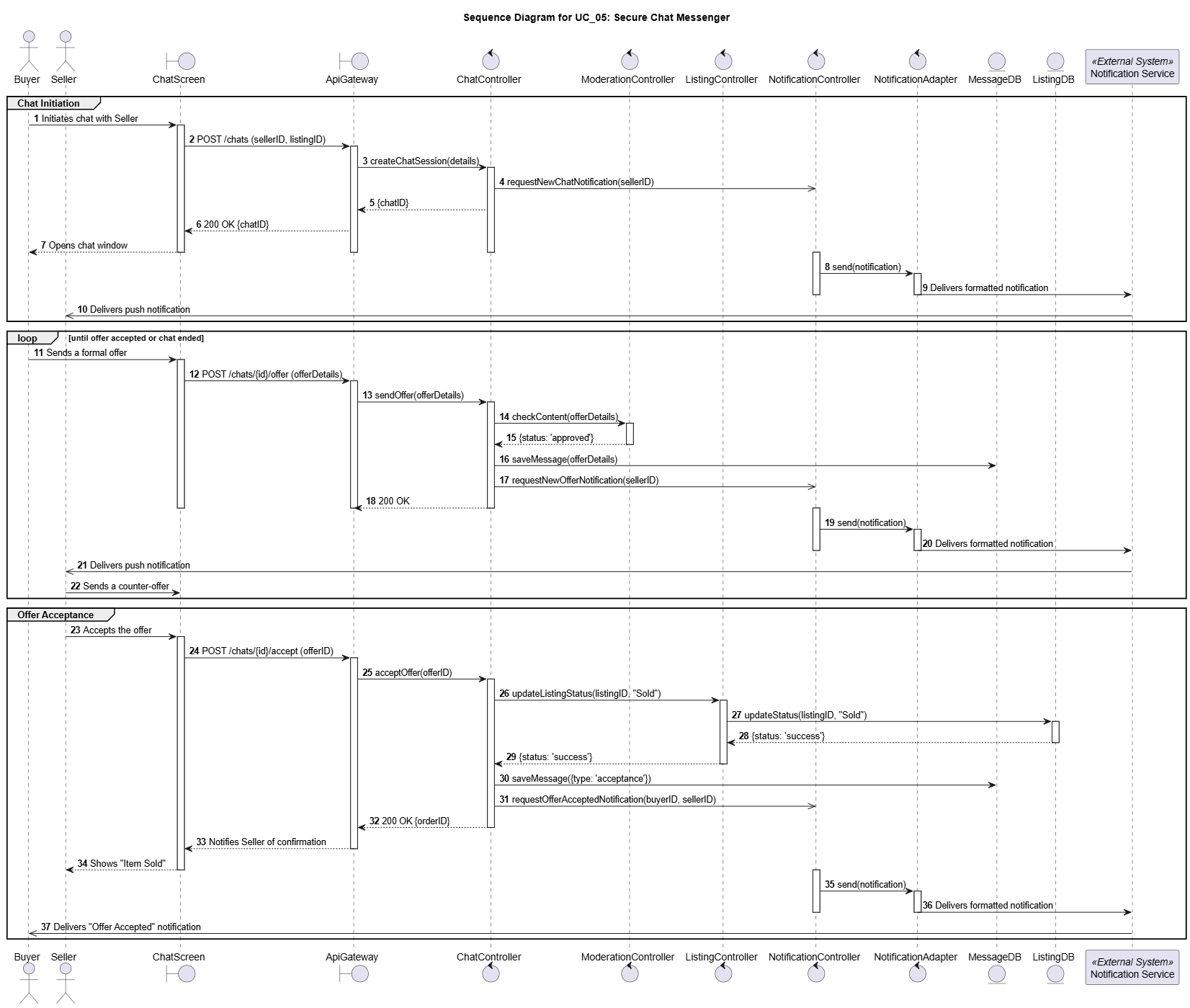


Figure 15: Sequence Diagram for UC05: Secure Chat Messenger

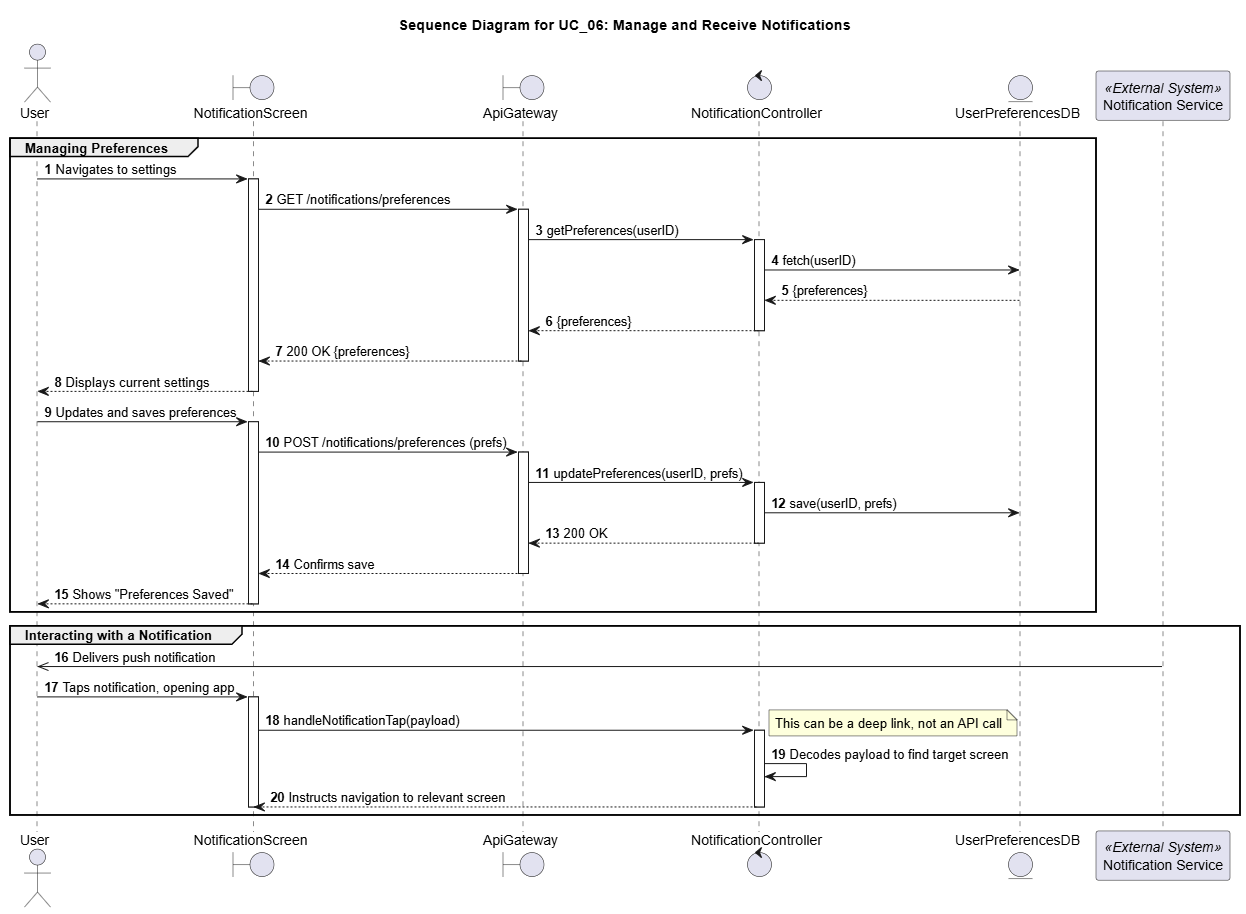


Figure 16: Sequence Diagram for UC06: Manage and Receive Notifications

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?

The following quality scenarios were identified through a combination of analyzing the core business drivers, reviewing common architectural concerns for C2C marketplaces, and conducting stakeholder interviews [representing Users (Buyers & Sellers), Business Stakeholders and Technical Stakeholders].

**B1. QAW (Quality Attribute Workshop)**

**B1.1. User Perspective**

* "The search should understand what I'm *really* looking for, even if my search terms aren't perfect or exact." *(Buyer)*
* “Listing has to be super easy. Upload photos, add important details, and it should appear fast.” *(Seller)*

**B1.2. Business Leader Perspective:**

* "The platform must attract and retain users. Search and recommendations are key – they have to be better than competitors."
* ”We need to launch new features quickly to stay competitive. The architecture shouldn't slow us down."

**B1.3. Developer Perspective:**

* “It should be easy to add support for new listing categories or change recommendation algorithms without breaking everything."

**B1.4. Utility Tree**

|  |  |  |
| --- | --- | --- |
| **Classification** | **QS\_ID** | **Quality Scenario** |
| Performance | QS-01 | Search operation response time |
| Performance | QS-02 | Listing visibility time (new/updated) |
| Performance | QS-03 | Home Screen Loading time |
| Performance | QS-04 | Chat message delivery time |
| Performance | QS-05 | Listing Media Processing Time |
| Performance | QS-06 | Service auto-scaling based on load |
| Reliability | QS-07 | Monitoring of Cloud Micro services |
| Reliability | QS-08 | Chat messaging ordering & durability |
| Reliability | QS-09 | System recovery from errors/infrastructure failure |
| Maintainability | QS-10 | Flexibility to adopt new Recommendation algorithms |
| Maintainability | QS-11 | Flexibility to add new Category in item listing section |
| Maintainability | QS-12 | System operation cost-efficiency |
| Security | QS-13 | Block Sharing of Personal Information |
| Security | QS-14 | Listing Moderation & Malicious Content Prevention |
| Security | QS-15 | Prevent unauthorized access to MicroService |
| Security | QS-16 | Real-time flagging of high-risk sellers |
| Usability | QS-17 | Recommendation usefulness to end users |
| Usability | QS-18 | Search Result relevance |

**B2. Quality Scenarios Details**

|  |  |
| --- | --- |
| **QS\_ID** | **Description** |
| QS-01 | Users should receive search results very quickly after submitting their query to ensure a responsive experience. |
| QS-02 | When a seller posts or updates a listing, it should become discoverable in search and recommendations almost immediately. |
| QS-03 | The application's home screen, including all personalized and general content, needs to load rapidly upon opening. |
| QS-04 | Messages sent between users in the chat feature should arrive with minimal delay to feel like a real-time conversation. |
| QS-05 | Media files (like images) uploaded during listing creation should be processed and ready for preview/use swiftly. |
| QS-06 | The system must automatically adjust resources to effectively handle large increases in user traffic without performance degradation. |
| QS-07 | Key system components should operate reliably with very few operational errors detected through monitoring. |
| QS-08 | The chat system must ensure that messages are never lost and are consistently delivered in the correct order. |
| QS-09 | In case of a component or infrastructure failure, the system needs to recover full functionality automatically and quickly. |
| QS-10 | Integrating new or improved recommendation algorithms into the system should require minimal development effort. |
| QS-11 | Adding support for new product categories in the listing process should be straightforward and require minimal development time. |
| QS-12 | Infrastructure costs should be managed effectively, aiming for a low operational cost relative to the number of user requests handled, especially through efficient auto-scaling. |
| QS-13 | The chat system must accurately detect and block messages containing sensitive personal information or attempts to transact off-platform, while rarely blocking legitimate messages. |
| QS-14 | The automated moderation for new listings should effectively prevent prohibited content while ensuring that very few legitimate listings are incorrectly blocked. |
| QS-15 | The system's internal services must be secure, blocking any unauthorized access. |
| QS-16 | Potentially fraudulent seller actions should be identified and flagged for review very quickly after they occur. |
| QS-17 | Recommended items presented to users should be relevant and appealing, leading to a high level of user engagement (e.g., clicks). |
| QS-18 | Search results displayed should closely match the user's intent, resulting in a high likelihood that users click on the results presented. |

1. Quality Scenario Analysis

// A5. Quality Requirement Specification

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

**C1. Quality Scenario Analysis Metric**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 | M | L | Ignore |
| **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 | H | M | QA\_06 |
| **Availability** | QS-09: System should be able to recover from any error in system or Server Infrastructure | M | L | Ignore |
| **Maintainability** | **Modifiability** | QS-10: System should be flexible to adopt new Recommendation algorithms | M | H | QA\_03 |
| **Modifiability** | QS-11: System should be flexible to add new Category in item listing section | L | L | Ignore |
| **Cost Efficiency** | QS-12: System should be Cost efficient | M | L | Ignore |
| **Security** | **Fraud Detection** | QS-13: System must block the sharing of contact information and any off the platform transaction | M | M | QA\_11 |
| **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 unauthorized access to Micro Service | M | L | Ignore |
| **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 |

**C2. Basis for QA & NFR Selection**

**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. However, since a few seconds of lag in Listing Visibility is acceptable, hence we can consider this a **medium-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. But since chat is not the primary feature of the app, this QA is classified as important but **low-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 to low-priority quality attribute**.

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

While ensuring the system can handle varying loads is crucial for performance and cost-efficiency (**NFR\_03**), the decision to deploy on a managed Kubernetes platform like **AWS EKS** provides powerful built-in auto-scaling capabilities. Features like the Horizontal Pod Autoscaler allow the system to automatically adjust resources based on observed metrics (CPU, memory, custom metrics), addressing the core requirement for dynamic scaling. Therefore, QS-06 is **dropped** as a primary architectural driver requiring distinct application-level design patterns, as the chosen platform inherently supports this capability.

**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\_10)**.

**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 **medium-priority reliability requirement**.

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

While high availability is essential, deploying on a managed platform like **AWS EKS** inherently provides robust fault tolerance, automatically handling container restarts and node recovery. Since the platform addresses much of the basic infrastructure recovery, this reduces the need for unique architectural focus on this specific scenario. Therefore, QS-09 is **dropped** as a primary architectural driver requiring distinct design solutions.

**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 a **high-priority attribute to ensure modifiability**.

**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**

While chat safety contributes to **long-term community trust**, the initial architectural design must prioritize core discovery and listing functionalities, which are fundamental to the platform's viability. Baseline mitigation for chat risks can be addressed through platform usage policies and user reporting mechanisms. Implementing advanced, real-time blocking introduces significant technical complexity and potential negative impacts on chat performance (QS\_04). Given the higher priority of core features and the complexity involved, this Quality Scenario is deemed a **lower priority quality attribute** for driving the initial architecture.

**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, high-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. **Hence, it is selected as Highest Priority QA.**

**Note: To Limit the scope of this report, some QAs have been dropped for analysis.**

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?

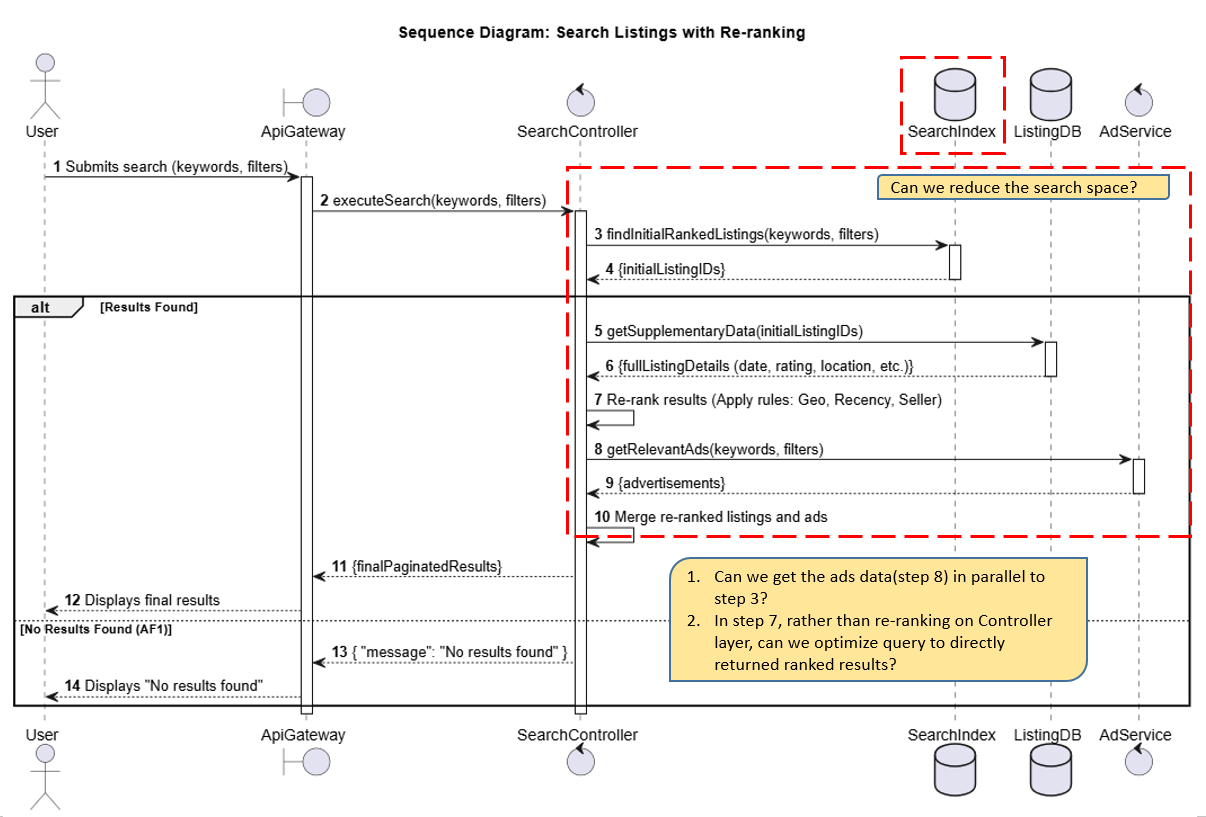
// C6-3. Are modifiability analysis and solution candidate appropriate?

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

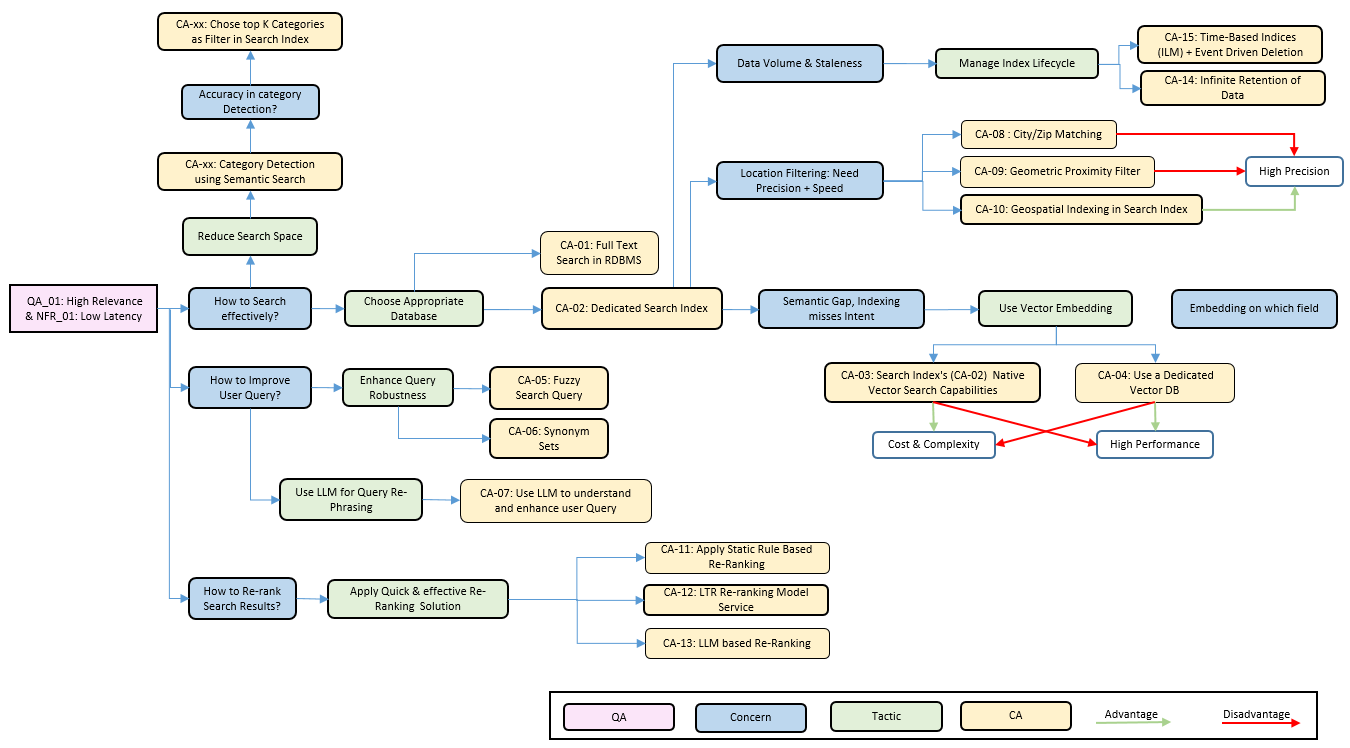
## D1. NFR\_01 [Performance] Search operation response time & QA\_01 [Usability] Search Result should be relevant.

### D1.1 Design Issues

**Add concerns**



**D1.2. Candidate Structure**



### CA-01: Full Text Search in Relational Database

We can use enhance the search by using the **built-in Full-Text Search (FTS)** capabilities of the relational database (e.g., PostgreSQL's tsvector).

### CA-02: Dedicated Search Index Service (e.g. Elasticsearch/Solr/OpenSearch)

We can use a Dedicated Search Index Service for storing listings that are presently live for Selling.

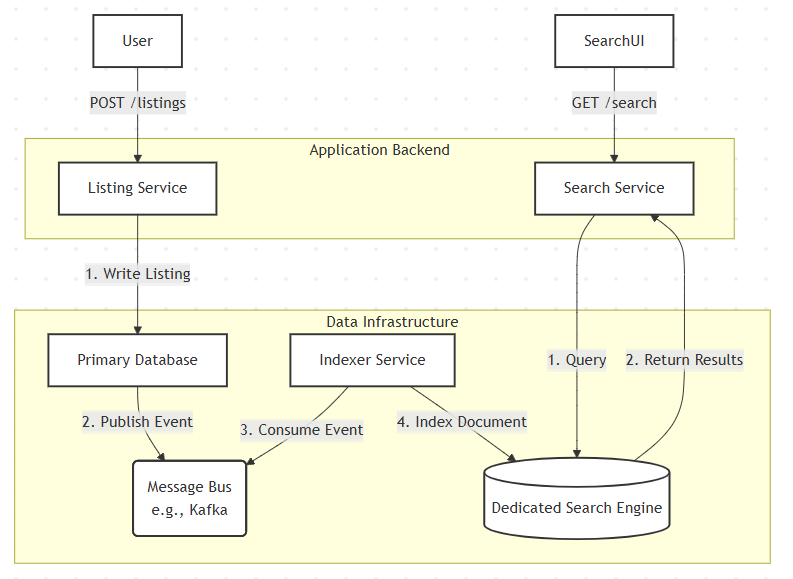


Figure: Search Index Service Data Flow

### CA-03: Use Search Index’s Vector Search Capabilities

We can use the in-build Vector Storage and Search capabilities of Dedicated Search Index.

### CA-04: Dedicated Vector DB (like Pinecone/Weaviate/Faiss)

### CA-05: Fuzzy Search Query

We can adopt fuzzy search logic to propose auto correction in search input text (like “moble” 🡪 “mobile”).

### CA-06: Use Synonym Set

We can use Synonym Set to handle situation where user can use different terms to refer to same item (like “TV” vs “television” or “cupboard” vs “cabinet” vs “wardrobe” vs “almirah”)

### CA-07: Use LLM to understand and Enhance User Query

Leverage LLM Capabilities to understand user Intent and enhance user Query before searching in Search Index.

### CA-08: City/Zip Matching Only

### CA-09: Geometric Proximity Filter

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.

### CA-10: 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.

### ~~CA-109: 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.~~

### CA-11: 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.

### CA-12: LTR Re-ranking Model Service

### CA-13: LLM Based Re-Ranking

### CA-14: Infinite Retention of Search Data

### CA-15: Time-Based Indices (ILM) + Event Driven Deletion

* 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.

## D2. NFR\_02 [Performance]: Home Screen Loading time should be low

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

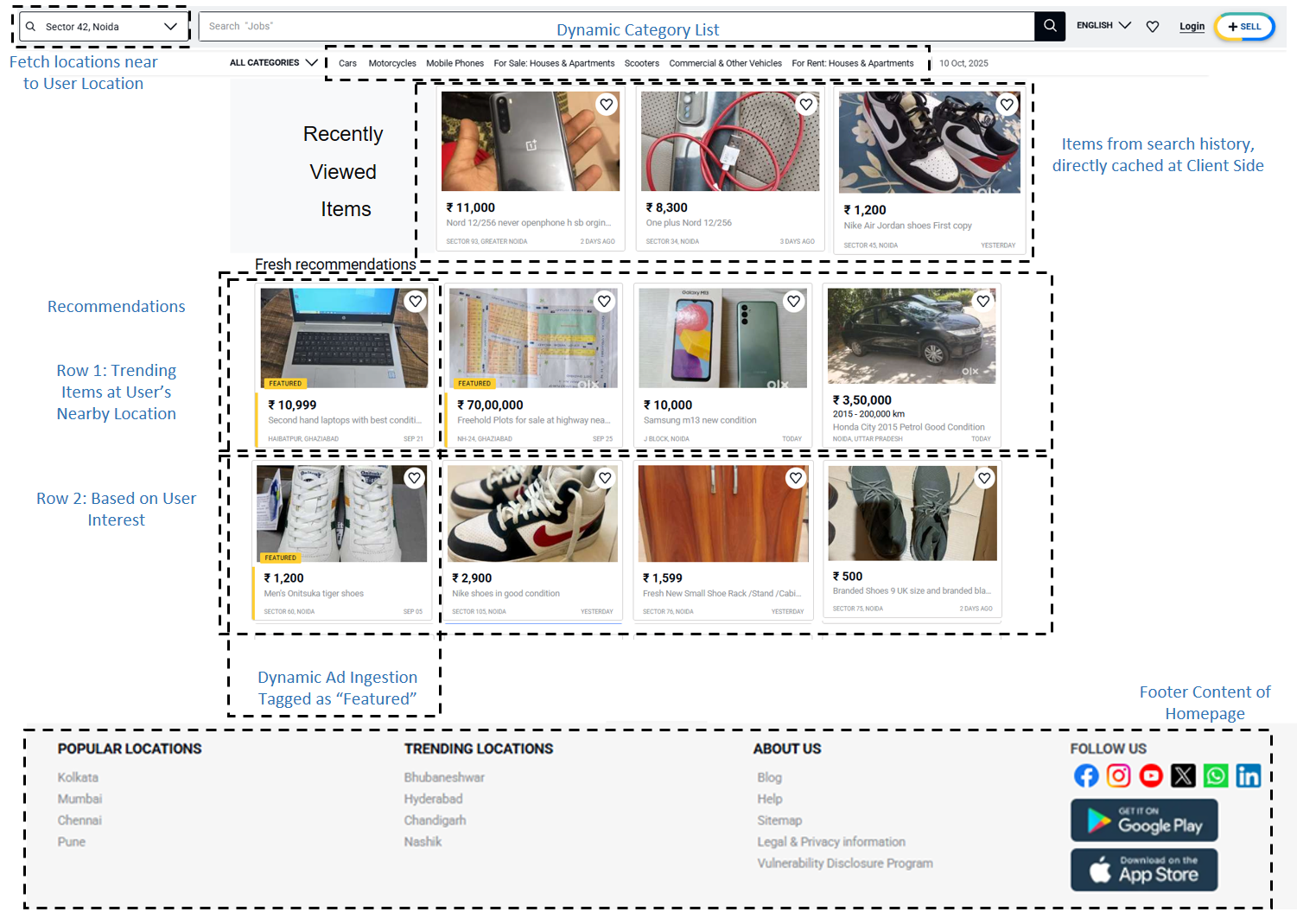
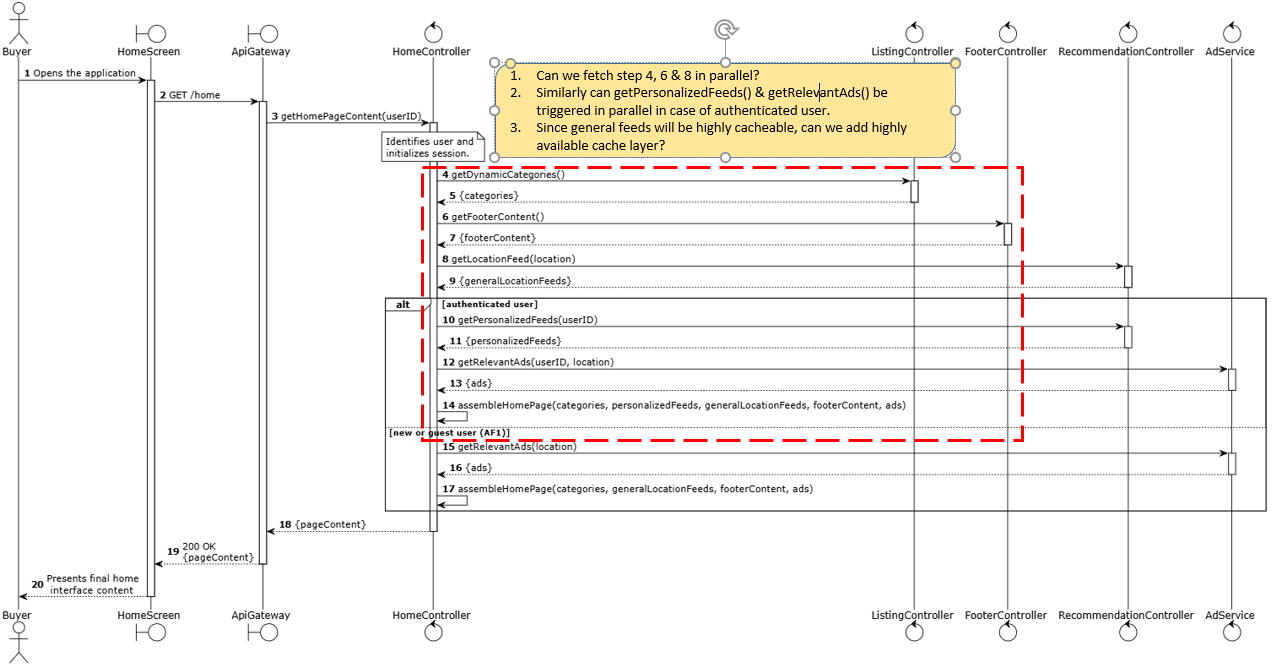
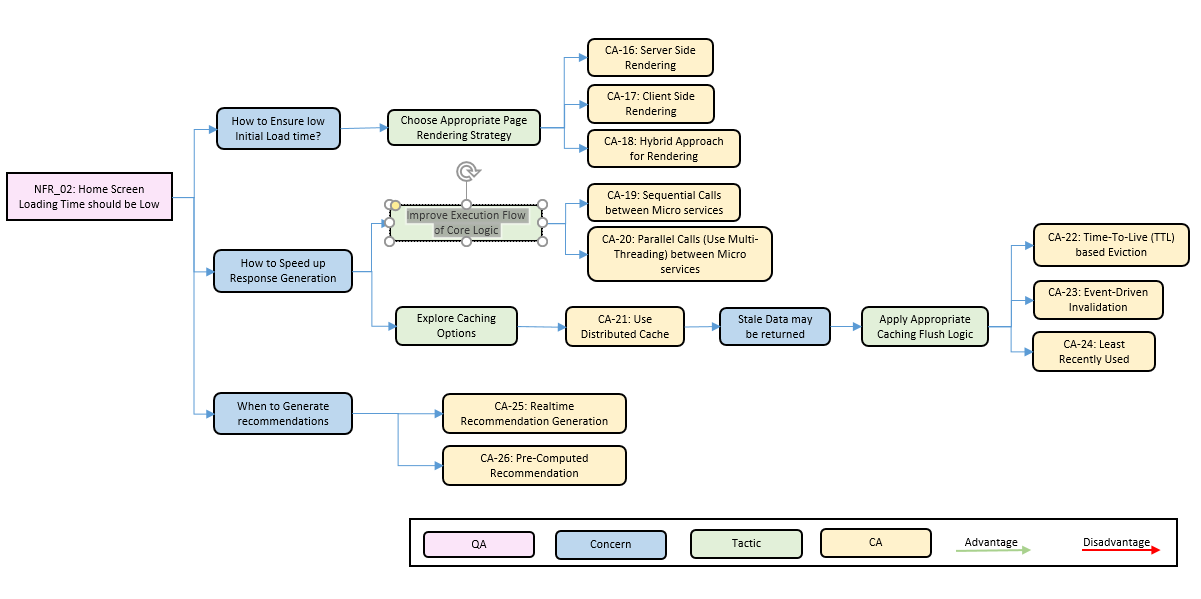


Figure: Home Screen Core Features

Major Problems:

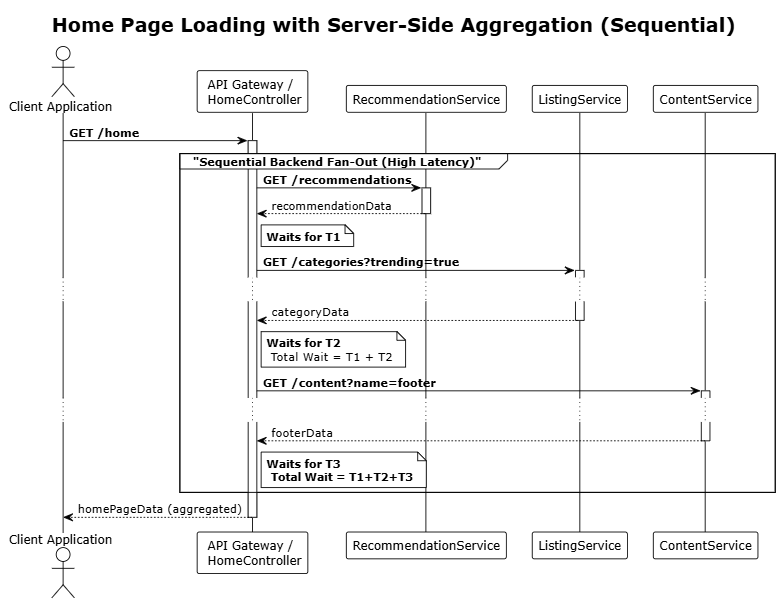
* **High Read Load & Concurency:** The homepage will receive a request from nearly every active user, resulting in a massive, concurrent read load. The system serving this page must be able to handle this load without creating a bottleneck.
* **Data Aggregation Latency:** A homepage is not a single piece of data. The server must assemble it by fetching multiple, independent components (e.g., "Trending Items," "Newest Listings," "Based on Your History," "Categories"). Waiting to fetch all these components in real-time will make the page slow.
* **Client-Side Rendering Latency:** If the server just provides data and the user's browser (client-side) has to make multiple follow-up API calls to get the recommendations, the user will see a slow-loading page with many "pop-in" elements, which feels broken and slow.





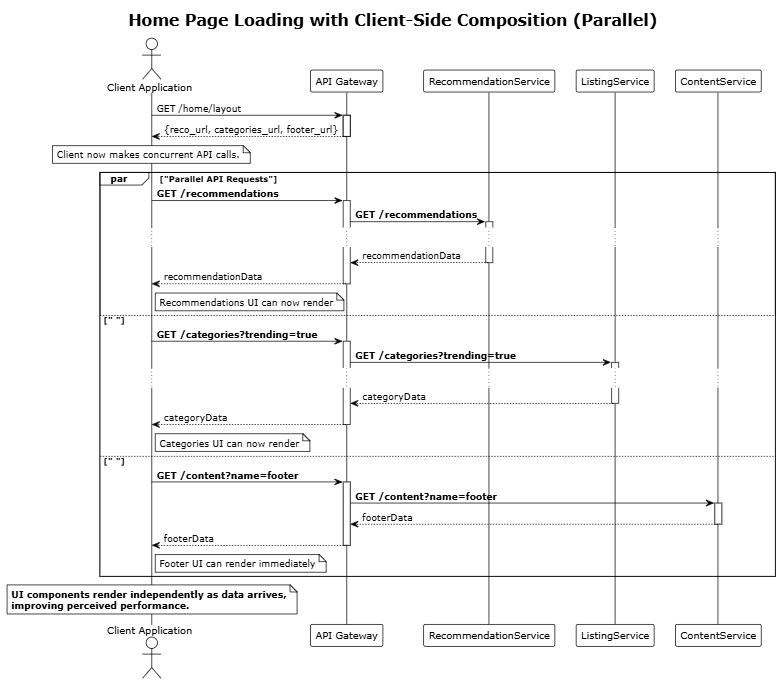
### CA-16: Server-Side Aggregation Pattern

* 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.



### CA-17: 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.



### CA-18: Hybrid Approach for Rendering

### CA-19: Sequential Calls between Micro Services

### CA-20: Parallel Calls (using Multi-Threading) between Micro Services

### CA-21: Use Distributed Cache

* 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).

### CA-22: Time-to-Live (TTL) based Cache Eviction

### CA-23: Event driven Invalidation of Cache

### CA-24: Least Recently Used (LRU) based Eviction

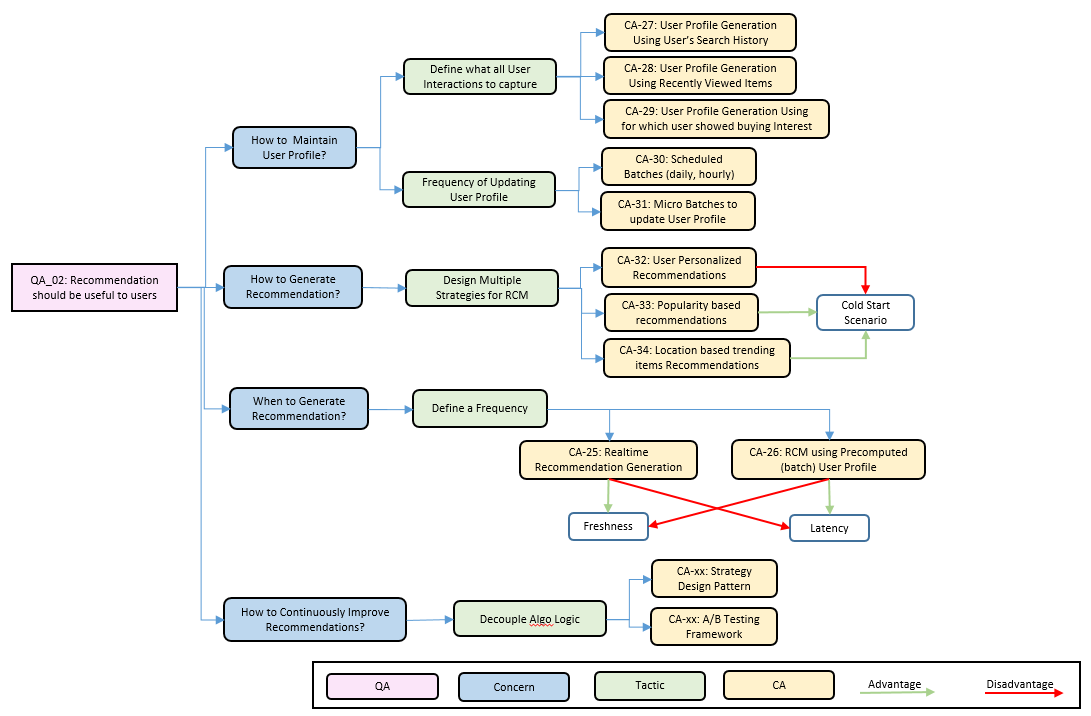
### CA-25: Real time Recommendation Generation

### CA-26: Pre-Computed Recommendation

## D3. QA\_02 [Usability]: Recommendation should be useful to users.

**Major Problems:**

* **User Profile & Activity Tracking:** To provide *personalized* recommendations, the system must collect, store, and process a user's behavior (e.g., listings viewed, items wishlisted, searches performed, categories explored). A robust and scalable structure is needed to capture this activity stream and build an actionable user profile.
* **Recommendation Generation Strategy:** What is the core architectural approach for generating the list of recommended items? This involves deciding on the *type* of recommendation (e.g., collaborative filtering, content-based, trending items) and, most importantly, *when* this list is generated (e.g., computed in real-time vs. pre-computed in a batch job).
* **The "Cold Start" Problem:** How does the system provide a relevant and engaging homepage for new or anonymous users who have no behavioral profile? An effective fallback strategy (e.g., "Popular on site," "Trending in your area") is essential for this large user segment.



### CA-25: Real time Recommendation Generation

### CA-26: Pre-Computed Recommendation

### CA-27: User Profile Generation using User’s Search History

### CA-28: User Profile Generation using Recently Viewed Items

### CA-29: User Profile Generation using Items for which user showed buying Interest

### CA-30: Scheduled Batches for Updating User Profile

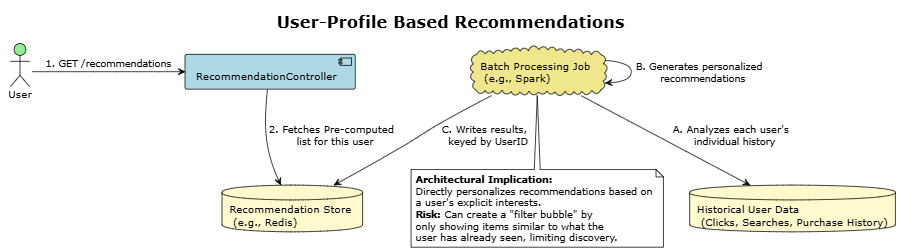
### CA-31: Micro Batches to update User Profile

### CA-32: User Personalized Recommendations

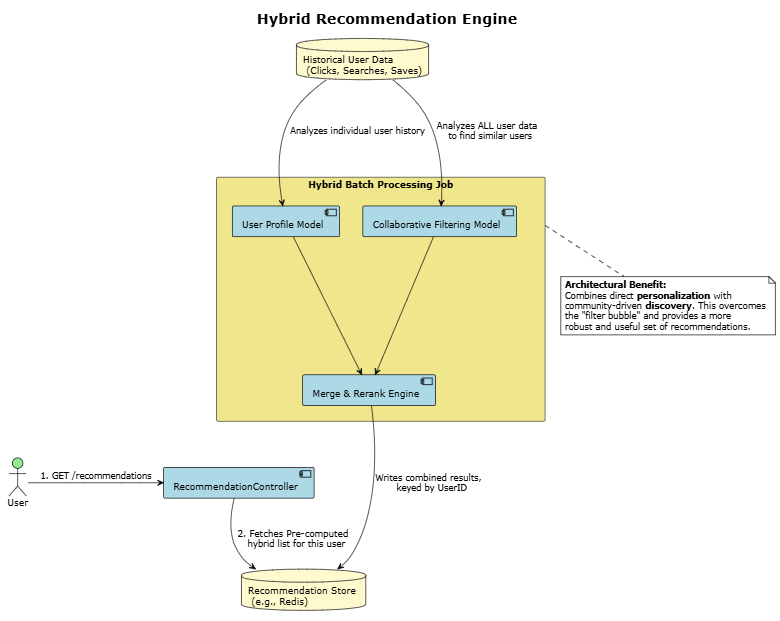
### CA-33: Popularity based Recommendations

### CA-34: Location based Trending Items Recommendations

* 1. CA-116: 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-117: 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-118: 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**

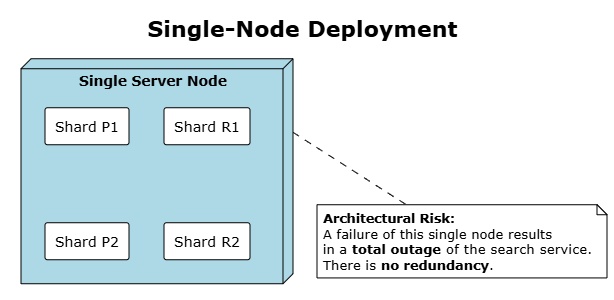
* 1. CA-119: Using CDN for reducing geo network latency
* We will utilize CDN server for keeping **website static content**
* CDN server replicates static content server to all **Edge location globally**
* User will be served with static content from nearest edge location, and will help in reducing geo network latency

**~~Targeted QA/NFR:~~**

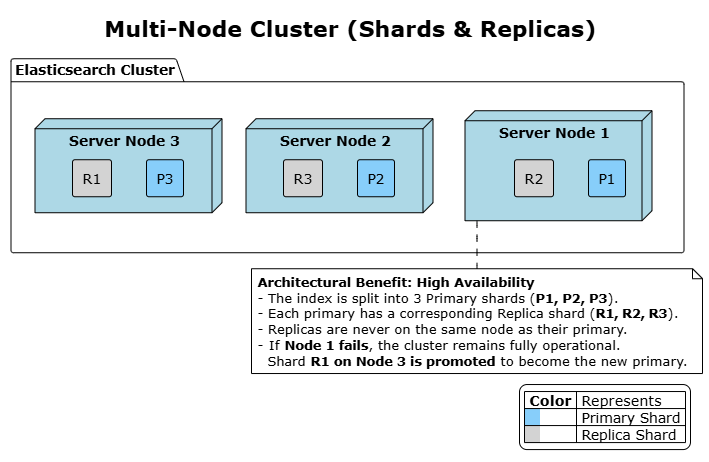
* ~~NFR\_03 (Performance): Service should be able to scale if load increase by N%~~
* ~~NFR\_04 (Reliability): System should be able to recover within ‘N’ minutes~~

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

* 1. ~~CA-120: 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-121: 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~~**

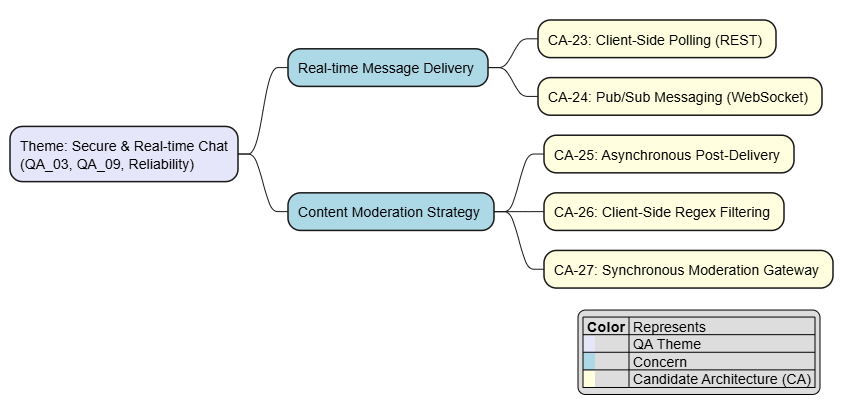
~~11.1 CA-122: Service Resilience via Circuit Breaker~~

* ~~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:**

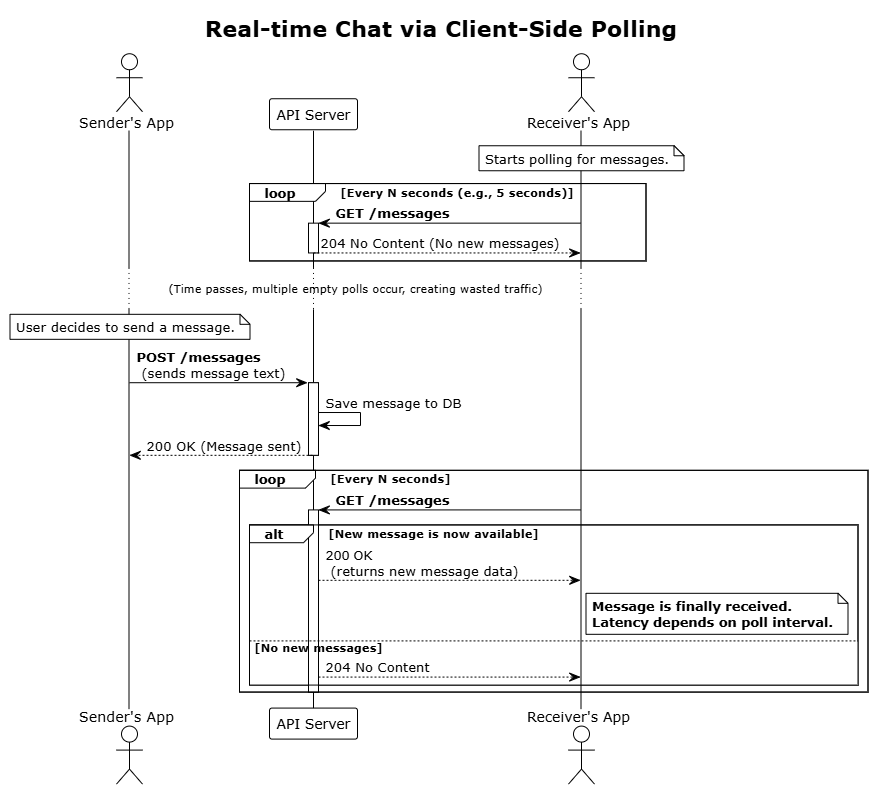
* ~~QA\_03 (Security): System must block the sharing of contact information and any off the platform transaction~~
* QA\_09 (Performance): Users should be able to get chat message quickly

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.

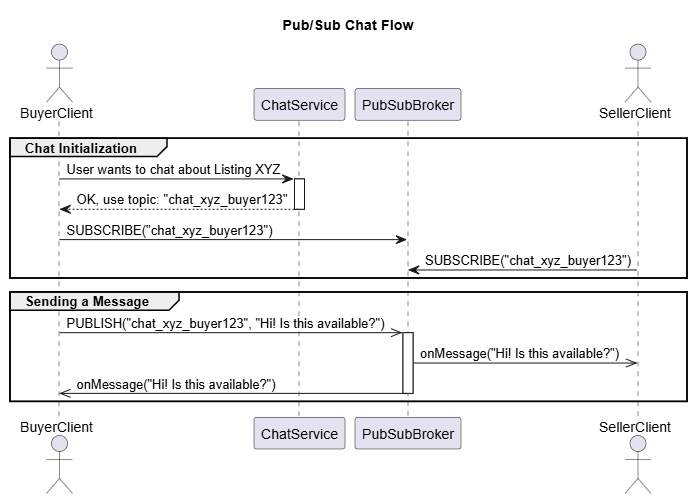


**12. Concern: Real-time Message Delivery Protocol**

* 1. CA-123: 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**.

~~~~

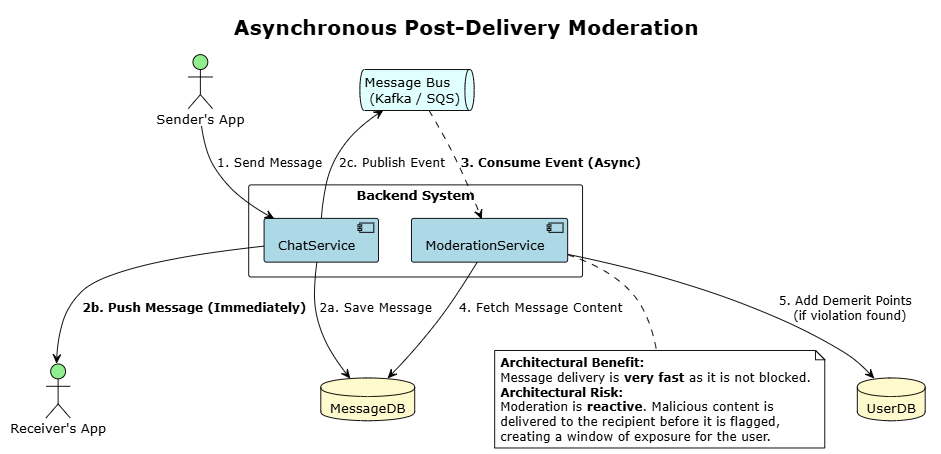
* 1. CA-124: Pub/Sub Messaging for Chat
* We can implement a **Pub/Sub (Publish/Subscribe) pattern** where each chat conversation is a unique 'topic'. When users join a chat, their client applications **subscribe** to that specific topic. To send a message, the application simply **publishes** it to 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.**



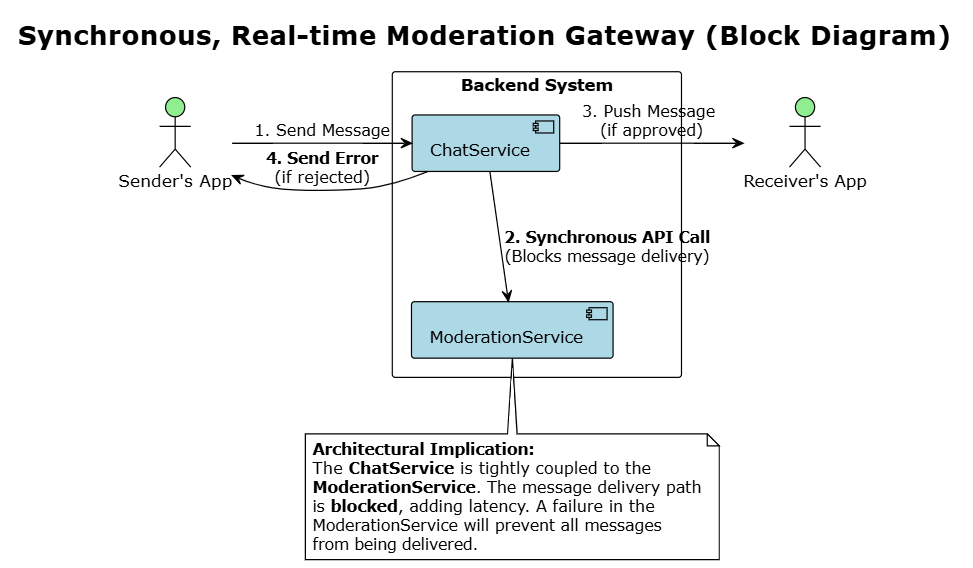
**13. Concern: Content Moderation in Chat**

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

* 1. CA-125: 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-126: 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-127: 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:**

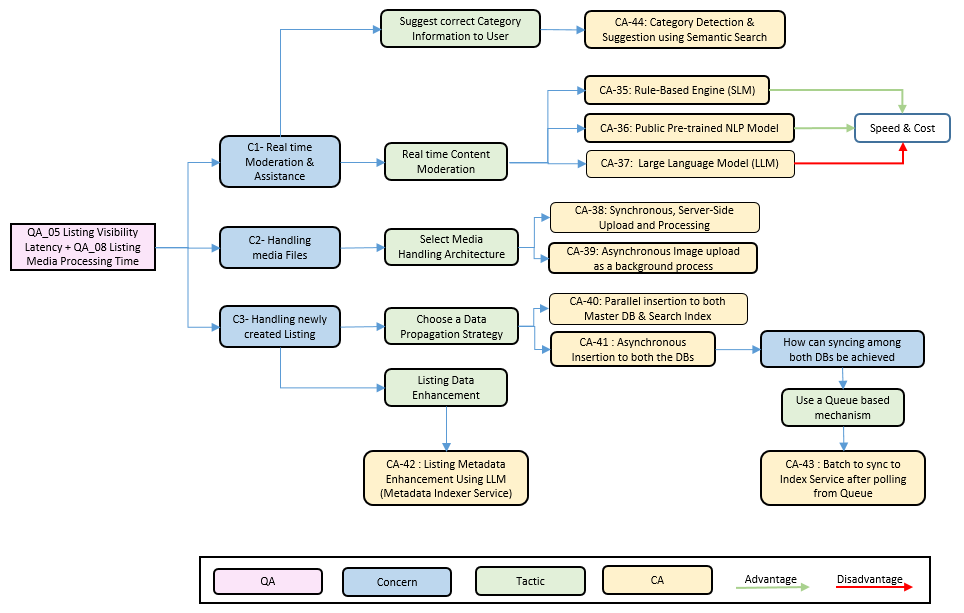
* QA\_04 (High Priority): Listing Moderation & Malicious Content Prevention. Ensures all new and updated listings are automatically scanned for prohibited content before becoming visible.
* QA\_05 (High Priority): Listing Visibility Latency. This defines the maximum acceptable delay from when a seller 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\_04), 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).

**Design Issues for Listing Creation & Management**

## D4. QA\_05 [Usability]: Listing Visibility Latency should be low. & QA\_05 [Usability]: Listing Media Processing Time should be low.

* **Optimizing Metadata Enrichment vs. Latency & Cost:** How to determine the right level and timing of metadata enrichment (e.g., basic LLM tags vs. conditional deep image analysis) to maximize discoverability without unacceptably impacting listing visibility latency (**QA\_05**) or incurring excessive operational costs?
* **Managing Asynchronous Complexity & Consistency:** How to reliably orchestrate the multiple background steps (media processing, enrichment, indexing) after user submission, ensuring data consistency across systems and preventing listings from getting stuck or lost?
* **Integrating Content Moderation Effectively:** How to include necessary content moderation (**QA\_04)**?



### CA-35: (On Client) Rule Based Engine for Content Moderation

* 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.

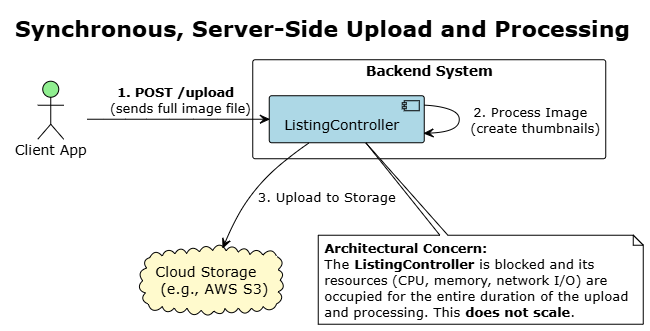
### CA-36: Public Pre-Trained NLP Model for Content Moderation

* 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**.

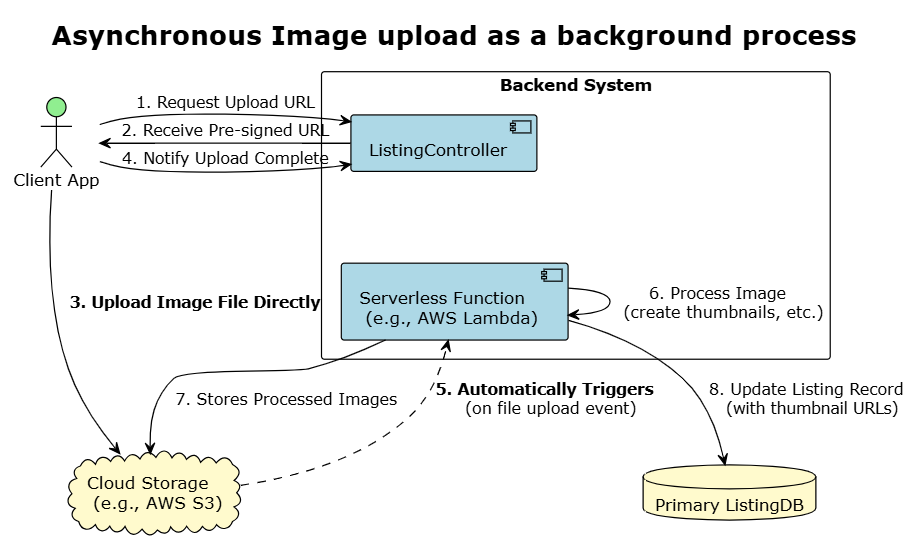
### CA-37: LLM for Content Moderation

* 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.

### CA-38: Synchronous, Server-Side Upload and Processing

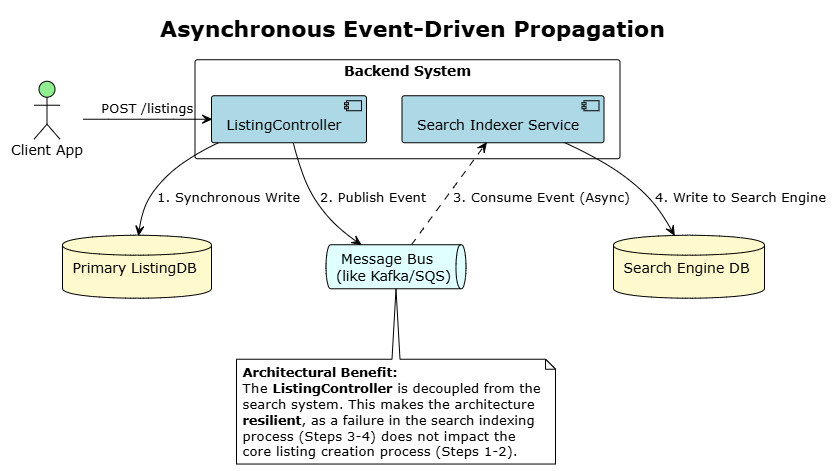


### CA-39: Asynchronous Image upload as a background process



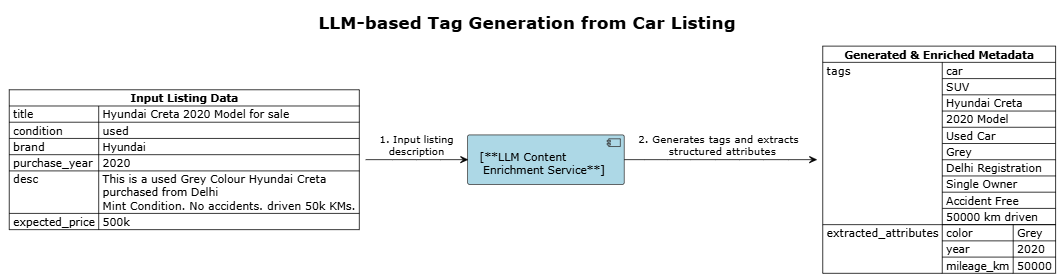
### CA-40: Parallel insertion to both Master DB & Search Index

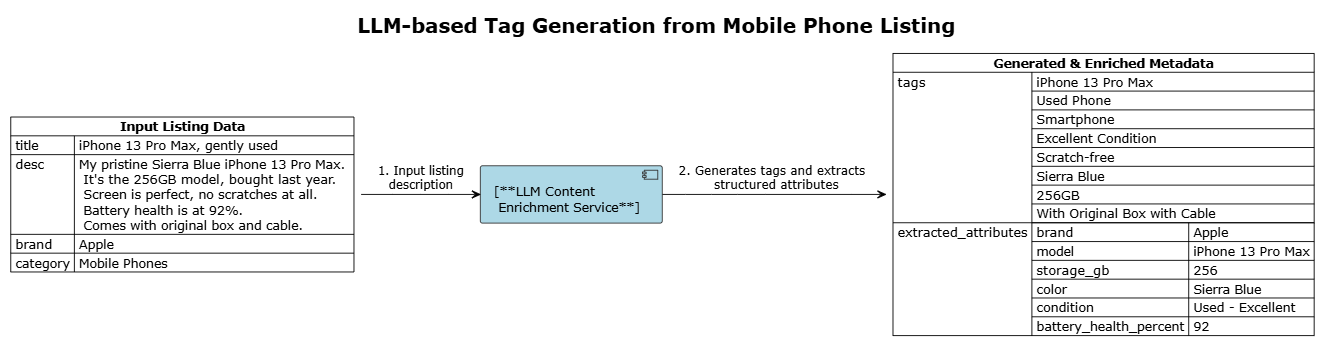
### CA-41: Asynchronous Insertion to both the DBs



### CA-42: Listing Metadata 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.





### CA-43: Batch to sync to Index Service after polling from Queue

### CA-44: Category Detection & Suggestion using Semantic

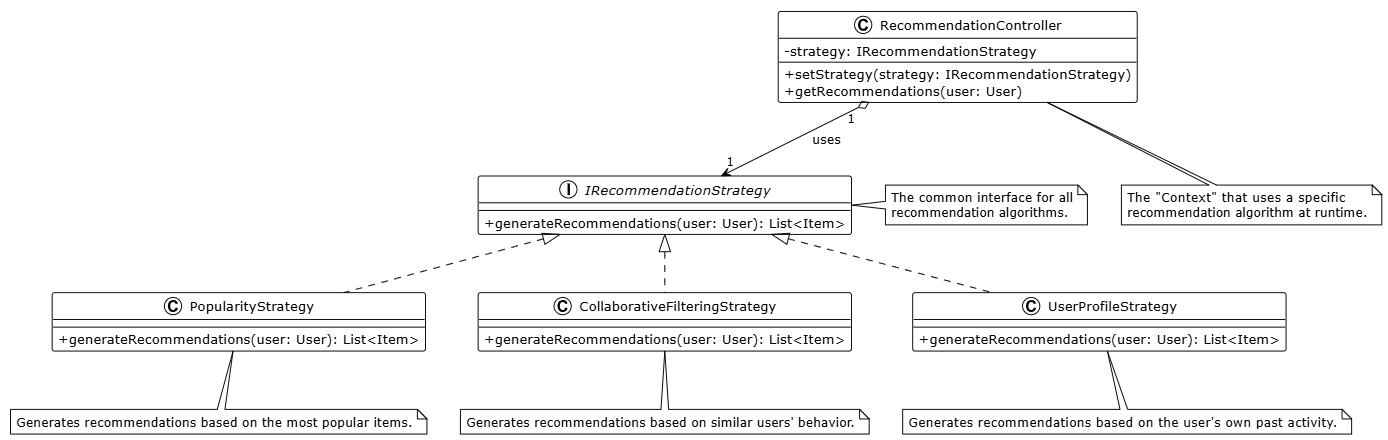
## D5. QA\_03 [Maintainability]: System should be flexible to adopt new Recommendation algorithms.

**CA-45: 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.

**CA-46: 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.



**CA-47: 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\_10 (Reliability): Monitoring and Alerting of Cloud MicroServices.

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

* 1. CA-142: 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-143: 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.

### CA-144: 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 | Full Text Search in Relational Database | Rejected |
| CA-02 | Dedicated Search Index Service (e.g. Elasticsearch/Solr/OpenSearch) | Selected |
| CA-03 | Use Search Index’s Vector Search Capabilities | Selected |
| CA-04 | Dedicated Vector DB (like Pinecone/Weaviate/Faiss) | Rejected |
| CA-05 | Fuzzy Search Query | Selected |
| CA-06 | Use Synonym Set | Selected |
| CA-07 | Use LLM to understand and Enhance User Query |  |
| CA-08 | City/Zip Matching Only | Rejected |
| CA-09 | Geometric Proximity Filter | Rejected |
| CA-10 | Native Geospatial Indexing | Selected |
| CA-11 | Static Business-Rule Re-ranking |  |
| CA-12 | LTR Re-ranking Model Service |  |
| CA-13 | LLM Based Re-Ranking | Rejected |
| CA-14 | Infinite Retention of Search Data | Rejected |
| CA-15 | Time-Based Indices (ILM) + Event Driven Deletion | Selected |
| CA-16 | Server-Side Aggregation Pattern |  |
| CA-17 | Asynchronous Client-Side Composition | Rejected |
| CA-18 | Hybrid Approach for Rendering |  |
| CA-19 | Sequential Calls between Micro Services | Rejected |
| CA-20 | Parallel Calls (using Multi-Threading) between Micro Services | Selected |
| CA-21 | Use Distributed Cache | Selected |
| CA-22 | Time-to-Live (TTL) based Cache Eviction | Selected |
| CA-23 | Event driven Invalidation of Cache | Selected |
| CA-24 | Least Recently Used (LRU) based Eviction | Rejected |
| CA-25 | Real time Recommendation Generation | Rejected |
| CA-26 | Pre-Computed Recommendation | Selected |
| CA-27 | User Profile Generation using User’s Search History | Selected |
| CA-28 | User Profile Generation using Recently Viewed Items | Selected |
| CA-29 | User Profile Generation using Items for which user showed buying Interest | Selected |
| CA-30 | Scheduled Batches for Updating User Profile | Rejected |
| CA-31 | Micro Batches to update User Profile | Selected |
| CA-32 | User Personalized Recommendations | Selected |
| CA-33 | Popularity based Recommendations | Selected |
| CA-34 | Location based Trending Items Recommendations | Selected |
| CA-35 | (On Client) Rule Based Engine for Content Moderation | Selected |
| CA-36 | Public Pre-Trained NLP Model for Content Moderation | Selected |
| CA-37 | LLM for Content Moderation | Rejected |
| CA-38 | Synchronous, Server-Side Upload and Processing | Rejected |
| CA-39 | Asynchronous Image upload as a background process | Selected |
| CA-40 | Parallel insertion to both Master DB & Search Index | Rejected |
| CA-41 | Asynchronous Insertion to both the DBs | Selected |
| CA-42 | Listing Metadata Enhancement Using LLM | Selected |
| CA-43 | Batch to sync to Index Service after polling from Queue | Selected |
| CA-44 | Category Detection & Suggestion using Semantic |  |
| CA-45 | Hardcoded Logic in Controller | Rejected |
| CA-46 | Strategy Design Pattern | Selected |
| CA-47 | A/B Testing | Selected |

1. Final Architecture

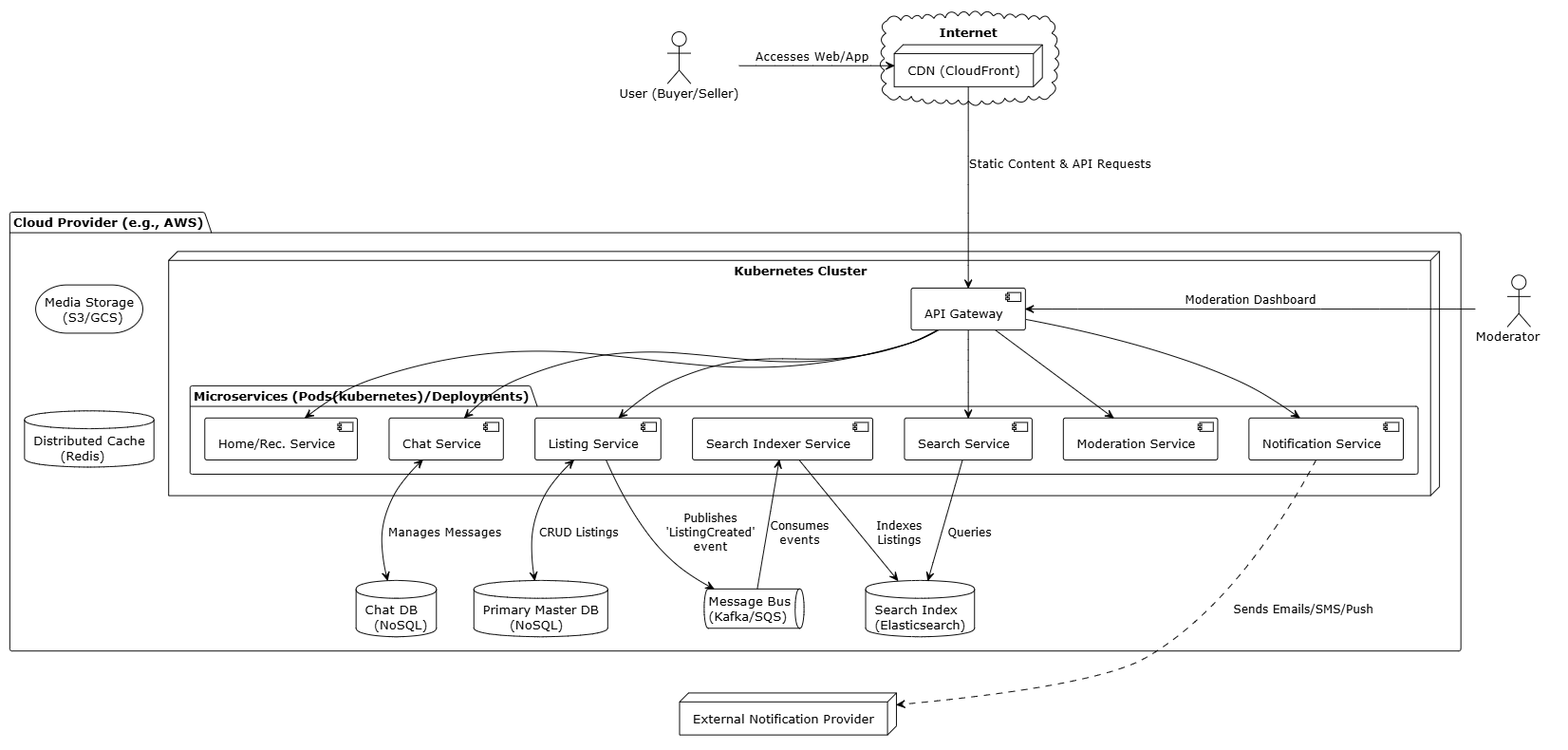
// A7. Architecture Design

// 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**

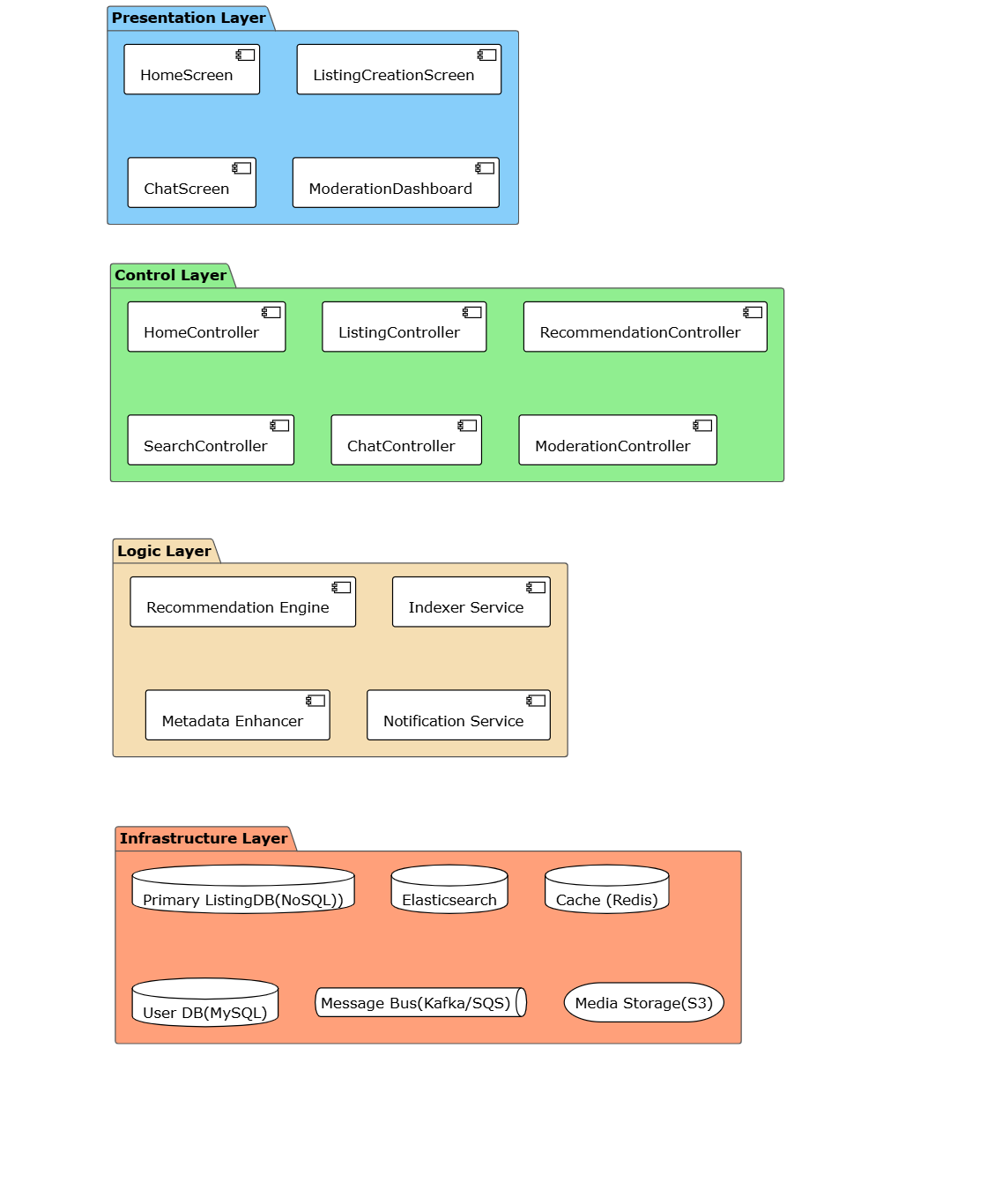
**Overview.** The marketplace is deployed as a cloud-native, Kubernetes-orchestrated system with an edge tier (DNS/CDN/WAF/API-Gateway), a services tier (microservices in K8s), and managed data plane (RDS/PSQL, Redis, Kafka/SNS+SQS, S3/GCS, OpenSearch/ES, Warehouse). Services scale horizontally and communicate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