Tunisian Republic

Ministry of Higher Education and Scientific Research

National Institute of Applied Science and Technology

INSAT

PERSONAL AND PROFESSIONAL PROJECT REPORT

Submitted as part of the academic requirements

for the Software Engineering Program

By

Mohammmed Achref Hemissi

Mohamed Dhia Medini

Leith Engazzou

Hiba Chabbouh

Hanine Khemir

Younes Abbes

FoundIT:

Computer Vision-Powered Lost and Found Mobile Application

Academic Supervisor :	**Mrs Hajer Taktak**
Academic Year :	**2024–2025**

Acknowledgments

We begin by acknowledging the divine guidance of God Almighty and Merciful. Our profound gratitude is extended to all who contributed to the successful completion of this endeavor and the production of this report.

We wish to express our deepest appreciation to the FoundIt Team. Their unwavering dedication, persistent effort, and remarkable resilience in the face of numerous challenges and competing commitments were fundamental to our collective success. The teamwork, diligence, and commitment to excellence demonstrated by each member were truly invaluable.

Our sincere thanks are also extended to the distinguished members of the jury. Their expert evaluation and generous commitment of time were crucial for a thorough and impartial assessment of our work. We particularly thank Mrs. **Hajer TAKTAK** for her insightful feedback and constructive guidance, which significantly enhanced the project's quality.

Finally, we are immensely grateful to the faculty and staff of INSAT. Their support, encouragement, and mentorship have profoundly shaped our academic journey. Their dedication to educational excellence and their role in nurturing future leaders have left an indelible mark on our development.

List of Acronyms

INSAT	National Institute of Applied Science and Technology
AI	Artificial Intelligence
UI	User Interface
UML	Unified Modeling Language
SDK	Software Development Kit
BLOB	Binary Large Object
HNSW	Hierarchical Navigable Small World

Contents

List of Figures

List of Tables

General Introduction

The increasing frequency of lost personal belongings in public spaces—such as transportation hubs, educational institutions, and commercial areas—has underscored the urgent need for an efficient and modernized solution to reunite owners with their items. Traditional lost and found systems, typically reliant on physical locations, manual documentation, and human oversight, often prove to be slow, unreliable, and difficult to scale. In contrast, the widespread availability of mobile technology and the growing capabilities of artificial intelligence present an opportunity to revolutionize this process. In this context, the **Lost & Found Mobile Application**, developed by a team of software engineering students, offers a digital platform where users can report lost or found items, search through categorized listings, and interact directly with other users to facilitate item recovery. Key features of the application include AI-powered smart matching and image recognition for improved search accuracy, an interactive map for geolocating items, and an integrated chat system to streamline communication. To ensure a well-structured and methodical development process, the project follows the Classical (Waterfall) software development methodology, allowing clear phase separation and thorough documentation.

This report presents the complete lifecycle of the project, from conceptualization to implementation, and is organized into six main chapters: *Project Context*, which describes the problem and motivation behind the solution; *Requirements Specification*, which outlines both functional and non-functional system requirements; *System Design and Architecture*, which details the application's structure and components; *Artificial Intelligence Integration*, which highlights the role of AI in improving matching accuracy; *Implementation and Testing*, which explains the tools, technologies, and testing strategies used; and *Conclusion and Perspectives*, which summarizes the results and proposes potential future enhancements such as real-time notifications, institutional partnerships, and multilingual support. Overall, this mobile application aims to deliver an intelligent, user-friendly, and scalable solution to the persistent issue of lost items in everyday life.

Chapter 1

Project Context

1.1 Introduction

This chapter presents the context of the project, outlining the problem it addresses, evaluating existing alternatives, and introducing the core functionalities and development approach adopted in the proposed solution.

1.2 Problem Statement

Losing personal belongings in public spaces such as campuses, malls, or transport systems is a frequent and frustrating experience. Despite the growing use of online platforms, the recovery process remains disorganized and inefficient due to:

- **Lack of centralization**: Users often rely on fragmented social media groups or websites.
- **Limited visibility**: Posts about lost or found items can easily go unnoticed.
- **No matching system**: Manual browsing is tedious and error-prone.
- **Absence of direct communication**: Platforms may not support messaging between users.
- **Moderation issues**: Posts are often unchecked, leading to spam or irrelevant content.

1.3 Existing Solutions and Their Limitations

Several platforms currently try to address this issue. Here is a comparison of some popular solutions:

Table 1.1: Comparison of Lost & Found Solutions

Platform	**Focus**	**Strengths**	**Weaknesses**	**AI Features**	**Matching**
Facebook	Community L&F	Wide reach, local groups	Fragmented	None	Manual
Reddit	Forum L&F	Large, global user base	Disorganized	None	Manual
Craigslist	Classifieds	Free, broad categories	Scams	None	Manual
Tile/AirTag	Item Tracking	Real-time tracking	Tracker-only	None	–
L&F Websites	Dedicated L&F	Searchable database	Regional	Basic	Keyword
Campus Portals	Campus-specific	Institution-focused	Campus-only	None	Manual

As seen above, no existing platform offers a complete, intelligent, and user-friendly solution. Most lack advanced features such as smart item matching, real-time notifications, and AI-assisted post generation.

1.4 Proposed Solution: Lost & Found Mobile App

To address the above limitations, we propose a centralized, AI-powered mobile application dedicated to lost and found item management.

Key Features

- **Item Management:**
  - **Report an Item:** Users can create a post with a description, photo, and location. An AI assistant can help auto-generate the post.
  - **Post Classification:** Posts are separated into "Lost" and "Found" categories for easier matching.
  - **Search and Filters:** Users can search using filters like category, color, location, and date. Matching results are ranked by relevance.
  - **Notifications:** Users are notified when a similar item is reported. This feature is powered by the AI-based Similarity Search Service.
- **User Management:**
  - Registration/Login with email or OAuth (Google, Facebook).
  - User dashboard to view history of reported items.

- **Interactive Map:**
  - Map showing the reported locations of lost/found items.
- **Integrated Chat:**
  - Enables direct communication between users.
- **AI Features:**
  - Smart matching based on description similarity.
  - Image recognition for visually similar items.
  - Automatic post suggestions.
- **Administration Tools:**
  - User moderation and banning of abusive users, powered by the suspicious user detection service.
  - Admin dashboard for monitoring platform usage.

1.5 Development Methodology: The Waterfall Model

The development of the **Lost & Found Mobile App** is based on the **Waterfall Model**, a classical and structured software development approach. It emphasizes completing each phase fully before moving on to the next, ensuring consistency and thorough documentation throughout the lifecycle.

- **Requirement Analysis:** In this initial phase, all functional and non-functional requirements of the system are collected and documented. This includes understanding user needs, defining system objectives such as lost/found reporting, AI-based matching, user notifications, and chat functionality.
- **System and Software Design:** The system architecture is designed based on the collected requirements. This involves designing UI mockups, database schema, module structure, and data flow diagrams. Decisions are made on the technology stack and system components.
- **Implementation:** The application is built in modules, aligning with the system design. Features such as user authentication, item posting, smart matching, and messaging are implemented. Both front-end and back-end development are carried out in this phase.

- **Testing:** After implementation, the system undergoes various levels of testing including unit testing, integration testing, and system testing. The goal is to ensure the application behaves as expected, meets requirements, and is free from critical bugs.

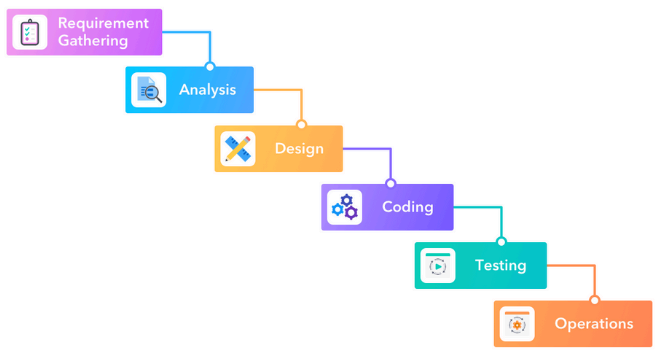


Figure 1.1: Overview of the Waterfall Development Model [1]

1.6 Conclusion

The Lost & Found Mobile App aims to streamline and modernize the item recovery process using a smart, centralized, and AI-assisted platform. It provides essential functionalities missing from existing systems and introduces new features to improve item recovery success rates and user experience. Adopting the Waterfall methodology allows us to maintain structure and clarity throughout the development lifecycle.

Chapter 2

Requirements Specification

2.1 Introduction

A deep understanding of requirements and a well-planned architecture significantly increase a system's or software's likelihood of success. This chapter aims to achieve precisely that. We'll begin by outlining the functional and non-functional requirements of our software, followed by a presentation of our use case diagram. Finally, we'll detail the architecture and environment of our system.

2.2 Specification of Requirements

This section will first identify our solution's stakeholders. Following that, we will define the essential functional and non-functional requirements our software must fulfill.

2.2.1 Identification of Actors

Our application involves two primary types of actors, each with distinct roles and responsibilities:

- **User:** This is the general user of the application. They can **register and log in**, browse and search for items, **report lost or found items**, and **receive notifications** when a match is found. They can also share posts about lost or found items.
- **Administrator:** This actor is responsible for the overall management and moderation of the system. Their duties include **moderating user-generated content** to prevent spam (potentially using anomaly detection), **managing user accounts** (including actions like banning abusive users), and **accessing system dashboards** for an overview of activity and performance.

2.2.2 Functional Requirements

he Lost & Found Mobile App provides a robust platform for users to report and search for lost or found items, significantly increasing the chances of reuniting them with their rightful owners. The system's functionalities are designed to cater to the specific needs of its primary actors:

For the User:

- **Register:** Users can create a new account on the platform to access its full range of features.
- **Login:** Users can securely log in to their account using traditional credentials or social media (OAuth).
- **Report Lost or Found Item:** Users can create a new post for a lost or found item by providing a detailed description, uploading photos, and specifying the location. This process can be manual or AI-assisted for efficiency.
- **Browse and Search Items:** Users can explore the list of reported items and utilize various filters (e.g., category, location, date, color) to narrow down search results. Results are sorted by relevance.
- **Receive Notifications:** Users are notified when a potentially matching item is found, or for other relevant updates regarding their reported items.
- **Communicate via Chat:** Users can directly interact with other users through an integrated messaging system to facilitate the return of items.

For the Administrator:

- **Manage User Accounts:** Administrators have the ability to manage user accounts, including actions such as banning abusive users to maintain a safe environment.
- **Access Dashboard:** Administrators can access a comprehensive dashboard providing an overview of system activity, performance metrics, and relevant statistics.

2.2.3 Non-Functional Requirements

Non-functional requirements define the quality attributes and constraints that ensure the Lost & Found Mobile App operates effectively and provides a positive user experience.

- **Usability (Ergonomics):** The application must provide an intuitive and easy-to-navigate user interface, allowing users to report, search for, and manage items with minimal effort and confusion. This includes clear visual cues, consistent layouts, and straightforward workflows.

- **Performance (Responsiveness):** The system must ensure quick response times for all user interactions, such as reporting an item, searching, or loading item details. Search results and map displays should load within a few seconds to ensure user satisfaction.
- **Real-time Updates:** Notifications for matching items, chat messages, and any changes in item status must be delivered to users in real-time to facilitate communication.
- **Security:** User data, including personal information and item details, must be protected against unauthorized access, modification, or disclosure.

2.3 Global Use Case Diagram

Figure 2.1 presents the global use case diagram of our Lost amp; Found Mobile App, visually outlining the interactions between various users and the system. The diagram identifies two main actors: the Account Holder (which encompasses both general users and administrators) and more specifically, the User and Administrator.

The User actor represents individuals who utilize the core functionalities of the application. This includes general actions such as Registering and Logging in to their account, enabling them to access the platform's features. Once authenticated, a User can Report an item, which is further specialized into Report lost item and Report found item, allowing them to specify the nature of the item. Users can also Share a post about an item, which can be either a Share post about a found item or a Share post about a lost item, to increase visibility. Additionally, Users can Search an item using various criteria to find specific items, View a map to see item locations (which extends to See item location for more detail), and Use chat to communicate with other users, specifically to Send a message and Receive a message.

The Administrator actor is responsible for the overall management and oversight of the application. Their key interactions include Manage users, which further specializes into Ban users and Allow users to control user access. They can also View dashboard to gain insights into system activity and Moderate posts to ensure content quality and adherence to platform guidelines. In summary, this diagram clearly illustrates the roles and responsibilities of each actor within the Lost amp; Found Mobile App, highlighting the specific functionalities they can perform to facilitate item reporting, searching, and user management.

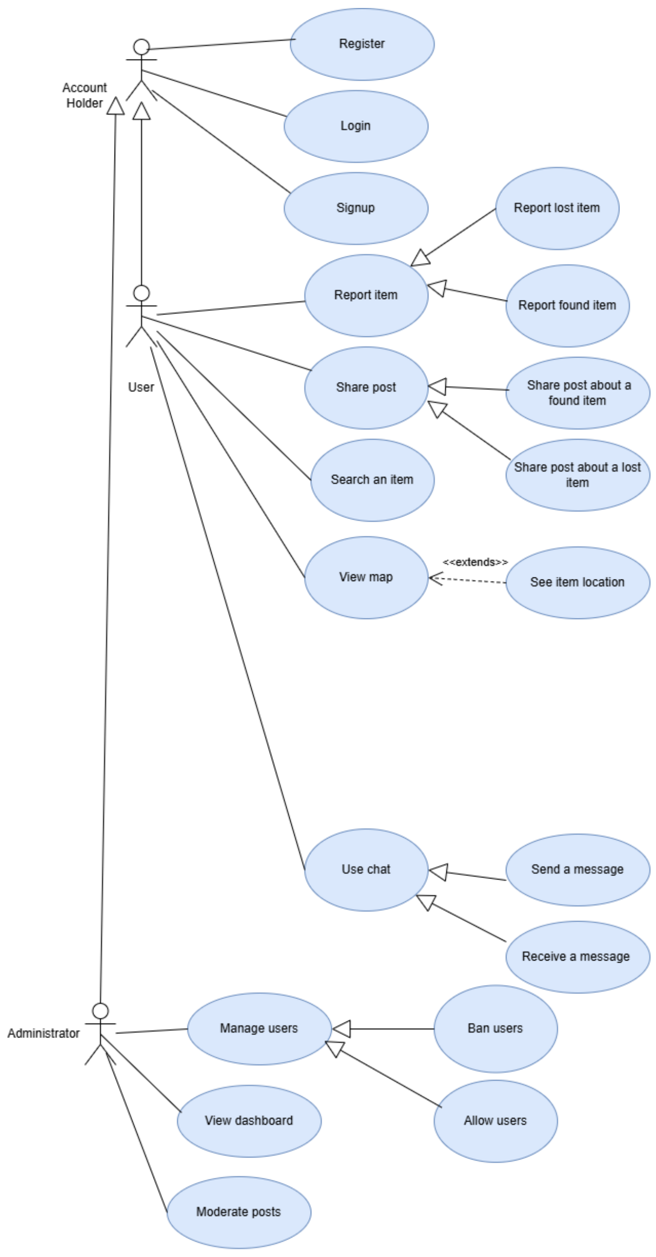


Figure 2.1: Global Use Case Diagram of the Lost & Found Mobile App

2.4 Software Environment

The software environment for our project encompasses all the essential tools for successful development, enabling efficient creation of the application's functionality and logic.

2.4.1 Organization and Design Tools

To achieve our project objectives, we utilized tools that foster collaboration and effective design:

This tool helps in visually representing the system's architecture, behavior, and interactions, making it easier for team members to understand and communicate complex designs.

Table 2.1: Organization and Design Tools

Tools	Logos	Description
draw.io		Tool for drawing diagrams and flowcharts.
Figma		Powerful collaborative interface design and prototyping tool.

2.4.2 Development Tools

Modern software development and testing heavily rely on robust tools to streamline the coding process, manage version control, and ensure thorough testing.

Table 2.2: Development Tools

Tools	Logos	Description
VS Code		Lightweight and powerful source code editor with integrated support for Git and extensions.
Git		Version control for code.
GitHub		Project collaboration and code hosting.

2.4.3 Front-end Tools

Developing a modern and user-friendly interface demands sophisticated front-end tools that enhance aesthetics and ensure a seamless, responsive user experience.

Table 2.3: Front-end Tools

Tools	Logos	Description
Flutter		An open-source UI software development kit developed by Google for building natively compiled applications for mobile, web, and desktop from a single codebase.

Flutter was chosen for its ability to create a consistent and engaging user interface across different platforms, leveraging a single codebase for efficiency.

2.4.4 Back-end Tools

Back-end development is pivotal for ensuring application performance, scalability, and security, involving server-side logic, database interactions, and integration with external services.

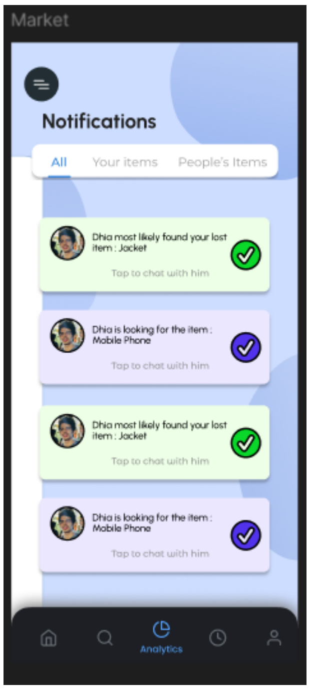
Table 2.4: Back-end Tools

Tools	Logos	Description
Firebase		Cloud-hosted NoSQL database that synchronizes data in real-time between users, enabling instant updates and live collaboration. Also provides robust user authentication.
FastAPI		Modern, high-performance web framework for building APIs with automatic OpenAPI documentation generation, built-in data validation, and asynchronous request handling capabilities.
Redis		Redis is an in-memory data structure store that functions as a high-performance cache. it is used to support our suspicious user detection service by providing temporary storage for time-sensitive data. This approach ensures fast data retrieval during detection processes while automatically expiring data that is no longer needed, making it optimal for our real-time security analysis requirements.
Neo4j		Graph database platform for storing and querying similarity relationships between posts, enabling efficient traversal of connection patterns and caching of similarity search results.
Qdrant		Vector similarity search engine optimized for high-dimensional embeddings, providing fast nearest neighbor searches for CLIP-generated multi-modal embeddings with metadata filtering capabilities.
Microsoft Azure		Cloud computing platform hosting our microservices infrastructure, providing scalable compute resources, container orchestration, and managed database services with enterprise-grade security.
RabbitMQ		Message broker facilitating asynchronous communication between microservices, ensuring reliable message delivery, load balancing, and decoupled service architecture through publish-subscribe patterns.

Our back-end architecture leverages a comprehensive toolkit of modern technologies. **Firebase** provides managed real-time data synchronization and authentication, while **FastAPI** serves as our high-performance API framework. **Redis** temporarily stores data to enable fast processing and analysis. **Neo4j** stores the relationships between similar posts, **Qdrant** for vector similarity searches, and **Firebase** for real-time data. **RabbitMQ** orchestrates asynchronous communication between our microservices, and **Microsoft Azure** provides effective storage solutions for unstructured data, particularly for storing post images in our application. This combination enables our microservices to focus on domain-specific logic while leveraging specialized tools for their respective strengths.

2.5 Prototyping

Prototyping in Figma was crucial for visualizing the user flow and interactions within our lost and found application. It allowed us to create interactive mockups that simulated the app's functionality, enabling us to test and refine the user experience early in the development process. By linking different screens and defining transitions, we could get a clear understanding of how users would navigate through reporting a lost item, claiming a found item, and browsing the market. This iterative approach helped us identify potential usability issues and make necessary adjustments before committing to development.



Dhia most likely found your lost item : Jacket
Tap to chat with him

Dhia is looking for the item : Mobile Phone
Tap to chat with him

Dhia most likely found your lost item : Jacket
Tap to chat with him

Dhia is looking for the item : Mobile Phone
Tap to chat with him

Figure 2.2: Notifications Screen Prototype

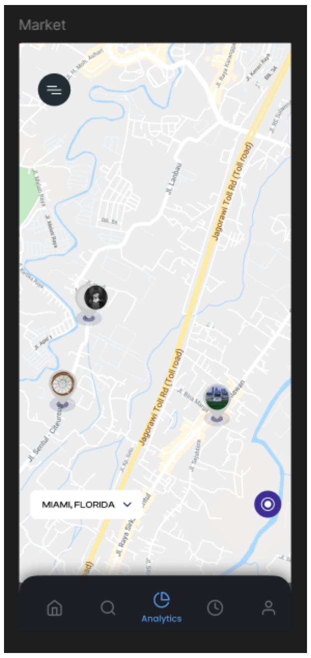


Figure 2.3: Location Selection Screen Prototype

2.6 Conclusion

In conclusion, this chapter successfully outlined the functional and non-functional requirements of our system. We also established the project's planning using the Scrum methodology and detailed the application's architecture and environment. The next chapter will focus on the development of our first release.

Chapter 3

System Design

This chapter presents the comprehensive system design for our lost and found application, detailing the architectural decisions, component interactions, and implementation structure that enables efficient item matching and user behavior analysis.

3.1 System Architecture Overview

The system's architecture is meticulously designed around a **microservices pattern**, a choice that significantly enhances scalability, resilience, and the ability for independent deployment of distinct functionalities. This architectural approach ensures a highly modular and maintainable system. Figure 3.1 provides a high-level overview of the implemented components and their interactions, directly illustrating the logical and physical structure derived from our system's design. This clear separation of concerns underpins the system's robustness and flexibility.

The core components comprising this architecture include:

- **Frontend (Flutter Mobile Application)**: The user-facing mobile application providing the primary interface for interaction with the lost and found system.
- **Firebase Backend**: Serves as the central backend system handling all the core business logic for the lost and found application, including data management, user operations, and service orchestration.
- **Gateway**: Acts as the central API entry point, efficiently routing external requests between Firebase and specialized microservices.
- **RabbitMQ**: A crucial message broker facilitating asynchronous and decoupled communication among various microservices.
- **Suspicious User Detection Service**: A specialized microservice dedicated to identifying and flagging anomalous user behaviors.

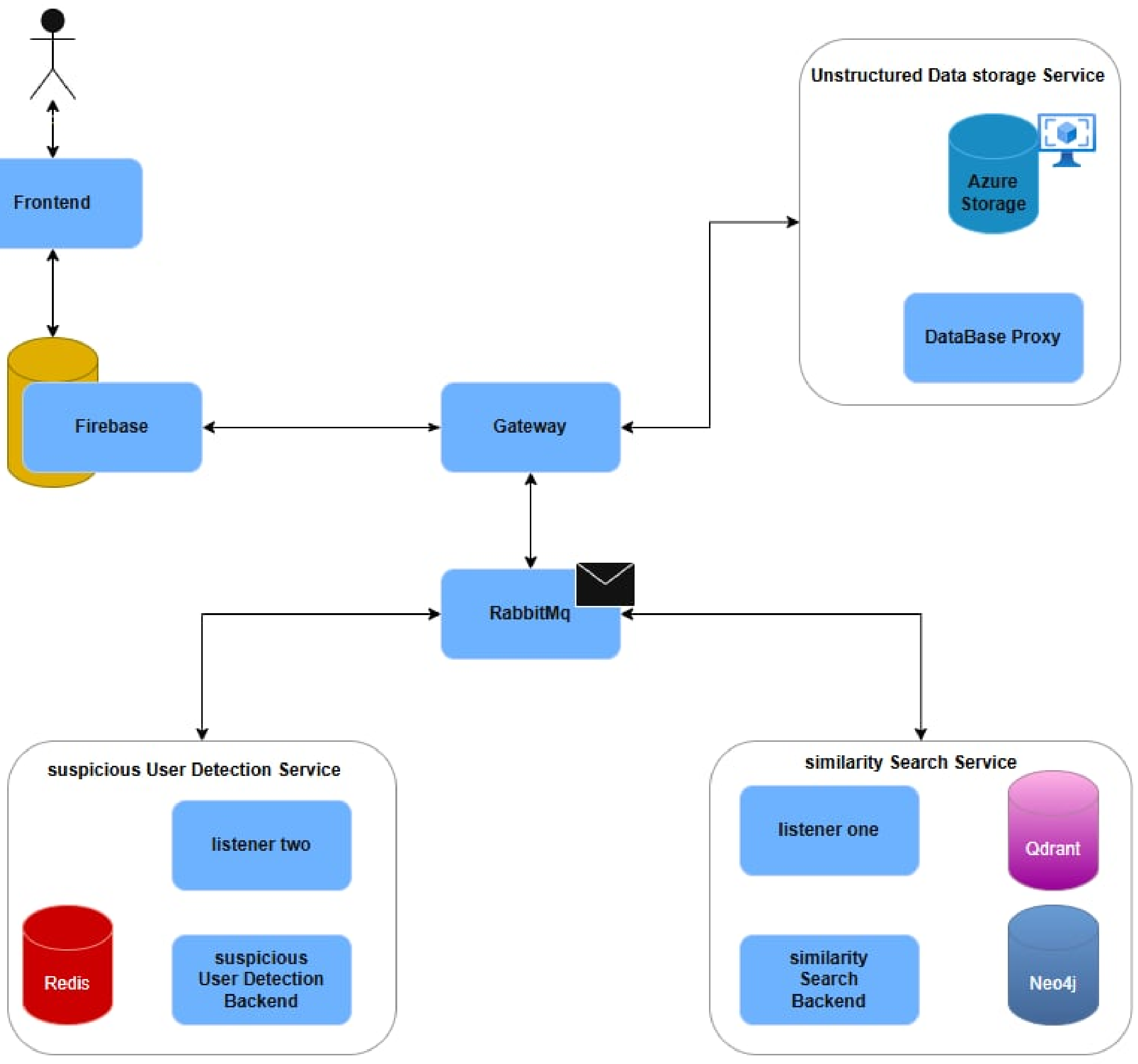

Unstructured Data storage Service
Azure Storage
DataBase Proxy
Frontend
Firebase
Gateway
RabbitMq
suspicious User Detection Service
listener two
Redis
suspicious User Detection Backend
similarity Search Service
listener one
Qdrant
similarity Search Backend
Neo4j

Figure 3.1: Implemented System Architecture Overview

- **Similarity Search Service**: A key microservice responsible for advanced similarity-based searches, particularly for item matching.
- **Unstructured Data Storage Service**: Manages the persistent storage and retrieval of diverse unstructured data types.

This setup ensures loose coupling between components, promoting independent development and deployment, and enabling specialized handling of diverse concerns.

3.2 Architectural Patterns

3.2.1 Microservices Architecture

The adoption of the microservices pattern provides several key advantages for our system. Each service can be developed, deployed, and scaled independently, allowing for specialized handling of different functional requirements. This approach enhances system resilience by isolating potential failures and enables the use of different technology stacks optimized for specific tasks.

3.2.2 MSP Pattern Implementation

Our project adopted the MSP (Models-Services-Providers) architectural pattern to ensure a clear separation of concerns. This allowed us to structure our application where Models handle data, Services contain the business logic and orchestrate operations on these models, and Providers abstract external dependencies, enabling flexible integration with various backend systems. This division of labor helps maintain a clean architecture, making the application easier to develop, test, and scale.

3.3 System Components

The overall system is meticulously decomposed into several distinct microservices and associated data stores, each assigned specific responsibilities. This modular breakdown contributes significantly to the comprehensive fulfillment of the functional requirements.

3.3.1 Frontend Module (Flutter Mobile Application)

Purpose: Acts as the primary interface through which users interact with the lost and found application. It provides a comprehensive mobile experience for posting lost items, searching for found items, and managing user interactions within the system.

Key Functionalities:

- User authentication and session management through Firebase integration
- Lost item posting with image capture and detailed descriptions
- Found item browsing and search functionality
- Real-time notifications and messaging between users
- Interactive form handling for item submissions and user profiles

- Responsive display of search results and item matches

Technologies Used: Flutter (for cross-platform mobile development), integrated with the Firebase SDK for backend communication.

3.3.2 Firebase Backend System

Purpose: Serves as the comprehensive backend solution that orchestrates all the core business logic for the lost and found application. Firebase is responsible for managing user posts creation, implementing the application's core logic, and coordinating with specialized microservices to obtain essential information such as similarity search results and suspicious user tracking data that enhance the system's decision-making capabilities.

Key Functionalities:

- **Authentication & User Management**: Secure user registration, login processes, session management, and comprehensive user profile handling
- **Post Creation & Management**: Complete lifecycle management of lost/found item postings, including creation, validation, storage, and metadata management
- **Core Application Logic**: Implementation of the primary lost and found business rules, matching algorithms, and workflow orchestration
- **Real-time Data Synchronization**: Live updates between users for new postings, matches, notifications, and system alerts
- **Decision Engine**: Processes information from microservices to determine appropriate actions such as flagging suspicious posts, prioritizing high-confidence matches, or triggering security measures
- **Data Storage**: NoSQL data storage for user profiles, item postings, application-specific data, and cached results from microservices

Dependency on Microservices: Firebase relies on specialized microservices to enhance its decision-making capabilities by incorporating advanced AI-powered similarity analysis and behavioral monitoring data into the core application logic.

Technologies Used: Google Firebase platform (Firestore, Authentication, Cloud Functions, Real-time Database).

3.3.3 API Gateway

Purpose: Functions as the intermediary API Gateway, serving as the communication bridge between Firebase and the specialized microservices ecosystem. It intelligently routes incoming requests from Firebase to the appropriate microservices via a messaging system (RabbitMQ). Once the target service processes the request and sends back the response through RabbitMQ, the Gateway captures the result and returns it to Firebase through an HTTP response.

Key Functionalities:

- **Service Orchestration**: Routes requests from Firebase to appropriate specialized microservices
- **Request Preprocessing**: Handles data transformation and validation before forwarding to microservices
- **Response Integration**: Processes results from microservices and formats them for Firebase consumption
- **Message Publishing**: Reliable message publishing to RabbitMQ queues for asynchronous processing

Technologies Used: Python with FastAPI (for building a high-performance, asynchronous API).

3.3.4 Message Broker Implementation (RabbitMQ)

Purpose: RabbitMQ serves as the central message broker, facilitating highly asynchronous and decoupled communication between the Gateway and the various specialized microservices. It ensures reliable message delivery and queuing requests, thereby significantly improving overall system resilience, scalability, and responsiveness.

Key Functionalities:

- **Message Queuing**: Efficient queuing of similarity search and suspicious user detection requests
- **Intelligent Routing**: Message routing based on predefined rules to appropriate service queues
- **Reliable Delivery**: Guarantees message delivery even under varying loads and service availability
- **Bidirectional Communication**: Supports both request queuing and result delivery back to requesting services

RabbitMQ Message Flow Architecture

Based on the message flow architecture shown in Figure 3.2, the RabbitMQ implementation follows a structured queue-based communication pattern that enables efficient task distribution and result collection.

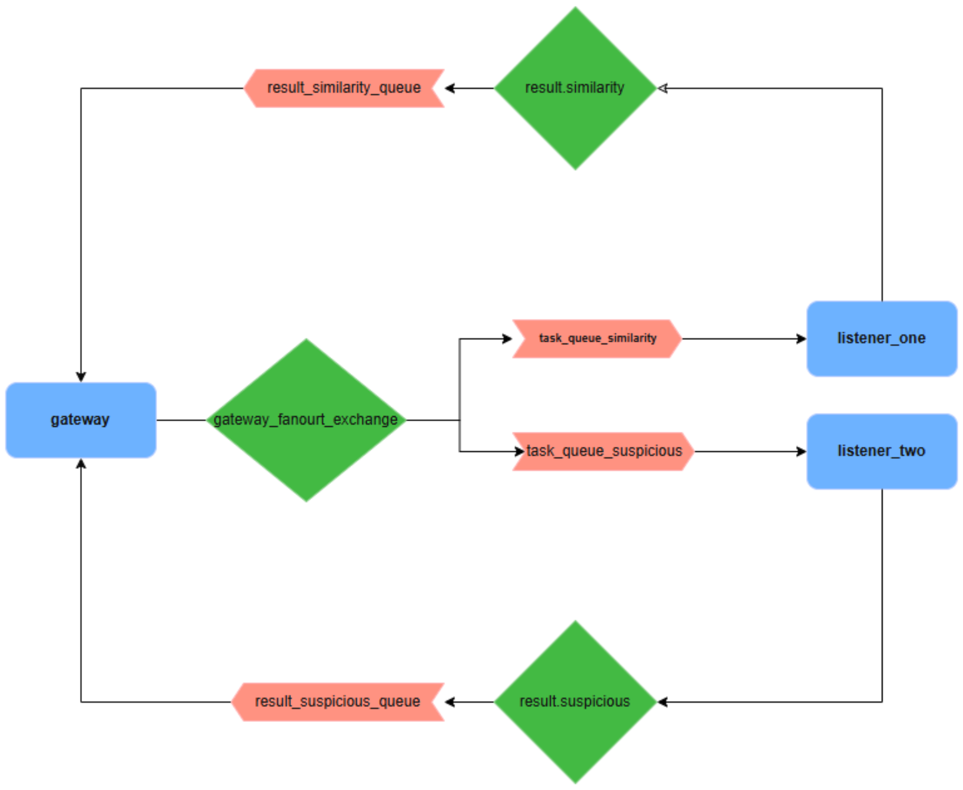


Figure 3.2: RabbitMQ Message Flow Architecture

Message Routing Strategy: The gateway acts as the central message dispatcher through the `gateway_fanout_exchange`, which intelligently routes incoming requests to specialized task queues:

- `task_queue_similarity`: Routes similarity search requests to `listener_one` for processing by the Similarity Search Service
- `task_queue_suspicious`: Routes user behavior analysis requests to `listener_two` for processing by the Suspicious User Detection Service

Bidirectional Communication: The system implements dedicated result queues for asynchronous response handling:

- `result_similarity_queue`: Receives processed similarity search results from the Similarity Search Service

- `result_suspicious_queue`: Receives analysis results from the Suspicious User Detection Service

This bidirectional communication pattern ensures that results are properly routed back to the gateway for Firebase integration, maintaining system responsiveness while enabling parallel processing of multiple service requests.

Technologies Used: RabbitMQ.

3.3.5 Suspicious User Detection Service

Purpose: A specialized microservice dedicated to real-time analysis of user behavior patterns to identify and flag suspicious or anomalous activities. This service enhances Firebase's security capabilities by providing advanced behavioral analysis that helps maintain system integrity and user safety.

Key Functionalities:

- **Behavioral Pattern Analysis**: Monitors user activities received from Firebase via RabbitMQ to identify unusual patterns
- **Anomaly Detection**: Applies sophisticated algorithms to detect suspicious behaviors such as rapid posting, or spam activities
- **Risk Scoring**: Generates risk scores for users based on their behavioral patterns and activity history
- **Real-time Alerting**: Provides immediate notifications to Firebase when suspicious activities are detected
- **Profile Management**: Maintains and updates suspicious user profiles for ongoing monitoring

Service Components

- **Listener Two**: Subscribes to and consumes user activity messages from RabbitMQ, processes behavioral data, and publishes analysis results back to the message queue
- **Suspicious User Detection Backend**: Contains the core business logic, statistical models, and AI/ML algorithms for anomaly detection
- **Redis Cache**: Provides high-performance temporary storage for user activity data and behavioral analysis results

Technologies Used: Python with FastAPI (for backend logic and ML models), Redis for caching.

3.3.6 Similarity Search Service

Purpose: A specialized microservice designed to perform intelligent similarity matching for lost and found items through advanced multi-modal AI processing. This service significantly enhances Firebase's matching capabilities by combining computer vision and natural language processing to create comprehensive item representations, enabling precise matching between lost and found posts.

Key Functionalities

- **Multi-Modal Embedding Generation**: Processes item posts by extracting and combining visual features from images and semantic features from text descriptions using CLIP (Contrastive Language-Image Pre-training) model to create unified 512-dimensional vector embeddings
- **Vector Storage and Retrieval**: Stores generated embeddings in Qdrant vector database with associated metadata, enabling efficient similarity searches through high-dimensional vector space comparisons
- **Intelligent Similarity Matching**: Performs semantic similarity searches by comparing query embeddings against stored vectors, identifying the most relevant matches based on combined visual and textual similarity
- **Graph-Based Result Caching**: Stores similarity relationships as nodes and edges in Neo4j graph database, creating a persistent similarity network that enables instant retrieval of previously computed matches
- **Firebase Integration**: Processes similarity requests from Firebase via the Gateway and RabbitMQ, returning enhanced matching results that improve the core lost and found functionality
- **Performance Optimization**: Leverages graph structure and vector indexing to provide rapid results for repeated queries, significantly reducing computational overhead

Service Components

- **Listener One**: Subscribes to and consumes similarity search requests from RabbitMQ, triggers processing tasks, and publishes results back to the message queue
- **Similarity Search Backend**: Contains sophisticated logic for processing search requests and orchestrating interactions with vector and graph databases
- **Qdrant Vector Database**: High-performance vector similarity search engine optimized for finding nearest neighbors in high-dimensional spaces

- **Neo4j Graph Database**: Stores relationships between similar items and enables complex similarity network analysis

Technologies Used: Python with FastAPI (for backend logic), Qdrant (vector database), Neo4j (graph database).

3.3.7 Unstructured Data Storage Service

Purpose: This service is specifically designed to manage the secure and scalable storage and retrieval of unstructured data, predominantly including images of lost/found items, documents, and other large binary objects. It works in coordination with Firebase to provide enhanced storage capabilities beyond the standard Firebase storage limits.

Key Functionalities:

- **Scalable File Storage**: Securely stores large binary objects including high-resolution images of lost and found items
- **Efficient Retrieval**: Provides fast file access based on unique identifiers for Firebase consumption
- **Access Control**: Manages fine-grained permissions and secure access to stored content
- **Content Optimization**: Handles image processing and optimization for mobile application consumption

Service Components

- **Azure Blob Storage**: Primary cloud-based storage solution providing enterprise-grade scalability and reliability
- **Storage Proxy**: Intermediary component that manages access to Azure Storage, adding caching and access control layers

Technologies Used: Azure Blob Storage SDK for programmatic interaction with cloud storage.

3.4 Data Storage Architecture

The system employs a polyglot persistence approach, utilizing different database technologies optimized for specific data types and access patterns:

3.4.1 Firebase Database Services

- **Firestore**: Primary database for user profiles, item postings, and core application data
- **Realtime Database**: Handles real-time synchronization for notifications and live updates
- **Authentication Database**: Manages user authentication data and session information

3.4.2 Specialized Storage Systems

- **Vector Database (Qdrant)**: Stores high-dimensional vector embeddings for similarity matching
- **Graph Database (Neo4j)**: Maintains similarity relationships and complex item connections
- **In-Memory Cache (Redis)**: Provides high-performance temporary storage for behavioral analysis
- **Cloud Storage (Azure Blob)**: Manages unstructured data storage for images and documents

3.5 System Integration and Communication

The system integration demonstrates how Firebase serves as the central hub for the lost and found application while leveraging specialized microservices to enhance its capabilities. This architecture ensures that:

- **Core Functionality**: Firebase handles all primary lost and found business logic, user management, and data storage
- **Enhanced Capabilities**: Specialized microservices provide advanced AI-powered matching and security features
- **Seamless Integration**: The Gateway and RabbitMQ ensure reliable communication between Firebase and enhancement services
- **Scalable Architecture**: Each component can be scaled independently based on demand and performance requirements

This comprehensive architecture effectively combines the reliability and ease of use of Firebase with the specialized capabilities of custom microservices, creating a robust and feature-rich lost and found application system. The asynchronous message-driven architecture ensures loose coupling between services while maintaining reliable communication patterns for complex multi-service operations.

Chapter 4

Class and Sequence Diagrams

4.1 Introduction

This chapter explores Class Diagrams and Sequence Diagrams, two essential tools in Unified Modeling Language (UML) for system modeling. Class diagrams provide a static view, mapping out the system's structure by showing classes, their attributes, operations, and relationships. In contrast, sequence diagrams offer a dynamic perspective, illustrating the time-ordered interactions and message flow between objects for specific scenarios.

4.2 Class Diagram

This class diagram provides a static view of the system's architecture, detailing the key entities and their relationships. It highlights core classes such as `AccountHolder`, `User`, `Administrator`, `Post`, `Item` (with specialized `LostItem` and `FoundItem`), `Notification`, `Chat`, and `ChatMessage`. The diagram depicts attributes for each class (e.g., `userId`, `postID`, `itemId`) and their respective operations (e.g., `register()`, `createPost()`, `sendMessage()`). Furthermore, it clearly defines the associations between these classes, such as an `Administrator` moderating `Post`s, an `AccountHolder` receiving `Notifications`, and `User`s participating in `Chat`s. This comprehensive structural overview is fundamental to understanding the system's design and data organization.

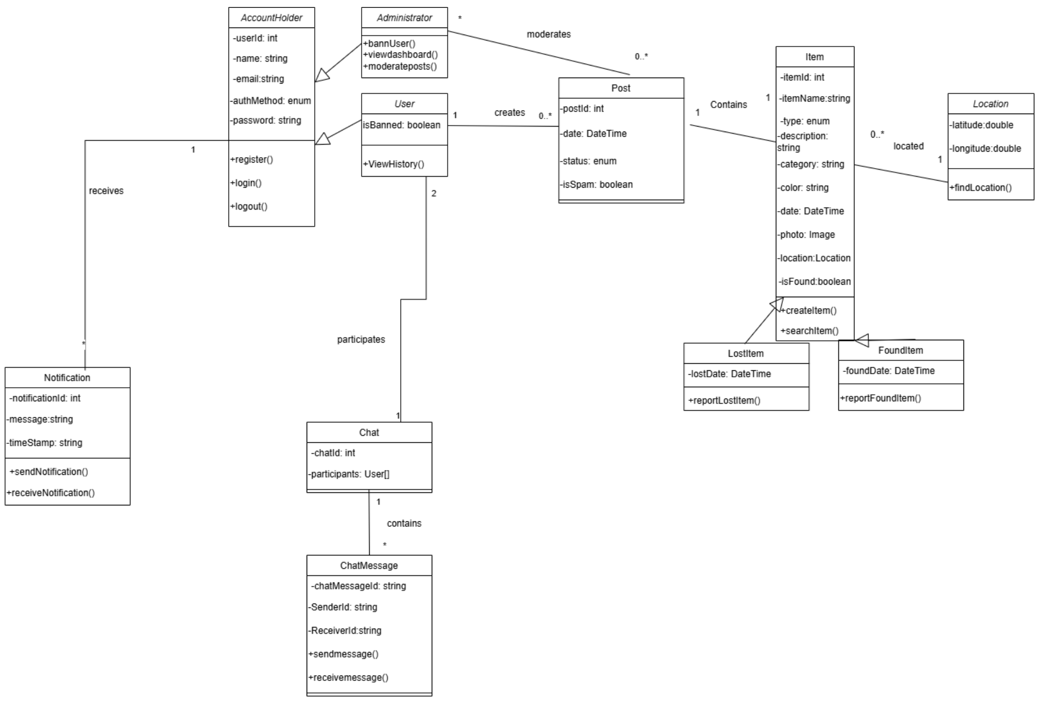


Figure 4.1: UML Class Diagram of the System

4.3 Sequence Diagrams

To complement the static structure, the following sequence diagrams illustrate the dynamic interactions within our applications for key scenarios.

4.3.1 Authentication Process

This sequence diagram meticulously details the authentication process, illustrating the step-by-step interactions between the user interface, state management, authentication service, and backend Firebase components. It particularly highlights the flow for email/password login, including successful authentication, fetching user data from Firestore, and handling various error conditions like banned user accounts.

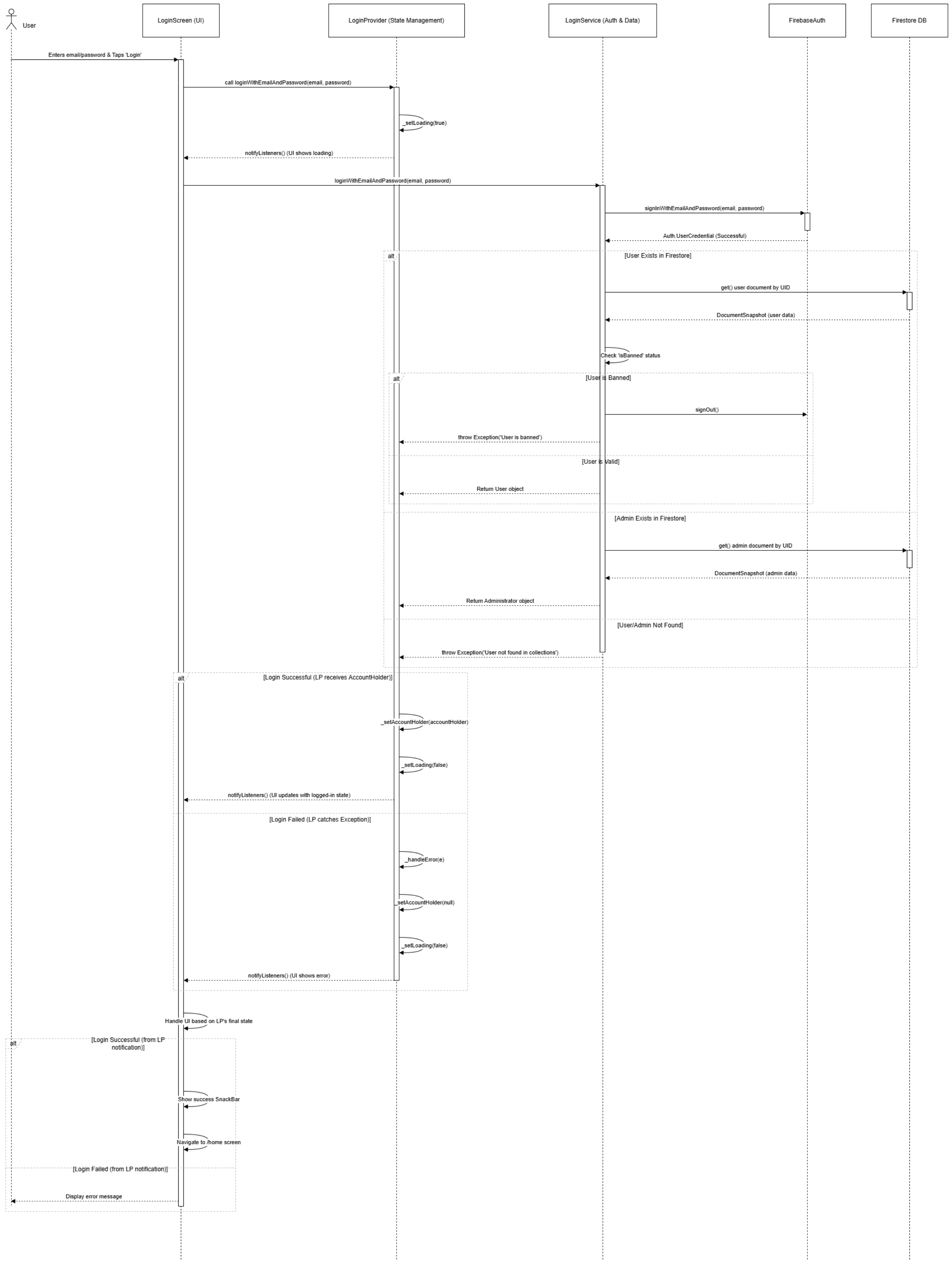

User
LoginScreen (UI)
LoginProvider (State Management)
LoginService (Auth & Data)
FirebaseAuth
Firestore DB
Enters email/password & Taps 'Login'
call loginWithEmailAndPassword(email, password)
_setLoading(true)
notifyListeners() (UI shows loading)
loginWithEmailAndPassword(email, password)
signInWithEmailAndPassword(email, password)
Auth.UserCredential (Successful)
alt
[User Exists in Firestore]
get() user document by UID
DocumentSnapshot (user data)
Check 'isBanned' status
alt
[User is Banned]
signOut()
throw Exception('User is banned')
[User is Valid]
Return User object
[Admin Exists in Firestore]
get() admin document by UID
DocumentSnapshot (admin data)
Return Administrator object
[User/Admin Not Found]
throw Exception('User not found in collections')
alt
[Login Successful (LP receives AccountHolder)]
_setAccountHolder(accountHolder)
_setLoading(false)
notifyListeners() (UI updates with logged-in state)
[Login Failed (LP catches Exception)]
_handleError(e)
_setAccountHolder(null)
_setLoading(false)
notifyListeners() (UI shows error)
Handle UI based on LP's final state
alt
[Login Successful (from LP notification)]
Show success SnackBar
Navigate to /home screen
[Login Failed (from LP notification)]
Display error message

Figure 4.2: UML Sequence Diagram of the Authentication Flow

4.3.2 Dashboard for Admin

This sequence diagram meticulously details the processes involved in displaying the dashboard for an administrator. It illustrates the step-by-step interactions between the user interface, state management, various data providers and the backend data store (Firestore). It particularly highlights the flow for fetching and displaying monthly statistics and item counts, including successful data retrieval and handling various error conditions like the absence of data or issues during data fetching.

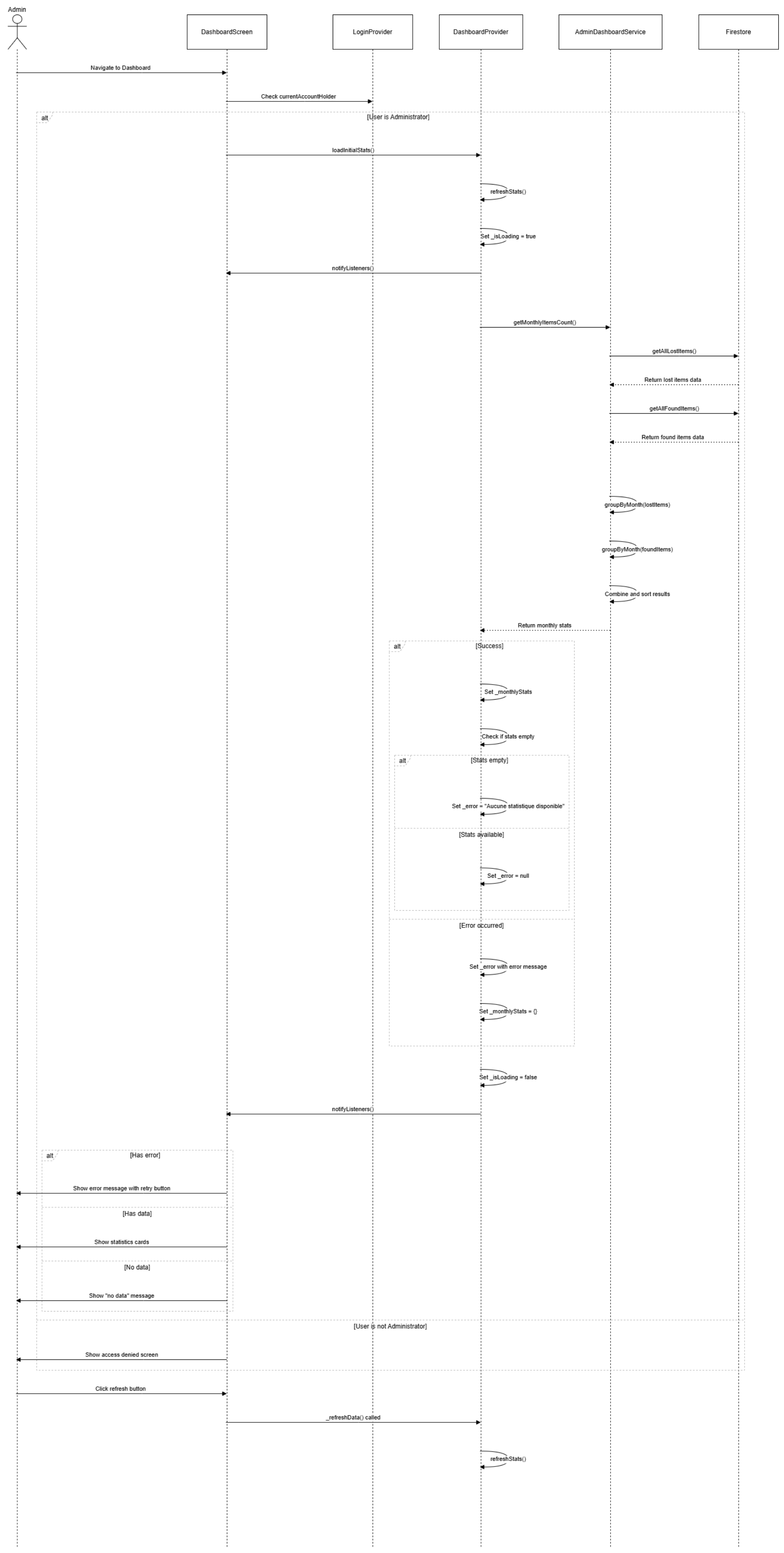


Figure 4.3: UML Sequence Diagram of the Dashboard Flow for Admin

4.3.3 Report Lost Item

This sequence diagram illustrates the process of reporting an item, from user initiation through various service interactions for image upload, data storage, and message processing. It details the steps involved in submitting an item report, including location determination, image handling, and subsequent processing by different backend services.

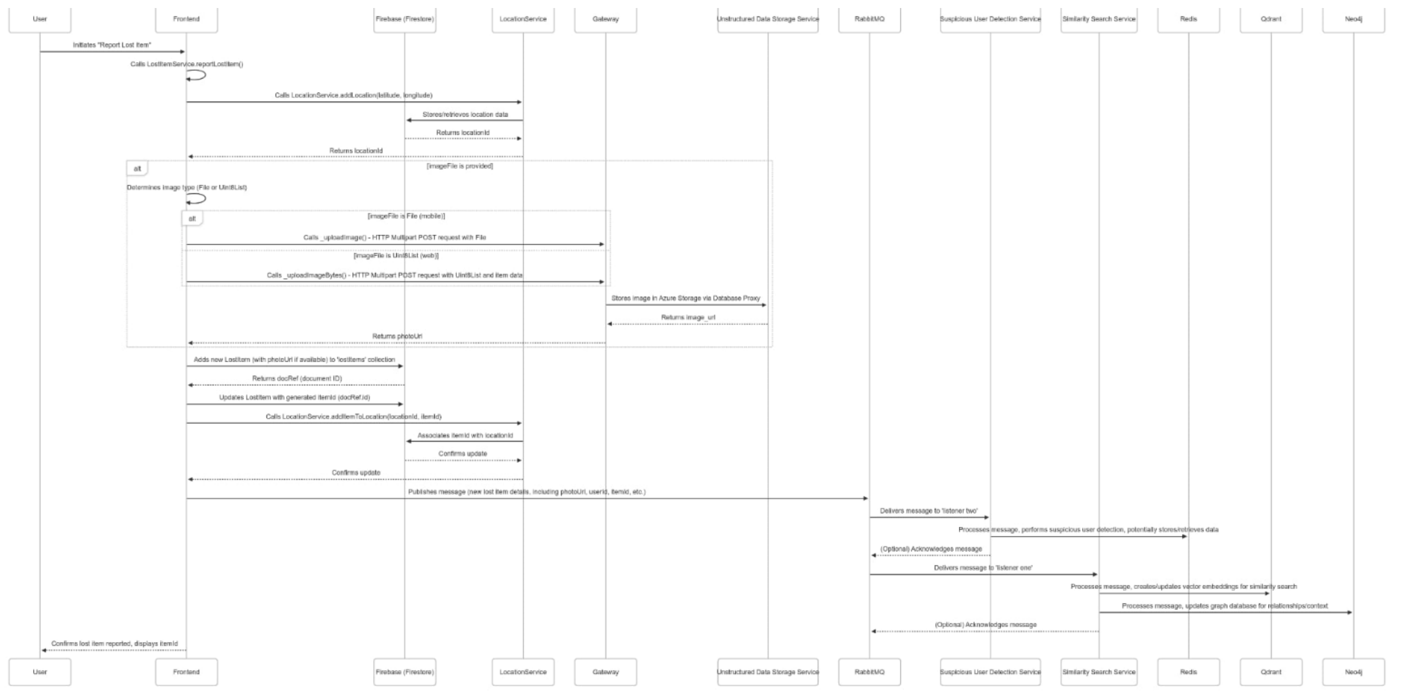

User
Frontend
Firebase (Firestore)
LocationService
Gateway
Unstructured Data Storage Service
RabbitMQ
Suspicious User Detection Service
Similarity Search Service
Redis
Qdrant
Neo4j
Initiates "Report Lost Item"
Calls LostItemService.reportLostItem()
Calls LocationService.addLocation(latitude, longitude)
Stores/retrieves location data
Returns locationId
Returns locationId
alt
[imageFile is provided]
Determines image type (File or Uint8List)
alt
[imageFile is File (mobile)]
Calls _uploadImage() - HTTP Multipart POST request with File
[imageFile is Uint8List (web)]
Calls _uploadImageBytes() - HTTP Multipart POST request with Uint8List and item data
Stores image in Azure Storage via Database Proxy
Returns image_url
Returns photoUrl
Adds new LostItem (with photoUrl if available) to 'lostItems' collection
Returns docRef (document ID)
Updates LostItem with generated itemId (docRef.id)
Calls LocationService.addItemToLocation(locationId, itemId)
Associates itemId with locationId
Confirms update
Confirms update
Publishes message (new lost item details, including photoUrl, userId, itemId, etc.)
Delivers message to 'listener two'
Processes message, performs suspicious user detection, potentially stores/retrieves data
(Optional) Acknowledges message
Delivers message to 'listener one'
Processes message, creates/updates vector embeddings for similarity search
Processes message, updates graph database for relationships/context
(Optional) Acknowledges message
Confirms lost item reported, displays itemId
User
Frontend
Firebase (Firestore)
LocationService
Gateway
Unstructured Data Storage Service
RabbitMQ
Suspicious User Detection Service
Similarity Search Service
Redis
Qdrant
Neo4j

Figure 4.4: UML Sequence Diagram of Report Lost Item

4.3.4 Chat feature

This sequence diagram illustrates the process of initiating and managing a chat between two users, detailing the interactions between the User, Frontend, and Firebase (Firestore). It outlines how a chat is created or retrieved, messages are sent and stored, and real-time message updates are handled to display conversations.

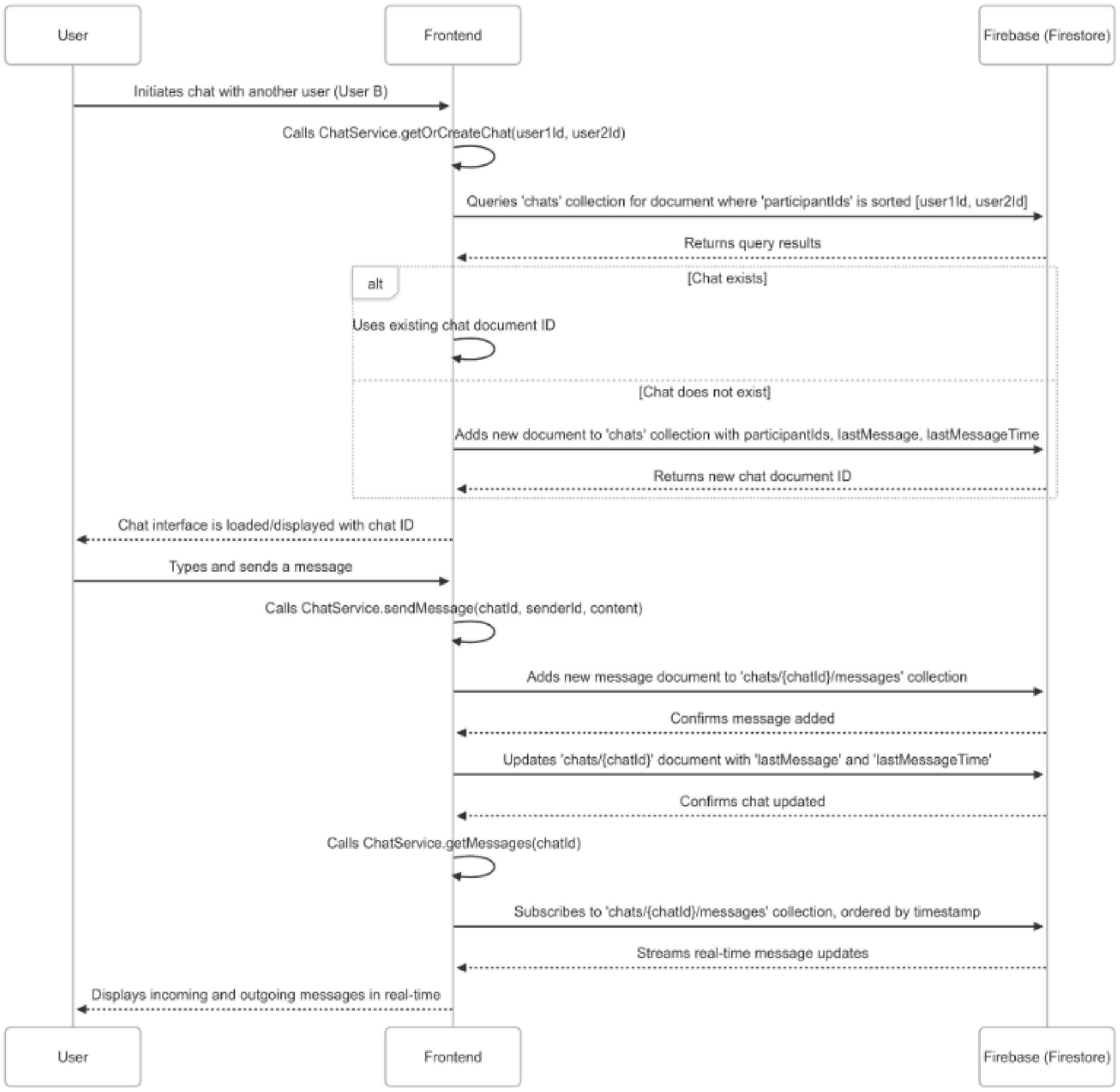

User
Frontend
Firebase (Firestore)
Initiates chat with another user (User B)
Calls ChatService.getOrCreateChat(user1Id, user2Id)
Queries 'chats' collection for document where 'participantIds' is sorted [user1Id, user2Id]
Returns query results
alt
[Chat exists]
Uses existing chat document ID
[Chat does not exist]
Adds new document to 'chats' collection with participantIds, lastMessage, lastMessageTime
Returns new chat document ID
Chat interface is loaded/displayed with chat ID
Types and sends a message
Calls ChatService.sendMessage(chatId, senderId, content)
Adds new message document to 'chats/{chatId}/messages' collection
Confirms message added
Updates 'chats/{chatId}' document with 'lastMessage' and 'lastMessageTime'
Confirms chat updated
Calls ChatService.getMessages(chatId)
Subscribes to 'chats/{chatId}/messages' collection, ordered by timestamp
Streams real-time message updates
Displays incoming and outgoing messages in real-time
User
Frontend
Firebase (Firestore)

Figure 4.5: UML Sequence Diagram of Chat feature

4.4 Conclusion

This chapter presented the Class and Sequence Diagrams for our applications. These diagrams effectively illustrated the core structural components and their dynamic interactions, providing a clear visual representation of how our system is designed and operates.

Chapter 5

Artificial Intelligence Integration

5.1 Introduction

The integration of Artificial Intelligence (AI) within the Lost & Found Mobile Application plays a transformative role in enhancing the platform's efficiency, security, and user satisfaction. By embedding intelligent functionalities such as automated item matching and user behavior monitoring, the system transcends the capabilities of traditional mobile applications and offers a smart, adaptive environment for users seeking to recover or report lost belongings.

AI technologies have been modularly incorporated as microservices to maintain a clean separation of concerns, scalability, and ease of maintenance. This section explores the design, implementation, and advantages of the AI-powered components in depth.

5.1.1 General Architecture of AI Microservices

The AI modules are implemented as independent microservices that operate asynchronously and communicate with the core application through a message broker (RabbitMQ). This decoupling ensures that heavy computational workloads, such as image processing or model inference, do not interfere with the responsiveness of the core application.

Each AI microservice is designed with the following considerations:

- **Autonomy:** Services can be deployed, updated, or scaled independently.
- **Reusability:** AI models and components are built to be reused in multiple contexts (e.g., both in reporting and search functionalities).
- **Resilience:** Failures in AI services do not disrupt the core app; fallback mechanisms ensure continuity.

These services are:

- **Similarity Search Microservice**
- **Suspicious User Detection Microservice**

5.1.2 Similarity Search Microservice

The Similarity Search Microservice is the backbone of intelligent item matching in the system. When users submit a report for a lost or found item, this service analyzes the content and searches for existing reports that exhibit high similarity.

Objective and Motivation

Traditional keyword-based search techniques often fail to capture the nuanced similarity between two item descriptions or images. For example, a user may report a "black leather wallet" while another describes it as a "dark men's wallet made of cowhide." AI bridges this gap by understanding semantics and visual patterns.

Functional Workflow

The microservice follows a multi-step process:

- **Data Ingestion:** When a report is created, its data (title, description, category, and image) is sent to the AI service via a RabbitMQ queue.
- **Feature Extraction with CLIP:** Both the textual descriptions and the uploaded image are processed using OpenAI's CLIP model, which generates joint image-text 512-dimensional vector embeddings that capture semantic and visual features in a shared vector space.
- **Similarity Computation:** Cosine similarity is used to measure the semantic closeness between the new item's embedding and those of existing items in the database. The embeddings are stored and indexed in Qdrant, which utilizes the Hierarchical Navigable Small World (HNSW) algorithm a graph-based approximate nearest neighbor search technique. This indexing enables efficient, scalable, and low-latency retrieval of the most similar vectors. Additionally, Qdrant supports filtering based on metadata, allowing searches to be constrained to relevant subsets of data. This combination ensures both accuracy and high performance in similarity search across large-scale vector datasets.
- **Ranking and Filtering:** The system retrieves the top-N most similar items based on cosine similarity scores computed by Qdrant's HNSW index. It applies configurable similarity thresholds

and dynamic metadata-based filters to ensure that only relevant and high-confidence matches are returned, reducing false positives and enhancing the overall precision of the results.

Technical Stack

- **Languages & Frameworks:**
  - Python
  - FastAPI (for API development)
- **Libraries & Tools:**
  - `torch` and `clip` (for loading and using the CLIP model)
  - `Pillow` (image processing)
  - `numpy` (numerical array operations)
  - `io` (byte stream processing)
- **Models:**
  - OpenAI CLIP (Vision Transformer ViT-B/32) for joint image-text embeddings, As explained in [2].
- **Storage:**
  - Qdrant (stores 512-dimensional vector embeddings with metadata)
  - Neo4j (stores similarity relationships as a graph)

5.1.3 Suspicious User Detection Microservice

This microservice is responsible for monitoring user activities to detect abnormal behavior that may indicate spam, fraud, or other forms of misuse. This component supports the trustworthiness and security of the platform.

Use Case Scenarios

- A user repeatedly posts similar items with inconsistent metadata.
- reuse of same picture in multiple posts.
- a lot of posts in a short period of time
- Big rewards or links in description.

Data Collection and Preprocessing

The data used for anomaly detection in suspicious user behavior includes multiple indicators derived from user posts:

- Frequency and timing of posts, tracking how many posts a user makes within a 24-hour period.
- Detection of duplicate images posted by the same user, based on perceptual hashing (pHash).
- Presence of external links within post content.
- Textual content length, linguistic patterns, and semantic analysis via a large language model (LLM).

To efficiently support these detections, the system leverages Redis as an in-memory data store:

- Image perceptual hashes are stored in Redis sets keyed by user ID, enabling quick duplicate detection without recalculating hashes for every post. These sets expire automatically after 30 days.
- Post frequency counters are stored with a 24-hour time-to-live (TTL), resetting daily to monitor daily posting activity.
- Flags for posts containing external links are tracked per user for 30 days to identify repeated external link usage.

Preprocessing steps include semantic analysis through the LLM, which assesses the contextual suspiciousness of user posts beyond simple heuristics. The LLM's output is validated against a strict JSON schema to ensure reliable automated decision-making.

Model Design The system employs a modular anomaly detection approach combining rule-based tools and a Large Language Model (LLM) for nuanced analysis:

- **Rule-based Tools:** Custom asynchronous functions detect specific suspicious behaviors such as duplicate image posting (using perceptual hashing), excessive posting frequency, and presence of external links. These tools track user activity and state via Redis to enable efficient real-time checks.
- **LLM Integration:** A LLaMA-based LLM is used to analyze post content for subtle signs of fraud or scam by generating a structured JSON assessment. This supplements the rule-based signals with deeper natural language understanding and reasoning.

- **Orchestration:** The detection pipeline manually invokes these tools and the LLM sequentially, aggregating results into a unified anomaly response. While LangChain is used for LLM calls, the system does not yet implement dynamic tool selection, context management, or multi-turn memory typical of a full LangChain agent architecture.

This design allows for scalable, extensible detection that combines explicit heuristics with flexible AI-powered analysis.

Alert System and Admin Interface

When suspicious behavior is detected:

- A risk score is computed and compared against a tunable threshold.
- An alert is sent to the admin dashboard, including logs, metadata, and anomaly explanations.
- The admin may choose to:
  - Issue a warning.
  - Temporarily suspend the account.
  - Mark the report for further human investigation.

Technical Stack

- **Languages & Frameworks:**
  - Python
  - FastAPI (for API development)
- **Libraries & Tools:**
  - `base64` (for decoding images)
  - `Pillow` (image processing)
  - `imagehash` (perceptual hashing for duplicate image detection)
  - `re` (regular expressions for link detection)
  - `langchain` and `langchain-openai` (LLM agent and prompt orchestration)
- **Models:**
  - LLaMA 3 (via Groq inference engine, As explained in [3]) used as the Large Language Model for fraud and scam detection, As explained in [4].

- **Storage:**
  - Redis (for caching user image hashes, post counts, and external link tracking)

5.1.4 Benefits and Strategic Impact of AI Integration

The incorporation of AI into the Lost & Found application provides numerous long-term advantages:

- **Increased Reclaim Success Rate:** AI-powered matching significantly raises the chances of a user recovering their lost item.
- **Time and Resource Optimization:** Automation reduces manual review and administrative overhead.
- **User Trust and Safety:** Anomaly detection builds a safer platform by reducing scams and harmful behavior.
- **Data-Driven Insights:** AI modules produce rich metadata and logs that can inform future features or marketing decisions.
- **Competitive Edge:** Integrating AI elevates the app's functionality beyond traditional platforms, offering a more intelligent, personalized experience.

5.1.5 Future Enhancements and Research Directions

The current AI-powered system establishes a robust and scalable foundation through two key services: *Suspicious User Detection*, which identifies anomalous behavior by leveraging behavioral modeling, and *Similarity Search*, which enables semantic content matching via CLIP embeddings and vector similarity using Qdrant. To further elevate the system's intelligence, efficiency, and personalization capabilities, several enhancements are under active consideration. In particular, the integration of high-performance inference backends such as *Groq* and *Ollama* will enable low-latency As explained in [5], on-device processing, while the adoption of advanced LLM orchestration frameworks like *LangChain* is expected to support context-aware reasoning, memory management, and autonomous agent capabilities. These directions will expand the system's ability to operate responsively and securely in real-world environments.

Federated Learning for Privacy-Preserving Behavior Modeling To enhance the *Suspicious User Detection* service while preserving user privacy, federated learning can be employed. This decentralized training approach allows behavioral models to learn directly on user devices, aggregating insights without transmitting raw data to central servers. This not only strengthens data confidentiality but also enables the system to continuously adapt to evolving usage patterns in a privacy-respecting manner.

Personalized Semantic Search and Recommendations The *Similarity Search* service can be further enriched by introducing personalized retrieval capabilities. By incorporating user preferences and interaction history, the system can refine its semantic ranking to deliver more relevant and context-aware results. This would significantly enhance user engagement and satisfaction by tailoring recommendations to individual behavioral profiles.

Extending LangChain Capabilities While **LangChain** is currently used to facilitate basic large language model (LLM) interactions, the system does not yet implement full agentic behavior. Planned enhancements include:

- **Dynamic Tool Selection**: Automatically selecting the most appropriate internal tools—such as vector databases, prompt engineering modules, or external APIs—based on user input and context.
- **Context-Aware Multi-Turn Memory**: Maintaining conversation history across multiple turns to support coherent, stateful dialogue with users.
- **Autonomous Agent Architecture**: Evolving toward a full LangChain agent setup to enable task decomposition, reasoning, and autonomous execution workflows.

5.2 Conclusion

These research directions aim to transform the system into a more intelligent, adaptive, and secure AI platform, well-suited for deployment in real-world environments with high demands for responsiveness, scalability, and ethical data usage.

Chapter 6

Implementation

6.1 Introduction

This chapter transitions from the theoretical design and requirements discussed in the previous chapters to the practical realization of the proposed solution. It provides a detailed account of how the system's architecture was translated into a tangible, working application. We will first identify the key modules and components and their specific responsibilities. Subsequently, this chapter will delve into the rigorous testing methodologies and debugging processes employed to ensure the robustness, reliability, and correctness of the implemented system. The aim is to demonstrate the successful conversion of design specifications into a functional and verifiable software product.

6.2 Key Modules and Functionalities

This section details the practical realization of the proposed solution, translating the requirements specified in Chapter 2 into a working system. It highlights the microservices architecture and specific components implemented, describing how each module contributes to the overall functionality as depicted in the system architecture diagram.

6.2.1 User Registration

This module handles the process of new users signing up for the platform. It involves collecting necessary user information and securely storing it, typically integrated with Firebase Authentication for robust user management.

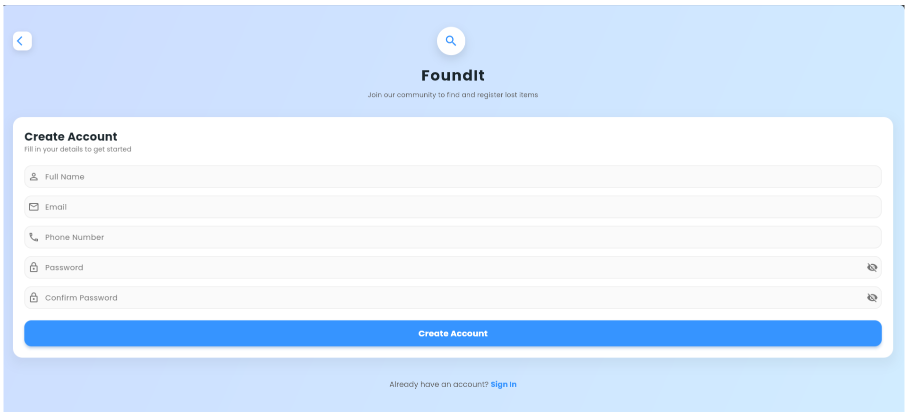

FoundIt
Join our community to find and register lost items
Create Account
Fill in your details to get started
Full Name
Email
Phone Number
Password
Confirm Password
Create Account
Already have an account? Sign In

Figure 6.1: Screenshot of the User Registration page.

6.2.2 User Login

The login module provides authenticated access to the system for registered users. It verifies user credentials against the stored records, enabling access to personalized features and content.

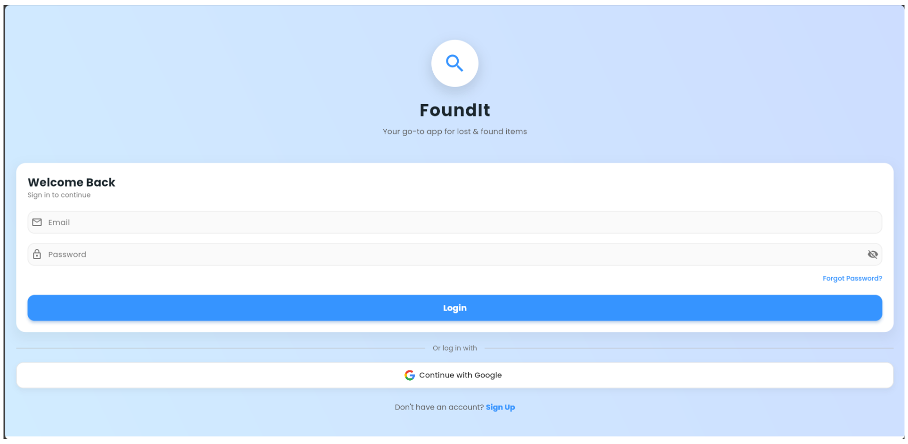


Figure 6.2: Screenshot of the User Login page.

6.2.3 Home Page

The home page serves as the primary entry point for users after login, offering quick navigation to key functionalities of the application.

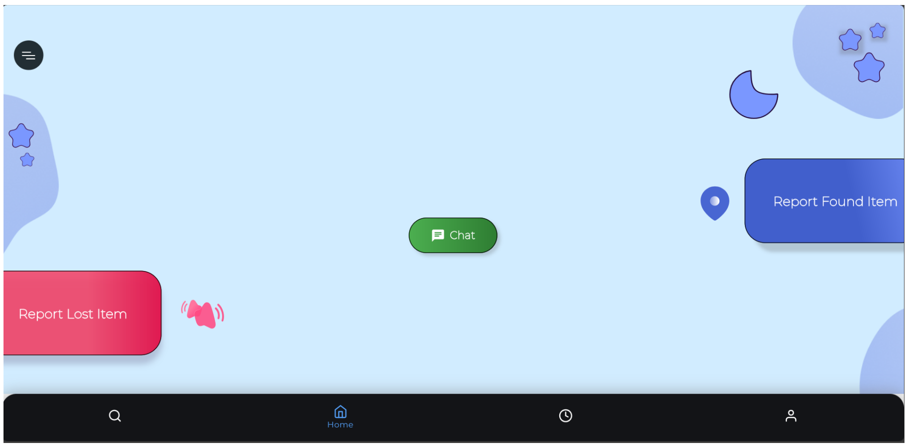


Figure 6.3: Screenshot of the Home Page.

6.2.4 Map

The interactive map module allows users to visualize the locations where found items have been reported. This feature is crucial for geographical context and helps users identify items in their vicinity.

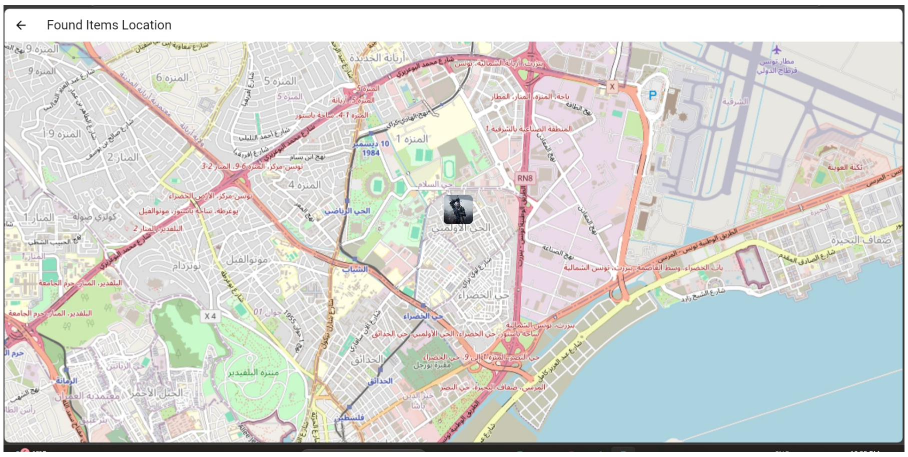
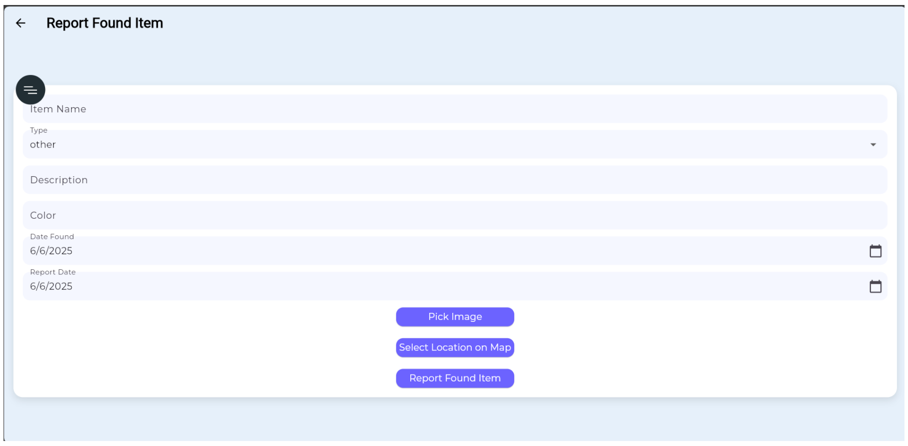


Figure 6.4: Screenshot of the Map feature displaying item locations.

6.2.5 Report Found Items

This module facilitates users in reporting items they have found. It captures details about the item and its location, pushing this data to the backend services for processing and matching.


Report Found Item

Item Name

Type
other

Description

Color

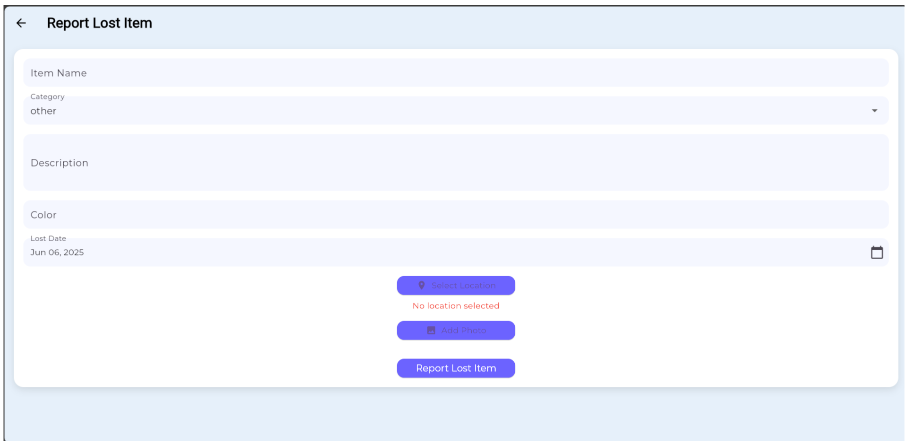
Date Found
6/6/2025

Report Date
6/6/2025

Figure 6.5: Screenshot of the Report Found Items form.

6.2.6 Report Lost Items

Conversely, the report lost items module enables users to submit details about items they have lost. This information is then used by the similarity search service to find potential matches with reported found items.


Report Lost Item

Item Name

Category
other

Description

Color

Lost Date
Jun 06, 2025

No location selected

Figure 6.6: Screenshot of the Report Lost Items form.

6.2.7 User Profile

The user profile section allows users to view and manage their personal information.

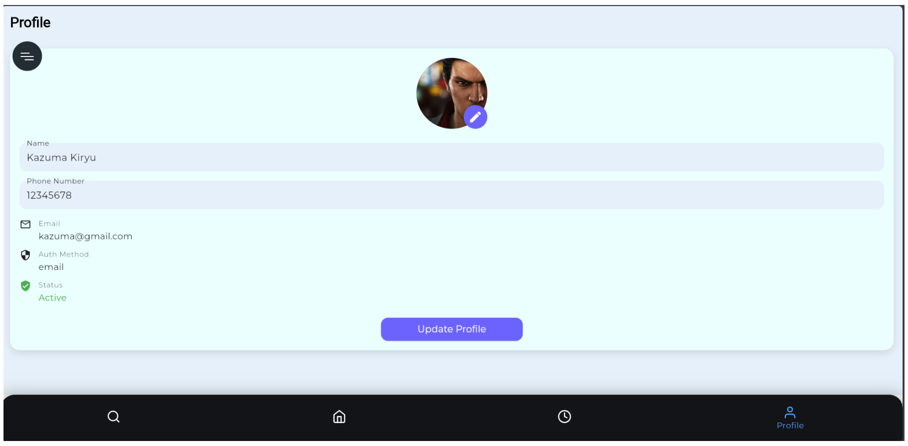


Figure 6.7: Screenshot of the User Profile page.

6.2.8 Notifications

This module is responsible for delivering real-time alerts and updates to users, such as potential matches for lost items and responses to reported items

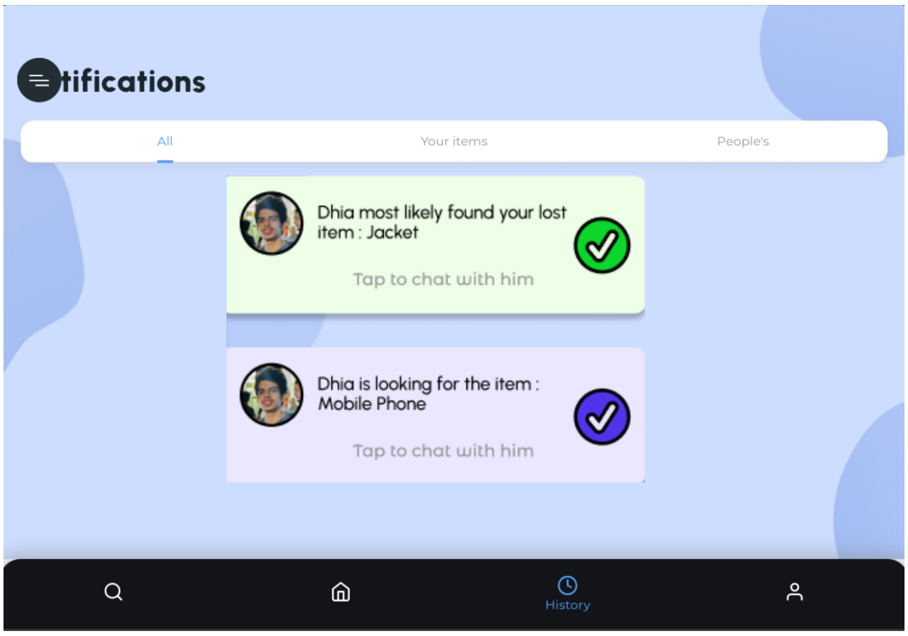

tifications
All
Your items
People's
Dhia most likely found your lost item : Jacket
Tap to chat with him
Dhia is looking for the item : Mobile Phone
Tap to chat with him
History

Figure 6.8: Screenshot of the Notifications page.

6.2.9 Chat

This module enables real-time communication between users, facilitating direct interaction for discussing found or lost items, arranging returns, or clarifying details.

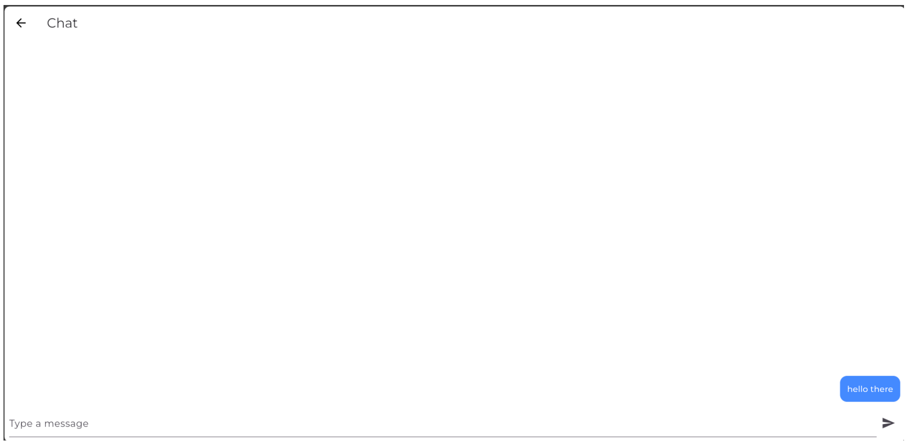


Figure 6.9: Screenshot of the Chat page.

6.2.10 Admin Dashboard

The administrative dashboard provides system administrators with tools to oversee platform operations.

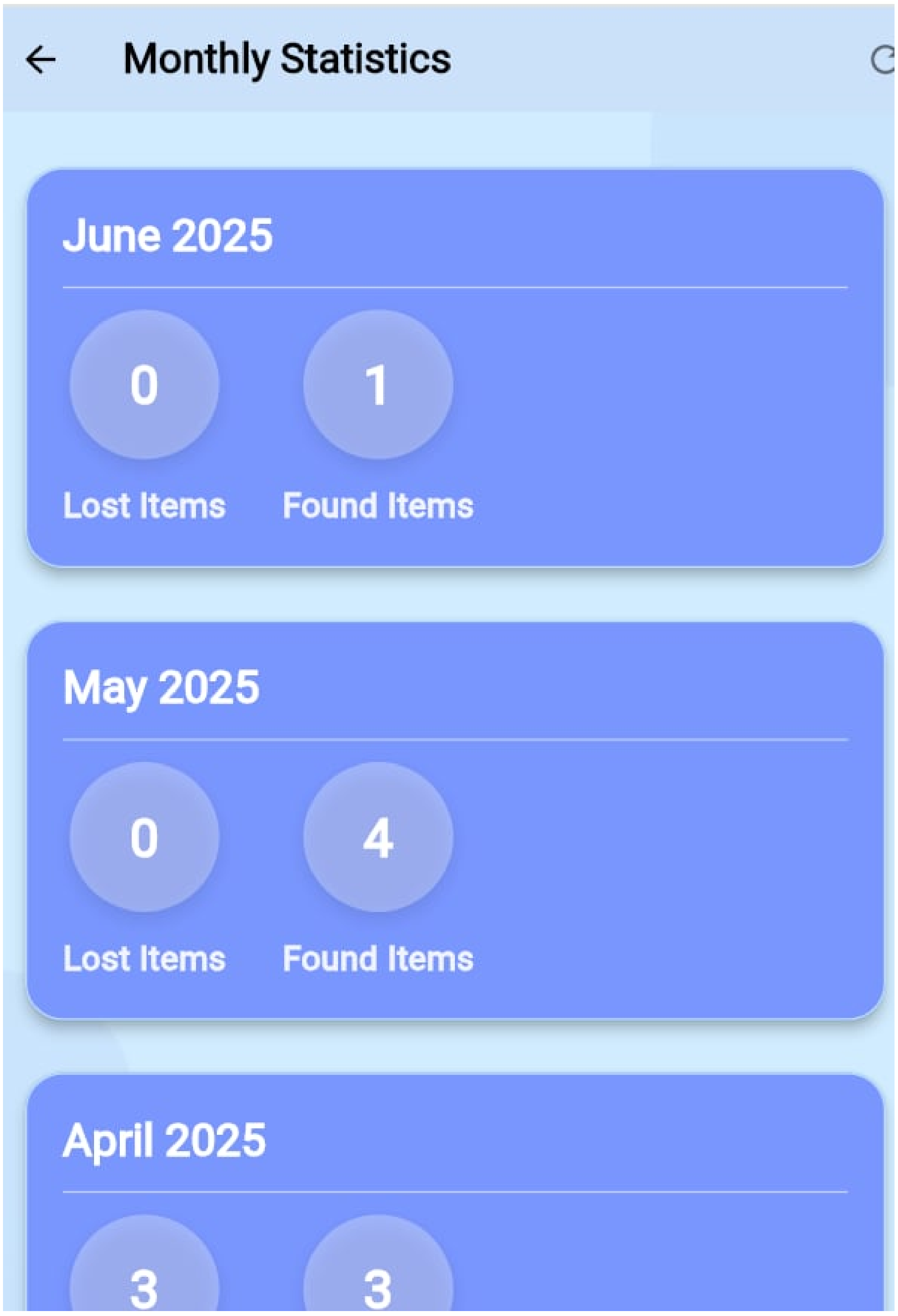
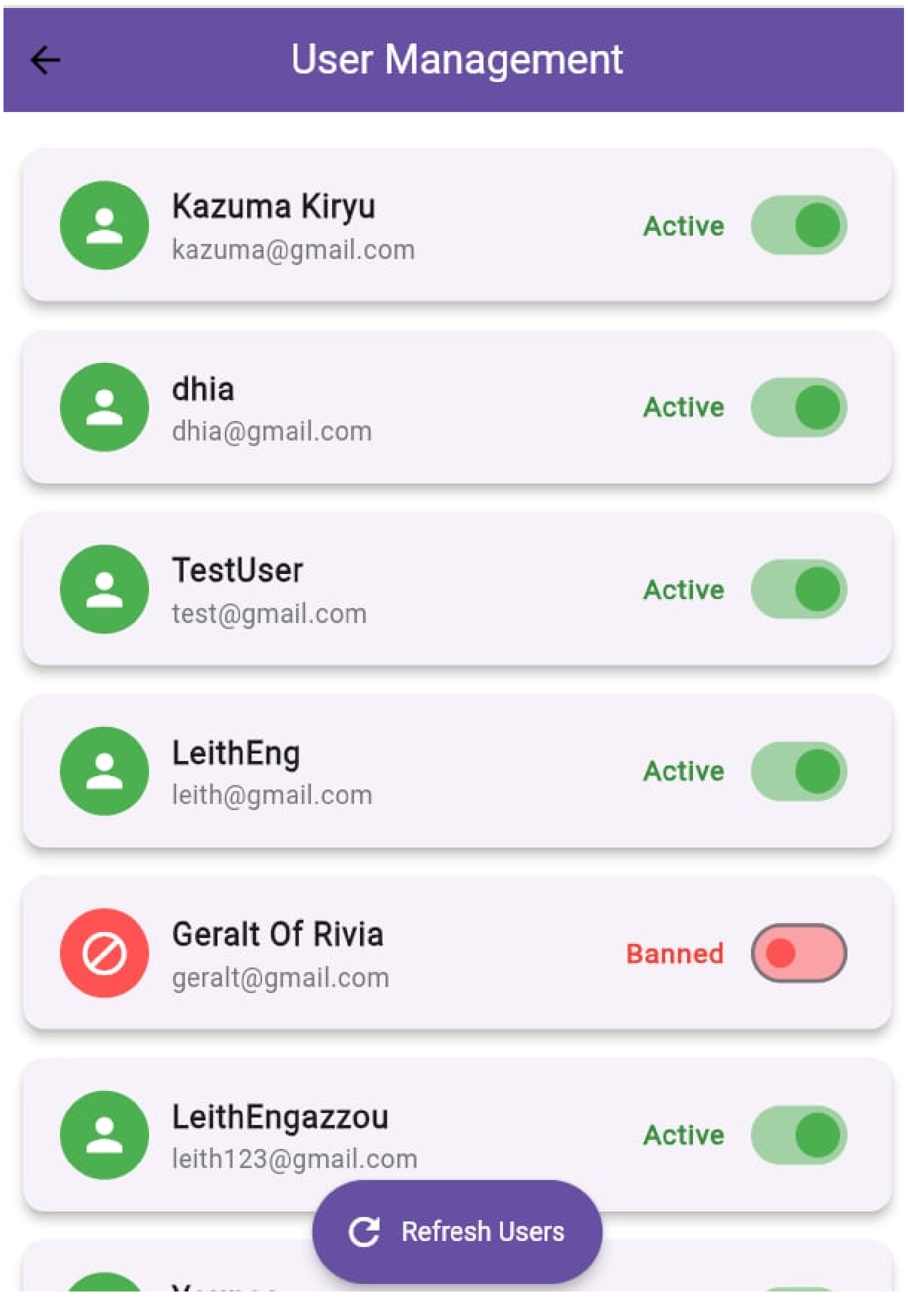


Figure 6.10: Screenshot of the Admin Dashboard.

6.2.11 User Management Page

This page, accessible by administrators, allows for the management of user accounts, including viewing user details, modifying permissions, or deactivating accounts, ensuring proper user governance.


User Management

Kazuma Kiryu
kazuma@gmail.com
Active

dhia
dhia@gmail.com
Active

TestUser
test@gmail.com
Active

LeithEng
leith@gmail.com
Active

Geralt Of Rivia
geralt@gmail.com
Banned

LeithEngazzou
leith123@gmail.com
Active

Figure 6.11: Screenshot of the User Management Page.

6.2.12 Backend Service Orchestration and Data Flow

Once a user submits a lost or found item via the frontend, the data is initially stored in Firebase. Beyond this point, a distributed microservices architecture is responsible for processing, inference, and storage. Figures3.1, 3.2, 6.12, and 6.13 illustrate the service interactions, messaging flow, and real-time data propagation.

Gateway Handling and Task Dispatching The *Gateway* service functions as the central orchestrator between the frontend and backend services. Upon receiving a new report:

- It uploads the associated image to Azure Blob Storage using the Unstructured Data Storage

Service.

- It broadcasts the task to RabbitMQ using a `fanout` exchange (`gateway_fanout_exchange`), directing the message to both `task_queue_similarity` and `task_queue_suspicious`.

Listener Services and AI Backends The messages are consumed by two specialized services:

- **Similarity Search Service**: This service listens to `task_queue_similarity`, uses CLIP embeddings for semantic representation, and queries Qdrant for similarity scoring. Additionally, Neo4j is used to extract contextually related items, improving result relevance.
- **Suspicious User Detection Service**: Connected to `task_queue_suspicious`, this service analyzes behavior patterns using Redis and anomaly detection models to flag potentially fraudulent actions.

Result Aggregation and Gateway Update Each service returns results to specific queues:

- Similarity results → `result_similarity_queue` via `result.similarity`.
- Suspicious activity results → `result_suspicious_queue` via `result.suspicious`.

The Gateway consumes these queues and updates Firebase records accordingly.

Illustrative Architecture and Implementation Snapshots

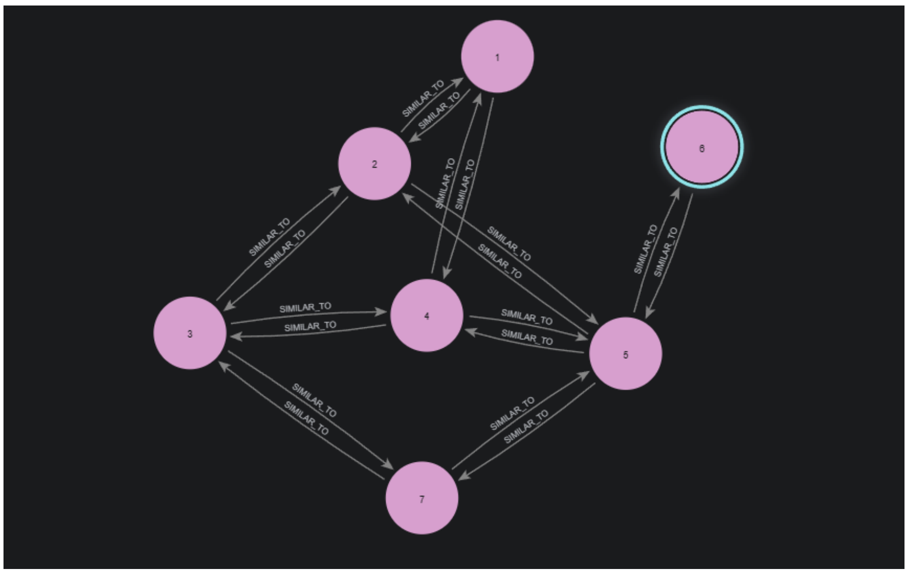
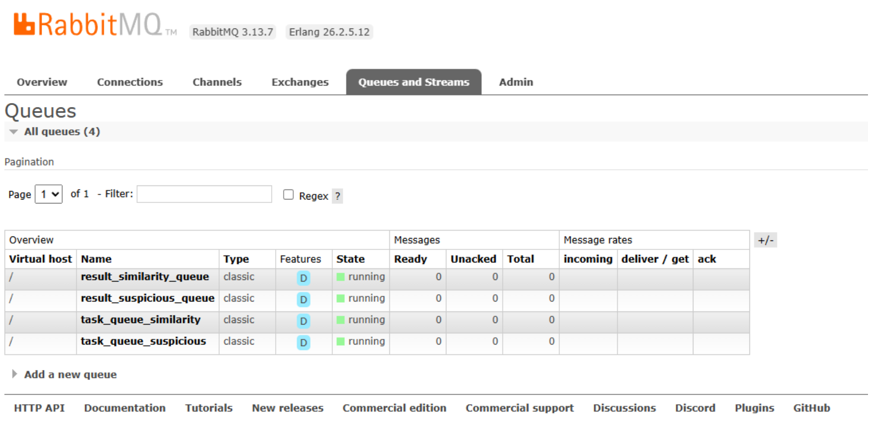


Figure 6.12: Neo4j Visualization of Related Posts



Overview					Messages			Message rates		
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/	result_similarity_queue	classic	D	🟩 running	0	0	0			
/	result_suspicious_queue	classic	D	🟩 running	0	0	0			
/	task_queue_similarity	classic	D	🟩 running	0	0	0			
/	task_queue_suspicious	classic	D	🟩 running	0	0	0			

Figure 6.13: Live RabbitMQ Queue Monitoring Interface

6.3 Testing and Debugging

A rigorous approach to testing and debugging was essential to ensure the robustness, reliability, and correctness of the implemented solution. This section outlines the methodologies and tools employed to identify and rectify defects throughout the development lifecycle within this microservices environment.

6.3.1 Testing Strategy

The testing strategy adopted for this project encompassed various levels of testing, ensuring comprehensive coverage of the system's functionalities and non-functional requirements, with particular attention to inter-service communication.

- **Unit Testing**:
  - **Purpose**: To verify the correctness of individual functions, classes, or small code units within each microservice in isolation.
  - **Methodology**: Each significant function or method within the Frontend, Gateway, Suspicious User Detection Backend, and Similarity Search Backend was tested independently, often using mock objects for external dependencies (e.g., database calls, message broker interactions).
- **Integration Testing**:
  - **Purpose**: To test the interactions between different modules or services, ensuring they work together seamlessly. This is crucial in a microservices architecture.
  - **Methodology**:
    - *Service-to-Database Integration*: Testing backend services' interaction with Redis, Qdrant, Neo4j, and Azure Storage.

    * *Message Broker Integration*: Testing that services correctly publish messages to and consume messages from RabbitMQ.
    * *API Integration*: Verifying that the Frontend correctly communicates with Firebase and the Gateway, and that the Gateway correctly routes requests and dispatches messages.
  - **Tools Used**: Postman for API testing, Docker Compose for spinning up dependent services during integration tests, specific client libraries for RabbitMQ/Firebase testing.

- **End-to-End (System) Testing**:
  - **Purpose**: To evaluate the complete and integrated software product from the user's perspective, verifying the entire flow across all microservices and external dependencies.
  - **Methodology**: Test cases covered major use cases (from Section 2.3) that involve multiple service interactions, from frontend action to final result, including complex flows like suspicious user detection and similarity search. Performed in an environment that closely mimics the production setup.
  - **Tools Used**: Cypress (for frontend E2E), Selenium, custom scripting using backend frameworks, load testing tools like JMeter for performance testing on the integrated system.

6.3.2 Debugging Process and Tools

Debugging was an iterative process throughout the development, aiming to identify and resolve defects efficiently across the distributed architecture.

- **Centralized Logging and Monitoring**:
  - **Methodology**: Comprehensive logging was implemented across all microservices (Frontend, Gateway, Suspicious User Detection Service, Similarity Search Service, Unstructured Data Storage Service) to record application events, errors, and warnings. Distributed tracing was also considered to follow requests across service boundaries.
- **Integrated Development Environment (IDE) Debuggers**:
  - **Methodology**: Utilized built-in debugger functionalities of IDEs to step through code, inspect variable states, and set breakpoints within individual microservices.
  - **Tools Used**: VS Code debugger. This was crucial for pinpointing issues during local development of each service.
- **Browser Developer Tools**:

  - **Methodology**: For frontend development, browser developer tools were extensively used to inspect HTML, CSS, JavaScript, network requests (especially calls to Firebase and the Gateway), and console errors.
  - **Tools Used**: Chrome DevTools, Firefox Developer Tools, Edge Developer Tools

- **Message Queue Inspection Tools**:
  - **Methodology**: Directly inspecting RabbitMQ queues and exchanges to ensure messages are being correctly published, routed, and consumed, and to diagnose issues with message flow.
  - **Tools Used**: RabbitMQ Management UI.

- **Version Control System (VCS) for Debugging**:
  - **Methodology**: Git was instrumental in tracking changes, enabling easy rollback to previous stable versions when a bug was introduced, and facilitating collaborative debugging through code reviews across different service repositories.
  - **Tools Used**: Git.

6.4 Conclusion

This chapter demonstrated the successful transition from theoretical design to a tangible, working application. We thoroughly explored the implementation of each key module, showcasing how the system translates into practical functionalities like user registration, login, the interactive map, and the reporting of lost and found items. The detailed screenshots provided a clear visual representation of the user interface and the seamless integration of various features.

Furthermore, we highlighted the rigorous testing and debugging methodologies employed throughout the development lifecycle. Our comprehensive approach, encompassing unit, integration, and end-to-end testing, ensured the robustness, reliability, and correctness of the implemented system. The systematic debugging processes, aided by centralized logging, IDE debuggers, browser developer tools, and message queue inspection, were crucial in identifying and resolving issues efficiently.

The successful implementation of these modules and the meticulous testing processes collectively validate the practical feasibility of our proposed solution, laying a solid foundation for the system's deployment and future enhancements.

Conclusion

We embarked on developing a Lost and Found Mobile App, a journey structured meticulously using the Waterfall methodology, which offered both predictable challenges and satisfying achievements. Our primary objective was to create an "intelligent" solution for locating misplaced items by integrating advanced computer vision technology. The app's core functionality harnesses AI for image recognition, enabling it to identify lost belongings from user-submitted photos, contingent on the quality of the image.

Our adherence to the Waterfall model meant a linear, sequential progression through distinct phases: meticulous requirements gathering, detailed design, implementation, thorough testing, and eventual deployment. This disciplined approach provided a clear roadmap and ensured comprehensive documentation at every step, which was invaluable for team alignment and managing expectations. We also prioritized user-friendliness, ensuring the app's theoretical compatibility across various devices and its ability to provide real-time updates for that thrilling moment a lost item might be located.

In essence, we built an app to automate the process of finding lost possessions. This project not only allowed us to gain significant skills in cutting-edge technologies but also reinforced the benefits of a structured development lifecycle. While some might perceive the integration of sophisticated AI for what seems like a simple problem as over-engineering, our hope is that it genuinely reduces the universal stress of losing things. After all, if our advanced AI can't locate your keys, at least we rigorously documented every attempt! Looking ahead, we envision enhancing the app with features such as crowd-sourced surveillance integration, allowing opted-in users to anonymously share camera feeds for real-time scanning of public spaces, and automated police report integration, streamlining the reporting process for high-value lost items by pre-filling details from the app.

Bibliography

[1] StartInfinity, *Project management methodologies: waterfall*, `https://startinfinity.com/project-management-methodologies/waterfall`.

[2] A. Radford, J. W. Kim, C. Hallacy, *et al.*, *Learning transferable visual models from natural language supervision*, `https://openai.com/index/clip/`, Accessed: 2025-06-08, 2021.

[3] G. Inc., *Groq: ultra low latency lpu inference for large language models*, `https://groq.com`, Accessed: 2025-06-08, 2024.

[4] H. Touvron, T. Lavril, G. Izacard, *et al.*, *Llama: open and efficient foundation language models*, `https://ai.meta.com/llama`, Accessed: 2025-06-08, 2023.

[5] Ollama Team, *Ollama: high-performance ai inference backend*, `https://ollama.com/`, Accessed: 2025-06-08, 2024.