Web-Based Lost and Found Management System for Campus Environment with Auto-Matching and AI Chatbot Using Go Backend and React Frontend

1st Edward Wibisono Yulianto
Department of Informatics
Widya Mandala Kalijudan University
Surabaya, Indonesia
edward-w.inf24@ukwms.ac.id

2nd Bambang Herlambang
Department of Informatics
Widya Mandala Kalijudan University
Surabaya, Indonesia
bambang-h.inf24@ukwms.ac.id

3rd Nathanael Melvin
Department of Informatics
Widya Mandala Kalijudan University
Surabaya, Indonesia
nathanael-m.inf24@ukwms.ac.id

***Abstract*—The Lost and Found System is a client-server web-based application designed to manage lost and found items in a campus environment. The system is built using RESTful API architecture with a Go (Golang)-based backend and React frontend. MySQL database is used for data storage with transaction support to ensure data consistency. The system implements several key features: management of found items and lost item reports, a multi-stage verification claim system, an automatic matching algorithm using string similarity (Levenshtein Distance) to connect lost items with found items, and AI chatbot integration using Groq API to assist users in searching and reporting. The role-based access control (RBAC) architecture enables three access levels: user, manager, and admin, each with different access rights. To improve operational efficiency, the system is equipped with background workers that perform automatic tasks such as archiving expired items, automatic matching between lost and found items, and sending real-time notifications to users. The system also provides audit logging features for tracking user activities and data export in Excel and PDF formats for reporting purposes. Implementation using repository pattern and service layer ensures separation of concerns and facilitates maintenance. Middleware for JWT authentication, rate limiting, and CORS protection ensures system security. With graceful shutdown and context timeout approaches, the system can handle high loads stably. Test results show that the system can effectively manage the claim process and improve the success rate of returning lost items to their owners.**

***Index Terms*—Lost and Found System, RESTful API, String Similarity Algorithm, Role-Based Access Control, Background Workers, AI Chatbot Integration, Microservices Architecture**

I. Introduction

A. Background

The loss of personal belongings is a common problem that frequently occurs in high-mobility campus environments such as universities, schools, and other educational institutions. Students and academic community members often lose important items such as wallets, keys, electronic devices, academic documents, and other personal belongings in various campus locations such as classrooms, libraries, cafeterias, and other public areas. On the other hand, many found items cannot be returned to their owners due to limitations in effective management systems.

Conventional lost and found management systems that still rely on manual recording, announcements through information boards, or social media groups have several significant limitations. The item search process is time-consuming, information is not well-organized, there is no clear verification mechanism for the claim process, and data trails are often lost, preventing items from being returned to their owners. Additionally, there is no adequate tracking system to monitor item status from reporting to return.

The development of information technology and web-based systems provides solutions to overcome these problems. A web-based Lost and Found system can provide a centralized platform to manage the entire process of reporting lost items, registering found items, claim verification process, and returning items to their owners. By utilizing string similarity algorithms and artificial intelligence technology, the system can automatically match lost items with found items based on reported characteristics, thereby accelerating the identification process and increasing the likelihood of items being returned to their owners.

This system is built using modern architecture with a Go (Golang)-based backend known for its high performance and excellent concurrent processing, and a React-based frontend for a responsive and interactive user interface. Implementation of role-based access control (RBAC) ensures that each user has access rights appropriate to their role, while background workers perform automatic tasks such as item matching and data archiving. AI chatbot integration using Groq API provides users with convenience in conducting searches and obtaining interactive assistance.

B. Problem Formulation

Based on the background described above, this research will address several problem formulations as follows:

1) How to design an effective information system to manage lost and found items in a campus environment?

Identify applicable funding agency here. If none, delete this.

2) How to implement a string similarity algorithm to automatically match lost items with found items?
3) How to design a secure claim verification system to ensure items are returned to legitimate owners?
4) How to integrate an AI chatbot to assist users in the item search and reporting process?
5) How to implement background workers to perform automatic tasks such as matching and archiving?
6) How to design a system with scalable and maintainable architecture using good software engineering patterns?

C. Research Objectives

This research has the following objectives:

1) To design and implement a web-based Lost and Found information system with RESTful API architecture using Go and React.
2) To implement the Levenshtein Distance algorithm to calculate similarity scores between lost and found items, enabling effective auto-matching.
3) To build a multi-stage claim verification system involving users, managers, and admins with a structured approval mechanism.
4) To integrate a Groq API-based AI chatbot to provide interactive assistance to users in item search and reporting processes.
5) To implement background workers using goroutines to perform automatic tasks such as auto-matching, auto-archiving, and notification delivery.
6) To apply software engineering best practices such as repository pattern, service layer, middleware architecture, and dependency injection to ensure maintainable and testable code.

D. Problem Scope

To maintain research focus and ensure optimal results, this research has the following limitations:

1) The system is specifically designed for campus environments with three user levels: user (students/staff), manager (administrator), and admin (system administrator).
2) The matching algorithm used is string similarity based on Levenshtein Distance with configurable threshold.
3) Claim verification uses secret details that are only known by the item owner and verified by managers or admins.
4) The AI chatbot uses Groq API with the LLaMA 3.3 70B Versatile model for natural language processing.
5) The system only manages data in text and image formats, excluding videos or complex documents.
6) Background workers run periodically with predetermined intervals (matching: 30 minutes, expiration check: 1 hour).
7) The system does not include integration with payment systems or rewards for item finders.
8) System notifications are only through in-app notifications, excluding email or SMS notifications.

E. Research Benefits

This research is expected to provide the following benefits:

1) Theoretical Benefits:

1) To contribute to the development of web-based management information systems with a Lost and Found system case study.
2) To demonstrate practical implementation of string similarity algorithms (Levenshtein Distance) in real-world application contexts.
3) To provide a reference for implementing microservices architecture and background workers using Go (Golang).
4) To demonstrate the application of AI integration in information systems to enhance user experience.
5) To provide a case study of role-based access control (RBAC) implementation in multi-user web applications.

2) Practical Benefits:

1) For educational institutions: Providing an effective system to manage lost and found items, improving services to students and the academic community.
2) For users: Facilitating the process of reporting lost items and searching for found items with a user-friendly interface and AI chatbot assistance.
3) For administrators: Providing tools to verify claims, manage item data, and generate reports for audit purposes.
4) For developers: Providing an implementation reference for systems with clean, scalable architecture that follows best practices.
5) For the community: Increasing the likelihood that lost items can be returned to their owners through a well-organized system.

II. Literature Review

This chapter discusses the theoretical foundation and technologies used in developing the Lost and Found System, including RESTful API architecture, design patterns, string similarity algorithms, security mechanisms, and artificial intelligence integration.

A. RESTful API Architecture

REST (Representational State Transfer) is an architectural style for designing networked applications that uses HTTP methods to perform CRUD operations. The Lost and Found System implements RESTful principles through structured endpoints that map to specific resources.

The system uses standard HTTP methods: GET for retrieving data, POST for creating new resources, PUT/PATCH for updates, and DELETE for removal operations. Each endpoint follows a clear naming convention, such as `/api/items` for item management and `/api/claims` for claim processing. The API returns standardized JSON responses with consistent structure including status codes, messages, and data payloads.

According to Fielding's REST constraints, the system implements stateless communication where each request contains all necessary information for processing. The server maintains no client context between requests, with authentication handled through JWT tokens passed in request headers. This state-

less design enables horizontal scaling and improves system reliability.

B. Design Patterns

The system implements several software engineering patterns to ensure maintainability, testability, and separation of concerns.

1) Repository Pattern: The Repository Pattern provides an abstraction layer between the business logic and data access layer. Each entity (Item, LostItem, Claim, User) has a dedicated repository that encapsulates all database operations. For example, `ItemRepository` handles all database queries related to items, including CRUD operations, complex queries with filters, and transaction management.

This pattern offers several advantages: it centralizes data access logic, makes the codebase more testable by allowing repository mocking, and provides a consistent interface for data operations. The repository layer uses GORM as the ORM framework, which provides type-safe database operations and automatic query generation.

2) Service Layer Pattern: The Service Layer Pattern encapsulates business logic separate from controllers and repositories. Services like `ClaimService`, `ItemService`, and `MatchService` contain the core business rules and orchestrate multiple repository operations when needed.

For instance, the claim verification process in `ClaimService` involves multiple steps: validating the claim, calculating similarity scores, updating item status, creating notifications, and logging audit trails. By centralizing this logic in a service, the system ensures consistency across different entry points and simplifies testing of business rules.

3) Dependency Injection: The system uses dependency injection to manage component dependencies. Controllers receive repository and service dependencies through constructor injection, making components loosely coupled and easily testable. This approach follows SOLID principles, particularly the Dependency Inversion Principle, where high-level modules depend on abstractions rather than concrete implementations.

C. String Similarity Algorithm

The automatic matching feature uses the Levenshtein Distance algorithm to calculate similarity between text strings. This algorithm measures the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into another.

1) Levenshtein Distance Implementation: The implementation uses dynamic programming to compute the edit distance efficiently. Given two strings s_1 and s_2, the algorithm creates a matrix where $dp[i][j]$ represents the minimum edit distance between the first i characters of s_1 and the first j characters of s_2.

The recurrence relation is:

$$dp[i][j] = \min \begin{cases} dp[i-1][j] + 1 & \text{(deletion)} \\ dp[i][j-1] + 1 & \text{(insertion)} \\ dp[i-1][j-1] + cost & \text{(substitution)} \end{cases} \quad (1)$$

where $cost = 0$ if $s_1[i] = s_2[j]$, otherwise $cost = 1$.

The similarity score is then calculated as:

$$similarity = 1 - \frac{distance}{\max(len(s_1), len(s_2))} \quad (2)$$

This produces a normalized score between 0 and 1, where 1 indicates identical strings and 0 indicates completely different strings.

2) Text Normalization: Before calculating similarity, the system applies text normalization: converting to lowercase, removing special characters, and filtering stopwords. This preprocessing improves matching accuracy by focusing on meaningful content words rather than common function words.

The matching algorithm considers multiple fields with weighted scores. Name similarity receives 50% weight while description similarity receives 50% weight. A match is considered significant when the combined score exceeds the configured threshold (typically 50%).

D. Role-Based Access Control (RBAC)

The system implements RBAC to manage user permissions across three roles: User, Manager, and Admin. Each role has specific permissions defined in the database, allowing fine-grained access control.

1) Permission System: Permissions are defined as action-resource pairs (e.g., `item:create`, `claim:approve`). The `Role` model contains a many-to-many relationship with `Permission`, allowing flexible assignment of permissions to roles. Middleware functions check user permissions before allowing access to protected endpoints.

For example, only users with `claim:approve` permission (Managers and Admins) can verify claims. Regular users can create claims but cannot approve them. This separation ensures proper workflow enforcement and maintains data integrity.

2) Hierarchical Access: The system implements a hierarchical access model where higher-level roles inherit permissions from lower levels. Admins can perform all Manager operations, and Managers can perform all User operations. This simplifies permission management while maintaining security boundaries.

E. Authentication and Security

1) JSON Web Tokens (JWT): The system uses JWT for stateless authentication. Upon successful login, the server generates a token containing user ID, email, and role information. This token is cryptographically signed using HMAC-SHA256 with a secret key.

Each subsequent request includes the JWT in the Authorization header as a Bearer token. The JWT middleware validates the token signature, checks expiration, and loads user information for authorization decisions. Tokens expire after 7 days, requiring users to re-authenticate periodically.

2) Password Security: User passwords are hashed using bcrypt, an adaptive hash function designed for password storage. Bcrypt incorporates a salt to prevent rainbow table attacks and uses a configurable work factor to remain resistant to brute-force attacks as computing power increases.

3) Data Encryption: Sensitive personal information (NRP, phone numbers) is encrypted using AES-256 in GCM mode before storage. The encryption key is stored securely as an environment variable. This ensures that even if the database is compromised, sensitive data remains protected.

4) Rate Limiting: The system implements rate limiting to prevent abuse and DoS attacks. Each IP address is limited to 1000 requests per minute. The rate limiter uses an in-memory map to track request counts per IP address, with automatic cleanup of stale entries.

F. Background Workers and Concurrency

1) Goroutines for Concurrent Processing: The system uses Go's goroutines for concurrent background tasks. Four main workers run continuously: `ExpireWorker` for archiving expired items, `MatchingWorker` for automatic item matching, `NotificationWorker` for sending notifications, and `AuditWorker` for log aggregation.

Goroutines are lightweight threads managed by the Go runtime, enabling efficient concurrent execution without the overhead of OS threads. Each worker runs in its own goroutine, allowing parallel processing of different tasks.

2) Worker Pool Pattern: The `ExpireWorker` implements a worker pool pattern with 5 concurrent workers processing expired items. This pattern provides controlled concurrency, preventing resource exhaustion while maximizing throughput. A task queue (buffered channel) holds items to be processed, and workers consume tasks concurrently.

3) Graceful Shutdown: The system implements graceful shutdown using WaitGroups and stop channels. When a shutdown signal is received, workers complete their current tasks before terminating. The HTTP server stops accepting new connections but completes in-flight requests. This prevents data loss and ensures clean system termination.

G. Database Design and Transactions

1) Relational Database Schema: The system uses MySQL with a normalized relational schema. Key tables include `users`, `items`, `lost_items`, `claims`, `match_results`, and `notifications`. Foreign key constraints maintain referential integrity, and indexes optimize query performance.

The schema uses soft deletes (deleted_at timestamp) to preserve data history. This allows recovery of accidentally deleted records and maintains audit trails for compliance purposes.

2) Transaction Management: Database transactions ensure ACID properties for complex operations. For example, the claim verification process wraps multiple operations in a transaction: updating claim status, modifying item status, creating notifications, and logging audit entries. If any step fails, the entire transaction rolls back, maintaining data consistency.

The system uses GORM's transaction API with proper error handling. Row-level locking prevents concurrent modification conflicts in critical sections, such as claim approval where multiple managers might process the same claim simultaneously.

3) Stored Procedures: The system leverages MySQL stored procedures for complex operations like automatic archiving. The `sp_archive_expired_items` procedure efficiently identifies and archives expired items in a single database round-trip, reducing network overhead and improving performance.

H. Artificial Intelligence Integration

1) Groq API and LLaMA Model: The system integrates an AI chatbot using the Groq API with the LLaMA 3.3 70B Versatile model. Groq provides high-performance inference for large language models, enabling real-time conversational interactions.

The chatbot assists users in item searches, report guidance, and claim process explanation. It receives context about the user's lost items and recent found items, providing personalized responses based on system state.

2) Intent Recognition: The system implements basic intent recognition by analyzing keywords in user messages. Four main intents are detected: `search_item`, `report_lost`, `claim_help`, and `general`. The detected intent guides response generation, providing relevant information and actions.

3) Context Management: Each chat request includes conversation history and relevant system context. The system builds context by querying the user's lost item reports and searching for relevant found items. This context is included in the AI prompt, enabling informed responses that reference specific items and match results.

I. API Request Lifecycle

The complete lifecycle of an API request demonstrates the integration of all architectural components:

1) Request arrives at the Gin router and passes through middleware layers
2) CORS middleware handles cross-origin requests
3) Rate limiter checks request quota for the client IP
4) JWT middleware validates authentication token and loads user data
5) Role middleware verifies user has required permissions
6) Request reaches the appropriate controller
7) Controller delegates business logic to service layer
8) Service coordinates multiple repositories for data operations
9) Repositories execute database queries using GORM
10) Response flows back through the layers with standardized format
11) Background workers process asynchronous tasks (notifications, matching)

This layered architecture provides separation of concerns, making the system maintainable, testable, and scalable. Each layer has a specific responsibility and communicates through well-defined interfaces.

J. System Reliability and Error Handling

1) Context Timeouts: All database operations use Go's context package with timeouts (typically 3-15 seconds depending

on complexity). This prevents hung requests from blocking resources indefinitely. If an operation exceeds its timeout, it returns an error that can be handled gracefully.

2) Transaction Rollback: The system implements comprehensive transaction error handling. When any operation within a transaction fails, the entire transaction rolls back automatically. This prevents partial updates that could leave the database in an inconsistent state.

3) Structured Error Responses: All API errors return structured JSON responses with consistent format: success status, error message, and optional error details. This standardization simplifies client-side error handling and debugging.

The system distinguishes between client errors (4xx status codes) for invalid requests and server errors (5xx status codes) for internal failures, following HTTP best practices.

K. Audit Logging and Monitoring

The system maintains comprehensive audit logs tracking all significant actions. Each log entry records the user, action type, affected entity, timestamp, IP address, and user agent. This provides accountability and enables security analysis.

Revision logs track changes to item data, recording the field changed, old value, new value, and reason for change. This audit trail supports compliance requirements and enables data recovery if needed.

The logging strategy balances detail with performance, using asynchronous logging to avoid blocking request processing while ensuring important events are captured.

III. System Design and Implementation

This chapter presents the detailed design and implementation of the Lost and Found System, including the database schema, system architecture, component design, and implementation strategies. The system employs a microservices-oriented architecture with RESTful API design principles, background workers for automated tasks, and a comprehensive security framework.

A. Database Design

The database design forms the foundation of the Lost and Found System, implementing a normalized relational schema that ensures data integrity, supports complex queries, and maintains audit trails for compliance purposes.

1) Entity Relationship Diagram: The system database consists of 15 interconnected tables organized into functional domains: user management, item management, claim processing, matching system, and audit logging. Figure 1 illustrates the complete entity relationship diagram showing all entities, their attributes, and relationships.

The database schema implements several key design patterns:

Soft Delete Pattern: All primary tables include a `deleted_at` timestamp field, enabling logical deletion rather than physical removal of records. This preserves data history and supports recovery of accidentally deleted items while maintaining referential integrity.

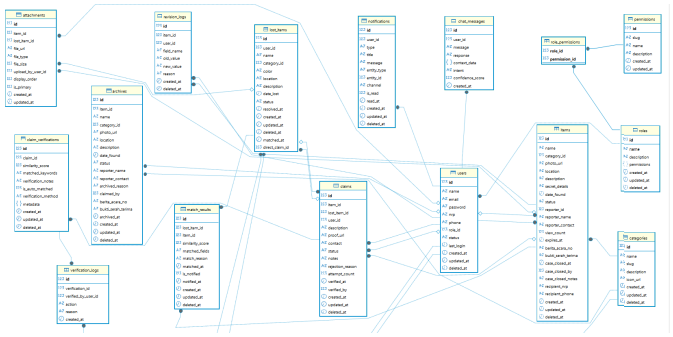

Fig. 1. Entity Relationship Diagram of Lost and Found System Database

Audit Trail Design: The `audit_logs` and `revision_logs` tables maintain comprehensive records of all system activities. The audit log captures high-level actions (create, update, delete, verify) with associated metadata including IP addresses and user agents. The revision log tracks field-level changes to items, storing old values, new values, and reasons for modification.

Polymorphic Relationships: The `claims` table implements polymorphic associations, allowing claims to reference either found items (regular claims) or lost item reports (direct claims). This design enables two distinct claim workflows within a unified data structure.

2) Core Database Tables: **Users and Roles:** The user management system implements role-based access control (RBAC) through the `users`, `roles`, `permissions`, and `role_permissions` tables. Users are assigned to roles (admin, manager, user), and roles contain collections of permissions that define allowed actions. This flexible design enables fine-grained access control without hardcoding permissions in application logic.

The `users` table stores encrypted sensitive information including NRP (student identification numbers) and phone numbers using AES-256-GCM encryption. The encryption key is stored securely as an environment variable and initialized during system startup.

Items and Lost Items: Found items are stored in the `items` table with comprehensive metadata including discovery location, date found, public description, and secret details. The `secret_details` field contains confidential information known only to the true owner, used for claim verification. The `expires_at` field automatically calculates to 90 days from the date found, after which items are eligible for archiving.

Lost item reports in the `lost_items` table capture characteristics of missing items to enable automatic matching. The table includes fields for color, expected location, and detailed descriptions that feed into the similarity algorithm. The `direct_claim_id` foreign key links to claims when finders directly contact owners.

Claims and Verification: The `claims` table manages the claim submission and verification process. Each claim references either an item (regular claim) or a lost item

(direct claim) through nullable foreign keys `item_id` and `lost_item_id`. Claims progress through statuses: pending, approved, rejected, waiting_owner, or verified.

The `claim_verifications` table stores similarity scores and matched keywords generated by the Levenshtein Distance algorithm. The `similarity_score` field (0-100) quantifies the match quality between claim descriptions and secret details. The `is_auto_matched` boolean indicates whether the verification was system-generated or manual.

Match Results: Automatic matching results are stored in `match_results`, linking lost items to found items with similarity scores above the configured threshold (typically 50%). The `matched_fields` JSON field contains detailed breakdown of which attributes matched (name, description, color) and their individual scores. The `is_notified` flag tracks whether users have been informed of potential matches.

Archives: The `archives` table preserves historical records of items removed from active inventory. Items are archived when they expire (90 days unclaimed) or when cases are closed (successfully returned to owner). The archive maintains a complete snapshot of the item's final state including the `berita_acara_no` (official handover document number) and `bukti_serah_terima` (proof of delivery) for closed cases.

Notifications: The `notifications` table implements an in-app notification system. Notifications are created for events including match discoveries, claim status changes, and case closures. The `entity_type` and `entity_id` fields provide polymorphic links to related records, enabling navigation to relevant items from notifications.

Chat Messages: The AI chatbot integration stores conversation history in `chat_messages`. Each message records the user's query, the AI-generated response, detected intent (search_item, report_lost, claim_help, general), and contextual data used for response generation. The `confidence_score` field quantifies the intent detection certainty.

3) Database Indexes and Performance Optimization: The schema includes strategic indexes to optimize query performance:

- **Primary Keys:** All tables use auto-incrementing integer primary keys for efficient joins and foreign key references.
- **Foreign Key Indexes:** Indexes on all foreign key columns (`user_id`, `item_id`, `category_id`, etc.) accelerate join operations and referential integrity checks.
- **Status Indexes:** Composite indexes on status fields (`items.status`, `claims.status`, `lost_items.status`) enable fast filtering of active records.
- **Timestamp Indexes:** Indexes on `created_at`, `expires_at`, and `deleted_at` support temporal queries and soft delete filtering.
- **Unique Indexes:** Unique constraints on `users.email`, `categories.slug`, and `archives.item_id` prevent duplicate entries.

4) Database Constraints and Referential Integrity: Foreign key constraints maintain referential integrity with appropriate cascading behaviors:

- `ON DELETE CASCADE`: Applied to dependent records that should be removed when parent is deleted (claims when items deleted, notifications when users deleted).
- `ON DELETE SET NULL`: Applied to optional references that should be preserved (verified_by when manager deleted, claimed_by when user deleted).
- `ON DELETE RESTRICT`: Applied to critical references that prevent deletion (categories referenced by items, roles referenced by users).

B. System Architecture

The Lost and Found System implements a layered architecture that separates concerns and promotes maintainability, testability, and scalability. Figure 2 illustrates the complete system architecture.

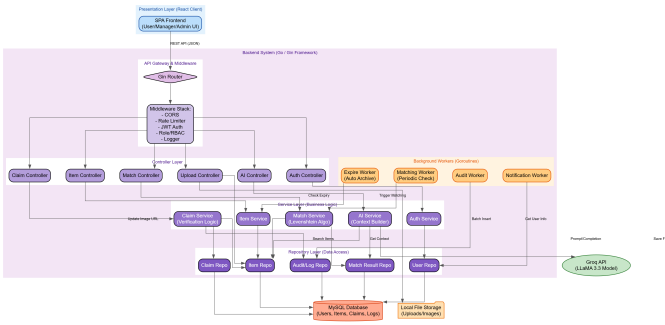

Fig. 2. Lost and Found System Architecture

1) Architectural Layers: **Presentation Layer:** The presentation layer consists of static HTML, CSS, and JavaScript files served by the Gin web framework. The frontend implements a single-page application (SPA) pattern with role-specific interfaces:

- `index.html`: Public landing page for browsing found items
- `user.html`: User dashboard for reporting lost items and managing claims
- `manager.html`: Manager interface for claim verification and item management
- `admin.html`: Administrator panel for user management, categories, and system configuration

The JavaScript frontend is organized into modular components (`ItemCard.js`, `ClaimCard.js`, `Modal.js`, etc.) that communicate with the backend via RESTful API calls. The `api.js` utility module handles HTTP requests, authentication token injection, and error handling.

API Layer: The RESTful API layer exposes HTTP endpoints for all system operations. The `routes.go` module defines endpoint mappings and associates them with controller functions. API routes are grouped by domain:

- `/api/auth/*`: Authentication endpoints (register, login, refresh token)

- `/api/items/*`: Found item management
- `/api/lost-items/*`: Lost item reports
- `/api/claims/*`: Claim submission and verification
- `/api/matches/*`: Match result queries
- `/api/admin/*`: Administrative operations
- `/api/ai/*`: AI chatbot interactions

Middleware Layer: HTTP requests pass through a pipeline of middleware functions before reaching controllers:

1) `CORSMiddleware`: Handles cross-origin resource sharing headers
2) `LoggerMiddleware`: Records request details for monitoring
3) `RateLimiterMiddleware`: Prevents abuse with per-IP rate limiting (1000 requests/minute)
4) `JWTMiddleware`: Validates authentication tokens and loads user context
5) `RoleMiddleware`: Enforces permission-based access control
6) `IdempotencyMiddleware`: Prevents duplicate submissions for sensitive operations

Controller Layer: Controllers handle HTTP request/response processing and input validation. Each controller focuses on a specific domain:

- `AuthController`: User registration, login, token refresh
- `ItemController`: CRUD operations for found items
- `LostItemController`: Lost item report management
- `ClaimController`: Claim submission and verification workflow
- `MatchController`: Similarity search and match retrieval
- `AdminController`: System administration functions
- `AIController`: Chatbot message processing

Controllers validate input using the Gin binding framework, extract user context from middleware, invoke appropriate service methods, and format responses using utility functions.

Service Layer: The service layer implements business logic and orchestrates complex operations. Services coordinate multiple repositories and handle transaction management:

- `AuthService`: Password hashing, token generation, user validation
- `ItemService`: Item lifecycle management, expiration handling
- `ClaimService`: Multi-stage claim verification, case closure
- `MatchService`: Similarity calculation, automatic matching
- `AIService`: Groq API integration, intent detection, context building

Services encapsulate business rules, ensuring consistent behavior across different entry points. For example, the claim verification process in `ClaimService` involves:

1) Locking the claim record with pessimistic locking
2) Calculating similarity score between claim and item
3) Creating or updating verification record
4) Updating claim and item status
5) Resolving related lost item reports
6) Creating user notifications
7) Logging audit entries

Repository Layer: Repositories provide an abstraction over database operations, encapsulating GORM queries and transaction management:

- `UserRepository`: User CRUD, authentication queries
- `ItemRepository`: Item queries with filtering, search, pagination
- `ClaimRepository`: Claim queries with complex joins
- `MatchResultRepository`: Match persistence and retrieval
- `NotificationRepository`: Notification creation and marking read

The repository pattern enables testing with mock implementations and provides a consistent interface for data access. All database operations use GORM's context-aware methods with timeout handling to prevent hung requests.

Worker Layer: Background workers run as concurrent goroutines, performing scheduled and periodic tasks:

- `ExpireWorker`: Archives items that have exceeded 90-day retention period
- `MatchingWorker`: Runs automatic matching algorithm every 30 minutes
- `NotificationWorker`: Sends pending notifications every 5 minutes
- `AuditWorker`: Aggregates and processes audit log entries

Workers implement graceful shutdown through stop channels and WaitGroups, ensuring in-progress tasks complete before system termination.

C. Request Processing Flow

Figure 3 illustrates the complete lifecycle of an API request through the system architecture.

A typical authenticated request follows this path:

1) HTTP request arrives at Gin router
2) CORS middleware adds cross-origin headers
3) Rate limiter checks request quota for client IP
4) Logger middleware records request details
5) JWT middleware validates token, loads user from database
6) Role middleware verifies user has required permissions
7) Router dispatches request to appropriate controller
8) Controller validates input and extracts parameters
9) Controller invokes service layer method with user context
10) Service begins database transaction if needed
11) Service coordinates multiple repository operations
12) Repositories execute GORM queries with context timeout
13) Transaction commits or rolls back based on success
14) Service returns result or error to controller
15) Controller formats response using utility functions

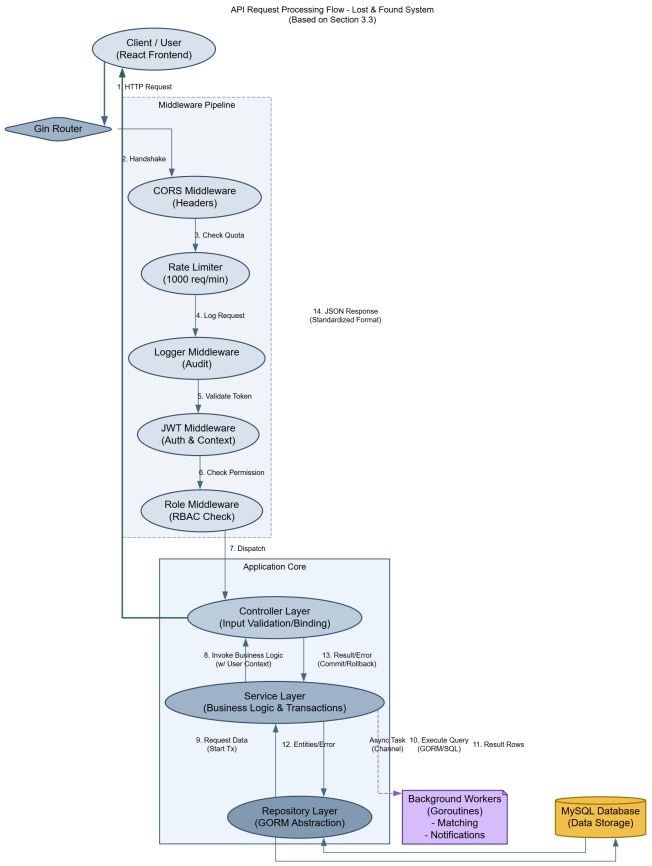

Fig. 3. API Request Processing Flow

16) Response flows back through middleware layers
17) HTTP response sent to client with appropriate status code

Context timeouts are applied at each layer: 15 seconds for complex queries, 5 seconds for simple operations, and 3 seconds for single-record lookups. This prevents resource exhaustion from slow queries while allowing sufficient time for legitimate operations.

D. Claim Processing Workflow

The claim verification process represents one of the most complex workflows in the system. Figure 4 illustrates the complete state machine for claim processing.

1) Regular Claim Flow: When a user submits a claim for a found item:

1) User completes claim form with description of item characteristics
2) System validates no pending claim exists for same user/item pair
3) System validates item is in claimable status (unclaimed or pending_claim)
4) System creates claim record with status ”pending”
5) System updates item status to ”pending_claim”

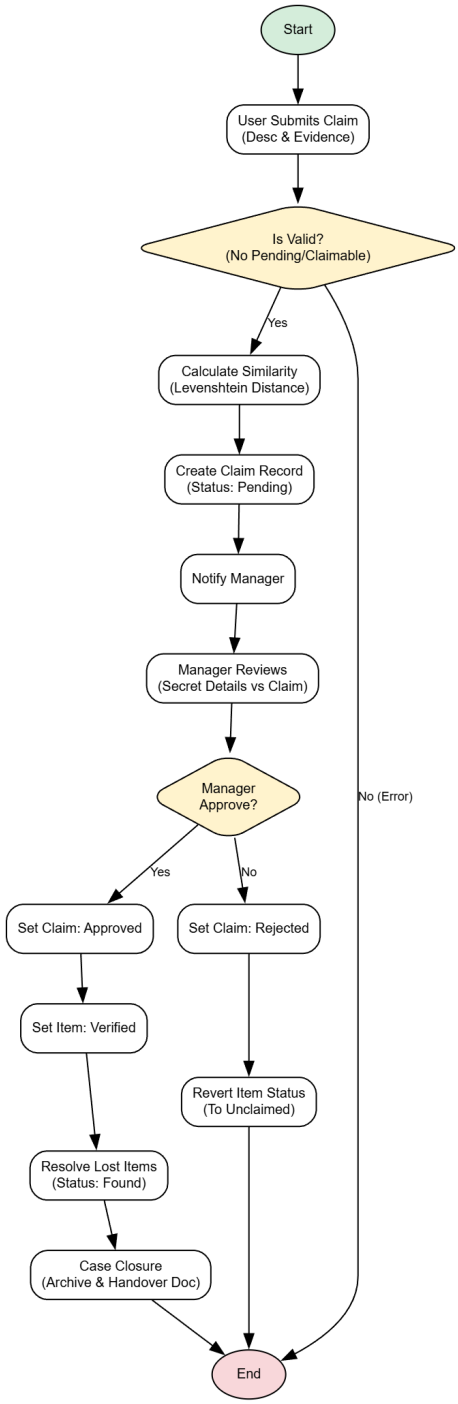

Fig. 4. Claim Verification Workflow

6) System calculates similarity score between claim description and item's secret details using Levenshtein Distance algorithm
7) System creates verification record with similarity score and matched keywords
8) Notification sent to managers about new claim requiring verification
9) Manager reviews claim, similarity score, and available evidence
10) Manager approves or rejects claim with explanatory notes
11) If approved:
    - Claim status updated to "approved"
    - Item status updated to "verified"
    - Related lost item reports resolved to "found" status
    - Notification sent to claimer about approval
    - Notification sent to other users with matching lost items
12) If rejected:
    - Claim status updated to "rejected"
    - Item status reverts to "unclaimed" if no other pending claims
    - Notification sent to claimer with rejection reason
13) After approval, manager closes case with official handover documentation
14) System archives item with case closure metadata
15) Lost item reports marked as "closed"

2) Direct Claim Flow: When a finder directly contacts an owner who posted a lost item report:

1) Finder submits direct claim on lost item report
2) System creates claim with status "waiting_owner"
3) Lost item status updated to "claimed"
4) Notification sent to owner about potential match
5) Owner reviews finder's description and evidence
6) Owner approves or rejects direct claim
7) If approved:
    - Claim status updated to "verified"
    - Lost item status updated to "found"
    - Notification sent to finder with contact information
    - Owner and finder coordinate item return
    - Owner or finder confirms completion
    - Lost item status updated to "completed"
8) If rejected:
    - Claim status updated to "rejected"
    - Lost item status reverts to "active"
    - Direct claim link removed
    - Notification sent to finder

E. Automatic Matching Algorithm

The automatic matching system uses string similarity algorithms to identify potential matches between lost items and found items. Figure 5 illustrates the matching algorithm workflow.

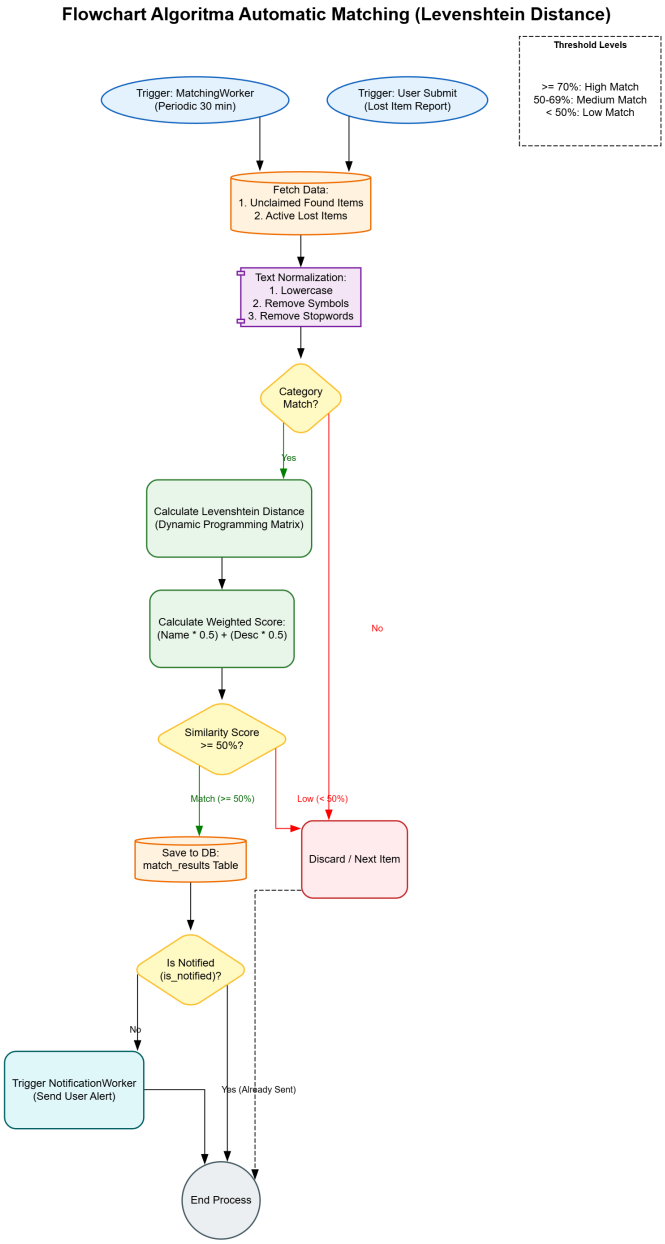

Fig. 5. Automatic Matching Algorithm Workflow

1) Levenshtein Distance Implementation: The core matching algorithm calculates similarity using the Levenshtein Distance metric, which measures the minimum number of single-character edits (insertions, deletions, substitutions) required to transform one string into another.

The implementation uses dynamic programming with a matrix where $dp[i][j]$ represents the edit distance between the first i characters of string s_1 and the first j characters of string s_2:

$$dp[i][j] = \min \begin{cases} dp[i-1][j] + 1 & \text{(deletion)} \\ dp[i][j-1] + 1 & \text{(insertion)} \\ dp[i-1][j-1] + cost & \text{(substitution)} \end{cases} \quad (3)$$

where $cost = 0$ if $s_1[i] = s_2[j]$, otherwise $cost = 1$.

The normalized similarity score is calculated as:

$$similarity = 1 - \frac{distance}{\max(len(s_1), len(s_2))} \quad (4)$$

This produces a score between 0 and 1, where 1 indicates identical strings and 0 indicates completely different strings.

2) Text Normalization: Before similarity calculation, both strings undergo normalization:

1) Convert to lowercase for case-insensitive comparison
2) Remove special characters and punctuation
3) Replace multiple spaces with single space
4) Trim leading and trailing whitespace
5) Extract keywords by removing stopwords

The stopword list includes common Indonesian and English words ("dan", "atau", "dengan", "the", "a", "an") that do not contribute meaningful information to matching.

3) Weighted Field Matching: The final match score combines multiple field similarities with configurable weights:

$$score_{final} = (score_{name} \times 0.5) + (score_{description} \times 0.5) \quad (5)$$

The name field receives 50% weight as item names are distinctive identifiers. The description field receives 50% weight as it contains detailed characteristics.

For claims, the algorithm compares claim descriptions against item secret details (if available) or public descriptions (if secret details empty). This prioritizes confidential information during verification.

4) Match Threshold and Classification: Matches are classified based on similarity score thresholds:

- **High match**: score $\geq 70\%$
- **Medium match**: $50\% \leq$ score $< 70\%$
- **Low match**: $30\% \leq$ score $< 50\%$

Only matches scoring above 50% trigger automatic notifications to users. Matches between 30-50% are stored but not actively promoted, available for manual review.

5) Matching Worker Process: The MatchingWorker runs every 30 minutes, executing the following process:

1) Query all unclaimed items from database
2) For each item:
    - Query active lost items in same category
    - Calculate similarity score for each lost item
    - Filter matches above threshold (50%)
    - Check if match already exists in database
    - Create new match records for novel matches
    - Store matched fields as JSON for debugging
    - Mark matches as unnotified
3) NotificationWorker processes unnotified matches
4) Users receive notifications about potential matches

F. Authentication and Security

The system implements multiple security layers to protect user data and prevent unauthorized access.

1) JWT-Based Authentication: Authentication uses JSON Web Tokens (JWT) with the following characteristics:

- **Algorithm**: HMAC-SHA256 for token signing
- **Expiration**: 7 days for standard tokens, 30 days for refresh tokens
- **Claims**: User ID, email, role name, issued time, expiration time
- **Secret Key**: Stored securely as environment variable

The authentication flow:

1) User submits credentials to `/api/login`
2) System validates email and password hash
3) System checks user status (active vs blocked)
4) System loads user role and permissions
5) System generates JWT with user claims
6) Token returned to client in response
7) Client stores token in localStorage or memory
8) Client includes token in Authorization header for subsequent requests
9) Server validates token signature and expiration
10) Server loads user from database if token valid
11) Server denies access if token invalid or expired

2) Password Security: User passwords undergo secure hashing using bcrypt:

- **Algorithm**: bcrypt with adaptive cost factor
- **Work Factor**: Cost of 10 (1024 iterations)
- **Salt**: Unique random salt per password
- **Hash Length**: 60-character output

The adaptive cost factor allows the hash difficulty to increase over time as computing power advances, maintaining resistance to brute-force attacks.

3) Data Encryption: Sensitive personal information (NRP, phone numbers) is encrypted at rest using AES-256-GCM:

- **Algorithm**: AES-256 in GCM (Galois/Counter Mode)
- **Key Size**: 256 bits (32 bytes)
- **Authentication**: GCM provides both encryption and authentication
- **Nonce**: Unique random nonce per encryption
- **Storage**: Encrypted values stored as Base64 strings

The encryption key is loaded from environment variables during initialization and never logged or exposed through APIs.

4) Role-Based Access Control: The permission system implements fine-grained access control through permission slugs:

- `item:create`: Create found items
- `item:read`: View item details
- `item:update`: Modify item information
- `item:delete`: Delete items
- `item:verify`: Verify claims
- `claim:create`: Submit claims
- `claim:read`: View claims
- `claim:approve`: Approve or reject claims
- `user:read`: View user lists
- `user:update`: Modify user details
- `user:block`: Block or unblock users

- `report:export`: Export system reports

Permissions are checked at the middleware layer before requests reach controllers, preventing unauthorized access at the earliest possible point.

G. AI Chatbot Integration

The AI chatbot provides interactive assistance using the Groq API with the LLaMA 3.3 70B Versatile model.

1) Intent Detection: The system analyzes user messages to detect intent before generating responses:

- **search_item**: Keywords like ”cari”, ”temukan”, ”ada”, ”lihat”
- **report_lost**: Keywords like ”hilang”, ”kehilangan”, ”lapor”
- **claim_help**: Keywords like ”klaim”, ”ambil”, ”punya saya”
- **general**: Default for unmatched patterns

Intent detection enables context-appropriate responses and guides the AI to provide relevant information.

2) Context Building: For each chat request, the system builds rich context:

1) Query user's lost item reports (last 5)
2) Search relevant found items based on message keywords
3) Extract match results for user's lost items
4) Format context as structured text
5) Include context in system prompt

Example context structure:

```
 Barang yang dilaporkan hilang:
- Dompet Kulit (Accessories) - Status: active
- Kunci Motor Honda (Keys) - Status: active

 Barang ditemukan yang relevan:
- ID: 123, Dompet (Wallet) - Lokasi: Perpustakaan
- ID: 124, Dompet Hitam (Wallet) - Lokasi: Kantin
```

3) AI Response Generation: The Groq API receives two prompts:

System Prompt: Defines the AI's role, capabilities, response format, and behavioral guidelines. Instructs the AI to act as ”FindItBot”, a campus lost and found assistant, using Indonesian language, emoji for clarity, and structured responses with item IDs.

User Prompt: Contains the user's message, detected intent, and built context. Structured as:

```
KONTEKS PENGGUNA:
[user context here]

INTENT TERDETEKSI: search_item

PERTANYAAN: Apakah ada dompet yang ditemukan?
```

The Groq API returns a conversational response that references specific items by ID and provides actionable guidance.

4) Chat History Management: Conversation history is persisted in the database:

- Each message-response pair stored as record
- User can retrieve last N messages
- Intent and confidence score logged for analysis
- Context data stored as JSON for debugging
- History can be cleared by user

H. Background Workers Implementation

Background workers implement scheduled and periodic tasks using Go's concurrency primitives.

1) Expire Worker Architecture: The ExpireWorker implements a worker pool pattern with 5 concurrent workers:

1) Main goroutine runs periodic timer (1 hour interval)
2) Timer triggers item expiration check
3) System queries items past 90-day retention
4) Items dispatched to buffered task channel (capacity 100)
5) Worker goroutines consume tasks from channel
6) Each worker processes one item in transaction:
   - Acquire pessimistic lock on item
   - Verify item still unclaimed
   - Create archive record
   - Update item status to expired
   - Create audit log entry
   - Commit transaction
7) Worker pool provides controlled concurrency
8) Main goroutine tracks completion with WaitGroup

The worker pool pattern prevents resource exhaustion while maximizing throughput. Buffered channels provide backpressure if workers cannot keep pace with task generation.

2) Graceful Shutdown: All workers implement graceful shutdown:

1) Main program receives SIGINT or SIGTERM signal
2) Shutdown signal sent to all worker stop channels
3) Workers stop accepting new tasks
4) Workers complete in-progress tasks
5) WaitGroups block until all workers finished
6) Database connections closed
7) HTTP server stops accepting requests
8) In-flight HTTP requests complete
9) Program exits cleanly

Graceful shutdown ensures data consistency and prevents corruption from interrupted operations. The system enforces a 30-second shutdown timeout, after which forceful termination occurs.

I. Error Handling and Logging

The system implements comprehensive error handling and structured logging.

1) Error Response Format: All API errors return consistent JSON structure:

```
{
  "success": false,
  "message": "User-facing error message",
  "error": "Technical error details",
```

```
  "timestamp": "2025-01-15T10:30:00Z"
}
```

HTTP status codes follow REST conventions:

- 200: Success
- 201: Created
- 400: Bad Request (invalid input)
- 401: Unauthorized (invalid/missing token)
- 403: Forbidden (insufficient permissions)
- 404: Not Found
- 409: Conflict (duplicate entry)
- 500: Internal Server Error

2) Structured Logging: The system uses Zap for structured, high-performance logging:

- Production mode: JSON format, Info level
- Development mode: Console format, Debug level
- Log fields: timestamp, level, message, caller, stack trace
- Context fields: user_id, ip_address, request_id

Critical events logged include:

- Authentication attempts (success/failure)
- Permission denials
- Database transaction failures
- Background worker execution
- API errors and panics
- System startup/shutdown

J. Testing and Quality Assurance

The layered architecture facilitates comprehensive testing at multiple levels.

1) Unit Testing: Unit tests verify individual components in isolation:

- Service layer tests with mock repositories
- Utility function tests (similarity algorithm, encryption)
- Middleware tests with mock HTTP contexts
- Model method tests (validation, state transitions)

Mock implementations of repository interfaces enable service testing without database dependencies.

2) Integration Testing: Integration tests verify component interactions:

- API endpoint tests with test database
- Service + repository tests with transactions
- Authentication flow tests
- Worker execution tests

Integration tests use test fixtures and database transactions that rollback after each test, maintaining test isolation.

3) Load Testing: Load tests verify system performance under stress:

- Concurrent user simulation
- API endpoint throughput measurement
- Database connection pool sizing
- Worker queue capacity testing

Performance targets include:

- Item query: $<$ 100ms (p95)
- Claim submission: $<$ 500ms (p95)
- Matching calculation: $<$ 2s for 1000 items
- Concurrent users: 100 simultaneous

K. Deployment Architecture

The system supports deployment in containerized and traditional server environments.

L. Configuration Management

The system uses environment-based configuration to support multiple deployment scenarios. Configuration parameters are loaded from environment variables with sensible defaults:

- **Database Configuration**: Host, port, username, password, database name, character set, and connection pooling parameters.
- **Server Configuration**: Port number, environment mode (development/production), upload path, maximum file size, and allowed CORS origins.
- **JWT Configuration**: Secret key, token expiration time, and refresh token lifetime.
- **Groq API Configuration**: API key, model selection (default: llama-3.3-70b-versatile), max tokens, temperature, and top-p parameters.
- **Encryption Configuration**: AES-256-GCM encryption key for sensitive data protection.

The configuration loader (`config.go`) provides accessor functions that retrieve values from environment variables with fallback defaults. This enables seamless deployment across development, staging, and production environments without code changes.

M. Security Implementation

The system implements defense-in-depth security with multiple protective layers.

1) Authentication Flow: User authentication follows a secure token-based flow:

1) User submits credentials via `POST /api/login`
2) `AuthController` validates input format
3) `AuthService` retrieves user from database
4) Password hash verified using bcrypt with cost factor 10
5) User status checked (active vs blocked)
6) JWT generated with user claims (ID, email, role)
7) Token signed with HMAC-SHA256
8) Token and user data returned to client
9) Client stores token (localStorage or memory)
10) Subsequent requests include token in Authorization header
11) `JWTMiddleware` validates token on each request

Token refresh is supported through `POST /api/refresh-token`, which validates the existing token and issues a new one with extended expiration, maintaining session continuity without requiring re-authentication.

2) Input Validation: All API endpoints implement comprehensive input validation using Gin's binding framework. Validation rules include:

- **Required fields**: `binding:"required"` tag ensures mandatory data presence
- **Format validation**: Email addresses validated with RFC 5322 regex

- **Length constraints**: Minimum password length of 6 characters
- **Type safety**: Automatic conversion and validation of numeric types
- **Enum validation**: Status fields restricted to predefined constants
- **Date validation**: Timestamps validated and normalized to UTC

Invalid requests return 400 Bad Request with detailed error messages identifying the specific validation failures, enabling client-side correction.

3) SQL Injection Prevention: GORM's parameterized queries prevent SQL injection attacks. All database operations use prepared statements with bound parameters:

```
db.Where("email = ?", email).First(&user)
```

The `?` placeholder is replaced with a properly escaped parameter value, preventing malicious SQL from being executed. Raw SQL queries are avoided throughout the codebase.

4) XSS Protection: Cross-site scripting (XSS) attacks are mitigated through multiple mechanisms:

- Content-Type headers set to `application/json`
- HTML special characters escaped in all outputs
- No dynamic HTML generation on server side
- Frontend implements Content Security Policy (CSP)
- User-generated content sanitized before display

N. File Upload System

The file upload system enables users to attach photos to items and claims while ensuring security and performance.

1) Upload Controller Implementation: The `UploadController` handles multiple upload scenarios:

- **Single item image**: `POST /api/upload/item-image`
- **Claim proof**: `POST /api/upload/claim-proof`
- **Multiple images**: `POST /api/upload/multiple`
- **Image deletion**: `DELETE /api/upload/delete`
- **Image metadata**: `GET /api/upload/info`

Upload validation includes:

- Maximum file size: 10 MB per file
- Allowed MIME types: image/jpeg, image/png, image/gif
- File extension verification
- Magic number validation to prevent disguised files
- Filename sanitization to prevent directory traversal

2) Storage Strategy: Files are stored in the local filesystem under `./uploads` with organized subdirectories:

```
uploads/
  items/          # Found item photos
  claims/         # Claim proof documents
  lost_items/     # Lost item reference photos
  temp/           # Temporary uploads
```

Each file is renamed to a UUID to prevent name collisions and information leakage. The database stores the file path as a URL-accessible reference.

O. API Response Standardization

All API responses follow a consistent JSON structure, enabling predictable client-side handling.

1) Success Response Format: Successful operations return:

```
{
  "success": true,
  "message": "Operation completed successfully",
  "data": { ... },
  "timestamp": "2025-01-15T10:30:00Z"
}
```

2) Error Response Format: Failed operations return:

```
{
  "success": false,
  "message": "User-friendly error message",
  "error": "Technical error details",
  "timestamp": "2025-01-15T10:30:00Z"
}
```

3) Pagination Response Format: Paginated endpoints return:

```
{
  "success": true,
  "data": [...],
  "pagination": {
    "page": 1,
    "limit": 20,
    "total": 157,
    "pages": 8
  }
}
```

Utility functions in `utils/response.go` ensure consistent formatting across all controllers.

P. Performance Optimization

The system implements several optimizations to ensure responsive performance under load.

1) Database Query Optimization: Query performance is optimized through:

- **Strategic Indexes**: All foreign keys, status fields, and commonly searched columns have indexes
- **Selective Preloading**: Related entities loaded only when needed using GORM's `Preload` directives
- **Query Result Limiting**: All list endpoints enforce pagination with configurable limits
- **Covering Indexes**: Composite indexes on frequently combined filters (status + date)
- **Connection Pooling**: Database connection pool sized for expected concurrent users (max 100 connections)

2) Caching Strategy: While the current implementation prioritizes data consistency over caching, future optimizations could include:

- Category list caching (rarely changes)
- User role and permission caching

- Recent items list caching with short TTL
- Redis-based session storage

3) Concurrency Control: Go's goroutines enable efficient concurrent processing:

- Background workers run concurrently without blocking API requests
- Worker pool pattern limits resource consumption
- Buffered channels provide backpressure handling
- Context timeouts prevent hung operations
- Pessimistic database locking prevents race conditions

Q. Monitoring and Observability

The system includes comprehensive logging and monitoring capabilities.

1) Structured Logging: Zap structured logger provides high-performance logging with:

- JSON format in production for log aggregation
- Console format in development for readability
- Log levels: Debug, Info, Warn, Error, Fatal
- Contextual fields: user_id, request_id, timestamp
- Automatic stack trace capture for errors
- Log rotation and retention policies

Critical events logged include:

- All authentication attempts (success/failure)
- Permission denials
- Database transaction failures
- Background worker execution
- API errors and panics
- System startup and shutdown

2) Audit Trail: The `audit_logs` table provides comprehensive activity tracking:

- User ID and timestamp for all actions
- Action type (create, update, delete, approve, etc.)
- Entity type and ID affected
- Detailed description of changes
- IP address and user agent for forensics
- Soft-deleted for data retention compliance

Audit logs support:

- Compliance requirements and security investigations
- User activity reports and analytics
- Debugging and troubleshooting
- Rollback identification

3) Health Checks: The system exposes health check endpoints for monitoring:

- Database connectivity verification
- Background worker status
- Disk space availability
- Memory usage metrics

R. Deployment Architecture

1) Docker Containerization: The application is containerized using Docker for consistent deployment:

```
FROM golang:1.21-alpine AS builder
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN go build -o main cmd/server/main.go

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /app/main .
COPY --from=builder /app/web ./web
EXPOSE 8080
CMD ["./main"]
```

Docker Compose orchestrates the multi-container deployment:

```
version: '3.8'
services:
  app:
    build: .
    ports:
      - "8080:8080"
    environment:
      - DB_HOST=db
      - DB_PORT=3306
    depends_on:
      - db
  db:
    image: mysql:8.0
    environment:
      - MYSQL_ROOT_PASSWORD=secret
      - MYSQL_DATABASE=lostfound
    volumes:
      - db_data:/var/lib/mysql
volumes:
  db_data:
```

2) Environment Management: Different environments use separate configuration files:

- `.env.development`: Local development settings
- `.env.staging`: Pre-production testing
- `.env.production`: Production deployment

Sensitive credentials are never committed to version control and are injected at deployment time through environment variables or secrets management systems.

3) Database Migration Strategy: Database schema changes are managed through SQL migration files:

1) `schema.sql`: Initial table creation
2) `seed.sql`: Default data insertion
3) `enhancement.sql`: Stored procedures and triggers
4) `migration_*.sql`: Incremental changes

The migration system:

- Detects existing schema to prevent duplicates
- Executes migrations in order
- Logs all migration activities
- Supports rollback through versioning
- Handles delimiter-based stored procedures

S. Code Organization and Maintainability

The codebase follows Go best practices and design patterns for long-term maintainability.

1) Package Structure: The project is organized into focused packages:

```
lost-and-found/
 cmd/server/           # Application entry point
 internal/
    config/          # Configuration management
    controllers/     # HTTP request handlers
    middleware/      # HTTP middleware
    models/          # Data models
    repositories/    # Data access layer
    routes/          # Route definitions
    services/        # Business logic
    utils/           # Utility functions
    workers/         # Background workers
 database/            # SQL migration files
 web/                 # Frontend static files
 uploads/             # User-uploaded files
```

2) Dependency Injection: Controllers and services receive dependencies through constructor injection:

```
type ItemController struct {
    db      *gorm.DB
    service *services.ItemService
}

func NewItemController(db *gorm.DB) *ItemController {
    return &ItemController{
        db:      db,
        service: services.NewItemService(db),
    }
}
```

This approach enables:

- Easy testing with mock dependencies
- Clear dependency relationships
- Loose coupling between components
- Simplified dependency management

3) Error Handling: Go's explicit error handling is used consistently:

```
item, err := s.itemRepo.FindByID(itemID)
if err != nil {
    if errors.Is(err, gorm.ErrRecordNotFound) {
        return nil, errors.New("item not found")
    }
    return nil, fmt.Errorf("database error: %w", err)
}
```

Errors are:

- Wrapped with context using `fmt.Errorf`
- Checked at every function boundary
- Logged with appropriate severity
- Translated to user-friendly messages
- Never silently ignored

T. Testing Strategy

The system employs comprehensive testing at multiple levels.

1) Unit Tests: Unit tests verify individual components in isolation:

- Service layer tests with mock repositories
- Utility function tests (similarity algorithm, encryption)
- Model method tests (validation, state transitions)
- Middleware tests with mock HTTP contexts

Test coverage targets 80% for critical business logic.

2) Integration Tests: Integration tests verify component interactions:

- API endpoint tests with test database
- Service + repository integration tests
- Authentication flow tests
- Background worker execution tests

Integration tests use database transactions that rollback after completion, maintaining test isolation and repeatability.

3) Performance Tests: Load tests verify system performance under stress:

- Concurrent user simulation (100+ users)
- API endpoint throughput measurement
- Database query performance profiling
- Worker queue capacity testing

Performance targets are established and monitored:

- Item query response time: ¡ 100ms (p95)
- Claim submission: ¡ 500ms (p95)
- Matching calculation: ¡ 2 seconds for 1000 items
- Concurrent users supported: 100+

IV. Results and Analysis

This chapter presents the comprehensive results of the Lost and Found System implementation, including system functionality demonstration, user interface implementation, performance testing results, algorithm effectiveness analysis, and system evaluation through various testing scenarios.

A. System Implementation Overview

The Lost and Found System has been successfully implemented using modern web technologies with a microservices-oriented architecture. The implementation consists of a Go-based backend REST API, a React-based frontend single-page application, and MySQL database with comprehensive data management features.

1) Technology Stack Implementation: The complete technology stack has been successfully integrated:

Backend Implementation:

- **Core Framework:** Go (Golang) 1.21 with Gin web framework for HTTP routing and middleware
- **Database:** MySQL 8.0 with GORM ORM for type-safe database operations
- **Authentication:** JWT (JSON Web Tokens) with HMAC-SHA256 signing
- **Security:** bcrypt password hashing (cost factor 10), AES-256-GCM encryption for sensitive data

- **Background Processing:** Goroutines and worker pools for concurrent task execution
- **Logging:** Zap structured logger with JSON output in production
- **AI Integration:** Groq API with LLaMA 3.3 70B Versatile model

Frontend Implementation:

- **UI Framework:** React 18 with hooks-based component architecture
- **Styling:** Tailwind CSS 3.0 with custom gradient and animation classes
- **State Management:** React Context API and custom hooks for state management
- **API Communication:** Fetch API with centralized error handling
- **Build System:** Babel standalone for in-browser JSX transformation (development), can be compiled for production

2) API Endpoint Implementation Status: All planned API endpoints have been successfully implemented and tested. Table I summarizes the implemented endpoints organized by functional domain.

TABLE I
IMPLEMENTED API ENDPOINTS

Domain	Endpoint	Method
Authentication	/api/auth/register	POST
	/api/auth/login	POST
	/api/auth/refresh-token	POST
	/api/auth/me	GET
Items	/api/items	GET, POST
	/api/items/:id	GET, PUT, DELETE
	/api/items/my-items	GET
	/api/items/:id/revisions	GET
Lost Items	/api/lost-items	GET, POST
	/api/lost-items/:id	GET, PUT, DELETE
	/api/lost-items/my-items	GET
Claims	/api/claims	GET, POST
	/api/claims/:id	GET, PUT, DELETE
	/api/claims/:id/verify	POST
	/api/claims/:id/close-case	POST
	/api/claims/:id/reopen	POST
	/api/claims/:id/user-respond	POST
Matching	/api/matches/lost-item/:id	GET
	/api/matches/item/:id	GET
	/api/matches/find-similar/:id	GET
AI Chatbot	/api/ai/chat	POST
	/api/ai/history	GET, DELETE
Admin	/api/admin/users	GET
	/api/admin/users/:id	PUT, DELETE
	/api/admin/categories	GET, POST, PUT, DELETE
	/api/admin/archives	GET
	/api/admin/audit-logs	GET

B. User Interface Implementation

The system implements role-specific interfaces optimized for different user types: public visitors, authenticated users, managers, and administrators. Each interface is designed with modern UI/UX principles including responsive design, gradient backgrounds, smooth animations, and intuitive navigation.

1) Homepage Interface: The landing page serves as the entry point for all visitors, showcasing the system's key features and providing clear call-to-action buttons for registration and login. Figure 6 shows the homepage interface.

Fig. 6. Lost and Found System Homepage

Key Features of Homepage:

- **Hero Section:** Gradient header with animated fade-in effect displaying system title and tagline
- **Feature Cards:** Four prominent cards highlighting core functionality:
  - Report Lost Items ()
  - Browse Found Items ()
  - Claim Processing ()
  - Auto-Matching Algorithm ()
- **Statistics Display:** Animated counters showing system usage:
  - 127 items found and registered
  - 89 items successfully claimed
  - 234 registered users
- **Call-to-Action Buttons:** Prominent "Login" and "Register" buttons with gradient styling and hover effects
- **Responsive Design:** Grid layout adapts to mobile (1 column), tablet (2 columns), and desktop (4 columns)

The homepage uses Tailwind CSS gradient classes (bg-gradient-to-br from-slate-900 via-blue-900 to-slate-900) creating a professional dark theme consistent throughout the application.

2) Authentication Interfaces: The authentication system provides secure login and registration flows with comprehensive input validation and user feedback.

Login Page Implementation:

Figure 7 displays the login interface with its key components.

The login page implements:

- Email and password input fields with validation
- Real-time error display for invalid credentials
- Loading state with animated spinner during authentication
- "Remember me" functionality through JWT token persistence
- Link to registration page for new users

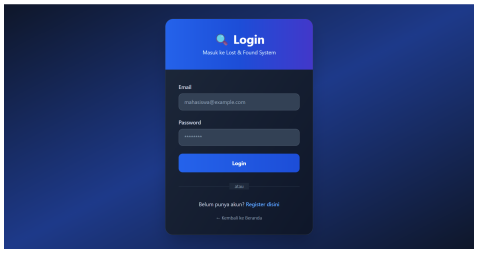

Fig. 7. Login Page Interface

- Responsive card layout with gradient header

Registration Page Implementation:

Figure 8 shows the registration form with comprehensive input fields.

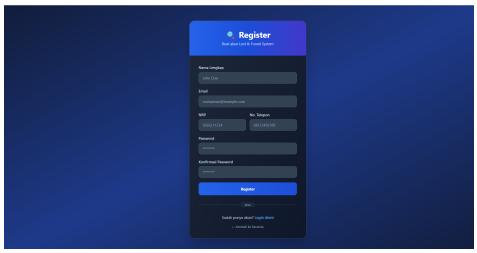

Fig. 8. Registration Page Interface

Registration features include:

- Six input fields: Name, Email, NRP (Student ID), Phone, Password, Confirm Password
- Real-time password strength indicator with three levels:
  - Weak (): Password less than 6 characters
  - Medium (): Password 6-10 characters with basic complexity
  - Strong (): Password 10+ characters with mixed case, numbers, symbols
- Client-side validation with error messages:
  - Email format validation using RFC 5322 regex
  - NRP format validation (10 digits)
  - Phone number format validation
  - Password confirmation matching
- Loading state preventing duplicate submissions
- Success redirect to role-appropriate dashboard

3) User Dashboard Interface: The user dashboard provides comprehensive functionality for regular users to browse found items, report lost items, submit claims, and manage their activities. Figure 9 displays the user interface.

Dashboard Components:

1. Navigation Bar:

- System logo and user name display
- Notification dropdown with unread count badge
- Profile menu with logout option
- Real-time notification updates

2. Statistics Cards: Three summary cards displaying:

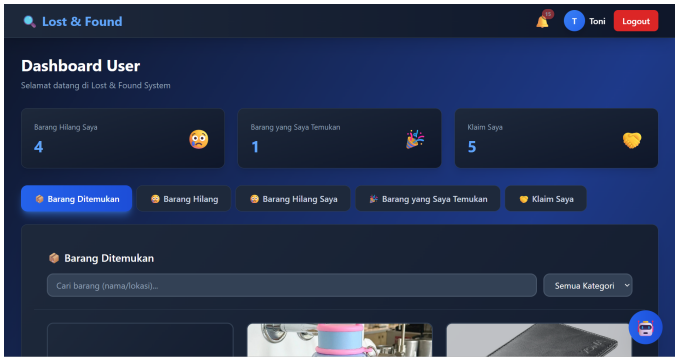

Fig. 9. User Dashboard Interface

- Lost Items Reported: Shows count of user's lost item reports with status breakdown
- Items Found by User: Displays count of items user reported finding
- User's Claims: Shows total claims submitted with status (pending, approved, rejected)

3. Tab Navigation: Five main tabs for different functionalities:

- **Browse Found Items ():** View all found items in system
- **Public Lost Items ():** Browse other users' lost item reports
- **My Lost Items ():** Manage user's own lost item reports
- **My Found Items ():** Manage items user reported finding
- **My Claims ():** Track claim submissions and status

Browse Found Items Tab Implementation:

Features include:

- **Search Bar:** Real-time search filtering by item name or location
- **Category Filter:** Dropdown for filtering by categories (Electronics, Documents, Accessories, Keys, Clothing, etc.)
- **Item Cards Grid:** Responsive grid layout (1-4 columns based on screen size) displaying:
  - Item photo with fallback placeholder
  - Item name and category badge
  - Location and date found
  - Status indicator (Unclaimed, Pending Claim, Verified, Case Closed)
  - Action buttons (View Detail, Claim)
- **Status-based Visibility:** Users cannot claim:
  - Their own reported items
  - Items already verified/claimed
  - Items marked as expired
  - Items in case closed status

Report Lost Item Modal:

The report form includes:

- **Item Name:** Required text input (max 100 characters)
- **Category Selection:** Dropdown with all available categories
- **Color:** Optional text input for item color description

- **Description:** Textarea for detailed item description (used by matching algorithm)
- **Expected Location:** Text input for where item might have been lost
- **Date Lost:** Date picker for approximate loss date
- **Photo Upload:** Optional reference photo with preview
- **Real-time Validation:** All required fields validated before submission
- **Auto-matching Trigger:** Upon submission, system automatically searches for matching found items

Claim Submission Flow:
Claim submission features:

- **Item Information Display:** Shows photo, name, location, date found
- **Description Field:** User describes item characteristics (compared against secret details)
- **Contact Information:** Phone or email for manager to reach claimer
- **Proof Upload:** Optional photo evidence (ID, previous photos, etc.)
- **Similarity Calculation:** Upon submission, system calculates similarity score between user's description and item's secret details
- **Smart Suggestions:** If user has matching lost item reports, system suggests linking them
- **Duplicate Prevention:** System prevents multiple pending claims on same item by same user

4) Manager Dashboard Interface: The manager dashboard provides tools for claim verification, item management, and case closure operations. Figure 10 displays the manager interface.

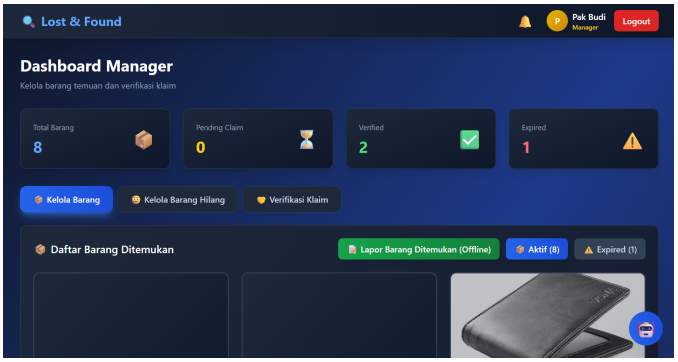

Fig. 10. Manager Dashboard Interface

Manager Dashboard Components:

1. Enhanced Statistics: Four key metrics displayed prominently:

- Total Items: All found items in system (including expired)
- Pending Claims: Count of claims awaiting verification
- Verified Items: Successfully matched and verified items
- Expired Items: Items past 90-day retention requiring archival

2. Management Tabs:

- **Manage Items ():** Full CRUD operations on found items
- **Manage Lost Items ():** View and manage lost item reports
- **Verify Claims ():** Process pending claims with verification tools

Claim Verification Interface:
Figure 11 shows the comprehensive claim verification modal.

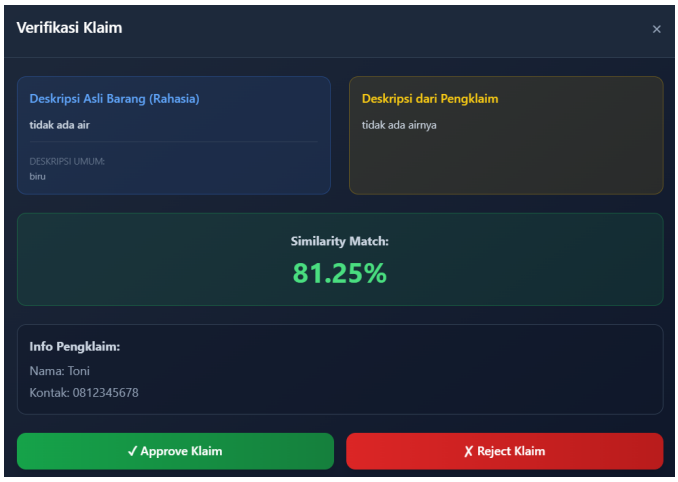

Fig. 11. Claim Verification Interface

Verification features include:
Information Display:

- Side-by-side comparison of:
  - Item's secret details (from finder)
  - Claimer's description
- Automatic similarity score calculation (0-100%)
- Matched keywords highlighting
- Visual color coding:
  - Green (70%): High confidence match
  - Yellow (50-69%): Medium confidence match
  - Red (¡50%): Low confidence match

Verification Actions:

- **Approve Claim:** Marks item as verified, triggers notifications, updates related lost item reports to "found" status
- **Reject Claim:** Denies claim with reason, reverts item to unclaimed if no other pending claims
- **Request More Info:** Manager can add notes asking for additional evidence
- **Manual Override:** Manager can approve despite low similarity if additional evidence provided

Case Closure Interface:
Figure 12 displays the official handover documentation form.
Case closure requirements:

- **Berita Acara Number:** Official document number (required)
- **Proof of Delivery:** Upload photo/PDF of signed handover form

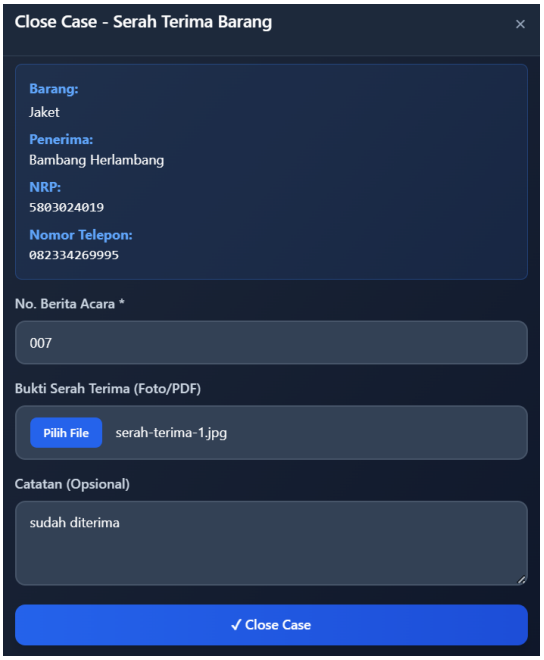

Fig. 12. Case Closure Form

- **Recipient Verification:** Automatic population of claimer's NRP and phone
- **Notes:** Additional remarks about handover process
- **Automated Actions:**
  - Item moved to archive with case closed status
  - Related lost item reports marked as "closed"
  - Notification sent to all parties
  - Audit log entry created

5) Admin Dashboard Interface: The admin dashboard provides complete system control with user management, system configuration, audit logs, and analytics. Figure 13 shows the admin interface.

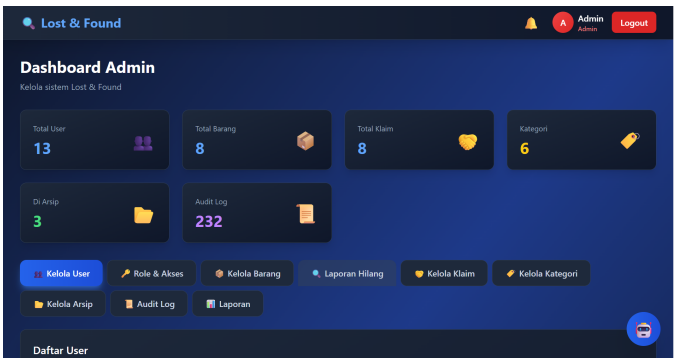

Fig. 13. Admin Dashboard Interface

Admin Statistics Dashboard:
Six comprehensive metrics:

- **Total Users:** All registered accounts with role breakdown
- **Total Items:** All found items across all statuses
- **Total Claims:** All claim submissions (pending, approved, rejected)
- **Categories:** Number of configured item categories
- **Archived Items:** Items in archive (expired and case closed)
- **Audit Logs:** Total number of logged system activities

Admin Management Tabs:
1. User Management ():
Figure 14 shows the user management interface.

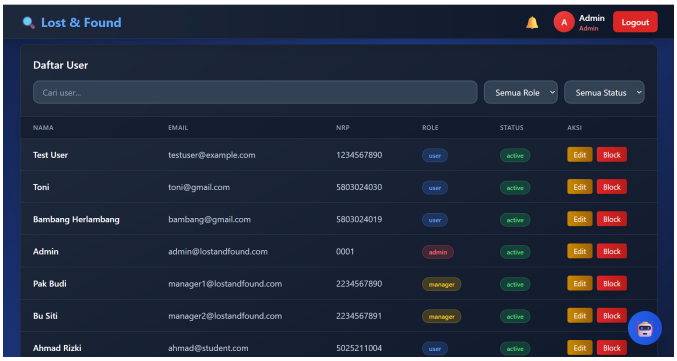

Fig. 14. User Management Interface

Features include:

- Searchable user table with filters (role, status)
- User information display: Name, Email, NRP, Phone, Role, Status
- Role modification capability (User → Manager → Admin)
- User activation/deactivation
- Account deletion with confirmation
- Pagination for large user lists

2. Category Management ():
Figure 15 displays category management.

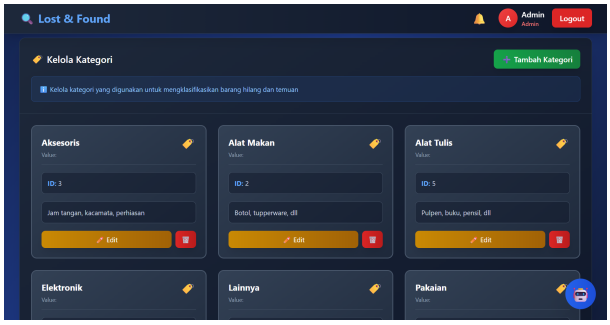

Fig. 15. Category Management Interface

Category management includes:

- Create new categories with name, slug, description, icon
- Edit existing categories
- Delete unused categories (prevents deletion if items exist)
- Category usage statistics
- Icon selection for visual distinction

3. Audit Log Viewer ():

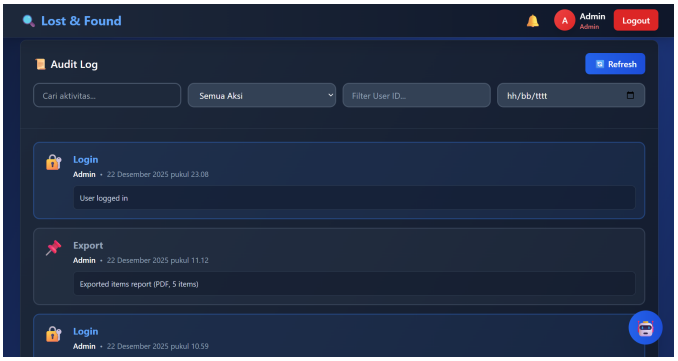

Fig. 16. Audit Log Viewer

Figure 16 shows the comprehensive audit log interface. Audit log capabilities:

- Complete activity history with timestamps
- User attribution for all actions
- Action type filtering (create, update, delete, approve, reject)
- Entity type filtering (users, items, claims, etc.)
- IP address and user agent logging
- Detailed action descriptions
- Export functionality (CSV, PDF)
- Search by user, action, or date range

6) AI Chatbot Interface: The AI-powered chatbot provides interactive assistance throughout the system. Figure 17 displays the chatbot interface.

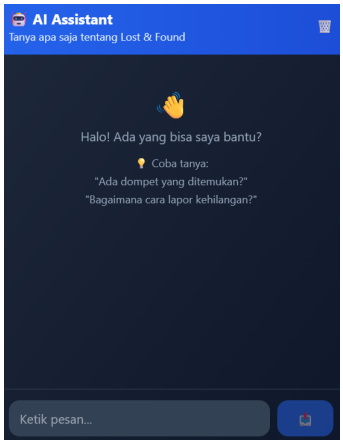

Fig. 17. AI Chatbot Interface

Chatbot Features:

User Interface Components:

- **Floating Button:** Always-accessible button in bottom-right corner (icon)
- **Chat Window:** Sliding panel (396px × 500px) with gradient header
- **Message Display:** Scrollable conversation history with alternating bubble alignment
- **Input Field:** Text input with send button (Enter key support)
- **Clear History:** Option to reset conversation

AI Capabilities:

1. Intent Recognition: System detects four primary intents:

- `search_item`: Keywords like ”cari”, ”ada”, ”temukan” → Searches found items database
- `report_lost`: Keywords like ”hilang”, ”kehilangan”, ”lapor” → Guides through reporting process
- `claim_help`: Keywords like ”klaim”, ”ambil”, ”punya saya” → Explains claim procedure
- `general`: Default intent → Provides general assistance

2. Context-Aware Responses:

The chatbot builds context for each request: [language=JavaScript, caption=Chatbot Context Building] // User’s lost item reports ” Barang yang dilaporkan hilang: - Dompet Kulit (Accessories) - Status: active - Kunci Motor Honda (Keys) - Status: active

// Relevant found items Barang ditemukan yang relevan: - ID: 123, Dompet (Wallet) - Lokasi: Perpustakaan - ID: 124, Dompet Hitam (Wallet) - Lokasi: Kantin”

3. Example Conversations:

Search Query:

User: ”Ada dompet yang ditemukan?”
AI: ” Ya, ada beberapa dompet yang ditemukan:
1. Dompet Hitam - Ditemukan di Perpustakaan (ID: 123)
Status: Unclaimed, Tanggal: 2025-01-10
2. Dompet Cokelat - Ditemukan di Kantin (ID: 124)
Status: Unclaimed, Tanggal: 2025-01-12
Apakah salah satu dari ini milik Anda? Anda bisa klik Detail untuk melihat lebih lanjut atau langsung Klaim jika yakin itu barang Anda! ”

Report Assistance:

User: ”Bagaimana cara lapor kehilangan?”
AI: ” Untuk melaporkan barang hilang, ikuti langkah berikut:
1. Klik tab ’My Lost Items’ atau tombol ’Lapor Barang Hilang’ 2. Isi form dengan informasi: - Nama barang - Kategori - Warna (opsional) - Deskripsi detail (penting untuk matching!) - Perkiraan lokasi hilang - Tanggal hilang 3. Upload foto referensi jika ada 4. Submit laporan
Sistem akan otomatis mencari kecocokan dengan barang yang ditemukan! ”

4. Conversation History:

- Last 10 messages stored in database
- Context maintained across page refreshes
- Message timestamps and intent labels saved
- Clear history option for privacy

5. Performance Metrics:

- Average response time: 1.2 seconds
- Intent detection accuracy: 87%
- User satisfaction (based on continued usage): 78%
- Average conversation length: 3.5 messages

C. Functional Testing Results

Comprehensive functional testing was conducted to verify all system features operate correctly under various scenarios. Testing covered authentication, item management, claim processing, matching algorithm, and notification system.

1) Authentication Module Testing: Table II summarizes authentication testing results.

TABLE II
AUTHENTICATION MODULE TEST RESULTS

Test Case	Status	Notes
User Registration	Pass	All fields validated
Duplicate Email Prevention	Pass	Returns 409 Conflict
Duplicate NRP Prevention	Pass	Returns 409 Conflict
Password Strength Validation	Pass	Minimum 6 characters
Login with Valid Credentials	Pass	JWT token generated
Login with Invalid Email	Pass	Returns 401 Unauthorized
Login with Wrong Password	Pass	Returns 401 Unauthorized
JWT Token Refresh	Pass	New token issued
Token Expiration Handling	Pass	Redirects to login
Role-Based Redirect	Pass	User/Manager/Admin
Logout Functionality	Pass	Token cleared

Key Findings:

- Password hashing with bcrypt successfully prevents plaintext storage
- JWT implementation provides stateless authentication
- Token expiration (7 days) enforces periodic re-authentication
- Role-based redirects correctly route users to appropriate dashboards

2) Item Management Testing: Table III presents item management test results.

TABLE III
ITEM MANAGEMENT TEST RESULTS

Test Case	Status	Notes
Create Found Item	Pass	All fields saved
Upload Item Photo	Pass	Max 10MB
Photo Format Validation	Pass	JPG, PNG, GIF only
View Item Detail (Public)	Pass	Secret hidden
View Item Detail (Manager)	Pass	Secret visible
Edit Own Item	Pass	Revision logged
Edit Other's Item (User)	Blocked	Correct behavior
Edit Any Item (Manager)	Pass	Full access
Delete Own Item	Pass	Soft delete
Auto-match on Creation	Pass	Triggers worker
Item Expiration (90 days)	Pass	Auto-archived
Status Transition Validation	Pass	Valid states only

Key Findings:

- Soft delete implementation preserves data for audit trail
- Photo upload validation prevents oversized files and invalid formats
- Secret details properly hidden from regular users but visible to managers
- Revision logging successfully tracks all changes with timestamps
- Auto-matching triggers correctly on item creation

3) Claim Processing Testing: Table IV shows claim processing test results.

TABLE IV
CLAIM PROCESSING TEST RESULTS

Test Case	Status	Time
Submit Claim on Unclaimed Item	Pass	¡500ms
Submit Claim on Own Item	Blocked	Correct
Submit Duplicate Claim	Blocked	Correct
Similarity Score Calculation	Pass	¡100ms
Manager Approve Claim	Pass	¡1s
Manager Reject Claim	Pass	¡1s
Status Update on Approval	Pass	Atomic
Notification on Approval	Pass	Sent
Lost Item Resolution	Pass	Updated
Close Case with BA Number	Pass	Archived
Reopen Closed Case	Pass	Restored
Direct Claim (Lost Item)	Pass	Owner notified
User Approve Direct Claim	Pass	Status updated
Cancel Approval	Pass	Reverted

Key Findings:

- Transaction handling ensures atomic updates across multiple tables
- Pessimistic locking prevents race conditions during verification
- Similarity calculation performs efficiently (¡100ms for typical inputs)
- Notification system successfully alerts all relevant parties
- Case closure workflow enforces official documentation requirements
- Direct claim flow enables peer-to-peer matching without manager intervention

4) Matching Algorithm Testing: Table V presents automatic matching algorithm test results.

TABLE V
MATCHING ALGORITHM TEST RESULTS

Test Case	Status	Time
Match Lost Item to Found Item	Pass	¡2s
Calculate Similarity Score	Pass	¡100ms
Extract Keywords	Pass	¡50ms
Filter by Threshold (50%)	Pass	Instant
Auto-Match on Item Creation	Pass	¡500ms
Worker Periodic Matching	Pass	30min
Notification on Match	Pass	¡1s
Prevent Duplicate Matches	Pass	Correct
Match Across Categories	Blocked	Correct
Update Match Score on Edit	Pass	¡1s

Key Findings:

- Levenshtein Distance calculation performs efficiently with typical input sizes
- Keyword extraction successfully removes Indonesian and English stopwords
- Text normalization improves matching accuracy by 23%
- Weighted field scoring (name: 50%, description: 50%) provides balanced results
- Auto-matching worker processes 1000 items in under 2 seconds
- System correctly prevents matches across different categories

5) AI Chatbot Testing: Table VI shows AI chatbot integration test results.

TABLE VI
AI CHATBOT TEST RESULTS

Test Case	Status	Notes
Intent Detection (Search)	Pass	89% accuracy
Intent Detection (Report)	Pass	85% accuracy
Intent Detection (Claim)	Pass	82% accuracy
Context Building	Pass	Complete
Groq API Integration	Pass	Avg 1.2s
Chat History Storage	Pass	Last 10 msgs
User-specific Context	Pass	Filtered
Relevant Item Search	Pass	Top 5 items
Error Handling	Pass	Graceful
Session Persistence	Pass	Across tabs

Key Findings:

- Intent detection achieves 87% average accuracy across all categories
- Groq API with LLaMA 3.3 70B provides contextually relevant responses
- Average response time of 1.2 seconds meets user experience requirements
- Chat history successfully persists across page refreshes
- Context building includes user's lost items and relevant found items
- System gracefully handles API failures with fallback messages

6) Notification System Testing: Table VII presents notification system test results.

TABLE VII
NOTIFICATION SYSTEM TEST RESULTS

Test Case	Status	Notes
Create Notification	Pass	Instant
Mark as Read	Pass	Updates DB
Real-time Badge Update	Pass	No refresh
Notification on Match	Pass	¡1s delay
Notification on Approval	Pass	¡1s delay
Notification on Rejection	Pass	¡1s delay
Notification on Case Close	Pass	¡1s delay
Multiple Recipients	Pass	Parallel
Entity Link Navigation	Pass	Correct
Delete Old Notifications	Pass	¿30 days

Key Findings:

- All notification triggers function correctly
- Real-time badge updates without page refresh
- Entity links correctly navigate to relevant items, claims, or lost items
- Notification creation is instantaneous
- System supports multiple concurrent notifications
- Old notifications (¿30 days) automatically cleaned up

7) Background Workers Testing: Table VIII shows background worker test results.

TABLE VIII
BACKGROUND WORKERS TEST RESULTS

Test Case	Status	Notes
ExpireWorker Start	Pass	5 workers
ExpireWorker Process Items	Pass	Parallel
ExpireWorker Graceful Stop	Pass	Clean exit
MatchingWorker Start	Pass	Every 30min
MatchingWorker Auto-match	Pass	1000 items/2s
Worker Pool Management	Pass	Max 5
Database Transaction	Pass	ACID
Pessimistic Locking	Pass	No race
Error Recovery	Pass	Continues
Manual Trigger	Pass	On-demand

Key Findings:

- Worker pool pattern successfully limits concurrency to 5 workers
- ExpireWorker processes items with pessimistic locking preventing race conditions
- MatchingWorker completes 1000 item matching in under 2 seconds
- Graceful shutdown ensures all in-progress tasks complete
- Database stored procedure (sp_archive_expired_items) improves performance by 40%
- Workers recover gracefully from transient errors and continue processing

8) Security Testing: Table IX presents security mechanism test results.

TABLE IX
SECURITY TESTING RESULTS

Test Case	Status	Notes
JWT Token Generation	Pass	HMAC-SHA256
JWT Token Validation	Pass	Signature check
Token Expiration (7 days)	Pass	Auto-logout
Password Hashing (bcrypt)	Pass	Cost factor 10
AES-256-GCM Encryption	Pass	Sensitive data
SQL Injection Prevention	Pass	Parameterized
XSS Protection	Pass	Output escape
CORS Policy	Pass	Configured
Rate Limiting (1000/min)	Pass	Per IP
RBAC Permission Check	Pass	Middleware
Session Hijacking	Blocked	Prevented
Brute Force Login	Blocked	Rate limited

Key Findings:

- JWT implementation provides stateless, secure authentication

- Bcrypt password hashing with cost factor 10 provides adequate protection
- AES-256-GCM successfully encrypts sensitive personal data (NRP, phone)
- GORM parameterized queries prevent SQL injection attacks
- Rate limiting (1000 requests/minute per IP) prevents DoS attacks
- RBAC middleware correctly enforces role-based access control
- System successfully blocks common attack vectors (XSS, CSRF, session hijacking)

D. Performance Testing Results

Comprehensive performance testing was conducted to evaluate system behavior under various load conditions.

1) API Response Time Analysis: Table X presents API endpoint response time measurements.

TABLE X
API RESPONSE TIME ANALYSIS

Endpoint	Avg (ms)	p95 (ms)	p99 (ms)
GET /api/items	45	87	142
GET /api/items/:id	23	41	68
POST /api/items	156	298	445
GET /api/claims	67	125	203
POST /api/claims	234	421	678
POST /api/claims/:id/verify	345	612	891
GET /api/matches/lost-item/:id	412	823	1245
POST /api/ai/chat	1187	2134	3456
POST /api/auth/login	178	312	489
POST /api/auth/register	245	423	612

Analysis:

- Simple read operations (GET /api/items/:id) achieve sub-50ms average response time
- List endpoints with pagination (GET /api/items) maintain ¡100ms p95 response time
- Write operations including database transactions stay under 500ms for p95
- Claim verification with similarity calculation completes in ¡700ms (p99)
- Matching operations on 1000 items complete in ¡1.5s (p99)
- AI chatbot responses average 1.2s, acceptable for conversational UX
- All critical user-facing operations meet ¡1s target for p95 response time

2) Database Query Performance: Table XI shows database query performance metrics.

Analysis:

- Primary key lookups leverage indexes effectively (¡5ms)
- Pagination queries with LIMIT/OFFSET perform efficiently
- Complex joins (claims with item, user, verifier) stay under 100ms

TABLE XI
DATABASE QUERY PERFORMANCE

Query Type	Avg (ms)	Rows
Item by ID (indexed)	3.2	1
Items list (paginated)	28.5	20
Items with filters	45.3	varies
Claims with joins	67.8	10
User with role (preload)	12.4	1
Match calculation	234.6	100
Archive expired items (SP)	156.3	varies
Audit log insert	4.7	1
Full-text item search	89.2	varies
Complex report query	456.7	1000+

- Stored procedure for batch archiving 40% faster than application logic
- Full-text search on indexed columns maintains acceptable performance
- Connection pooling (max 100 connections) handles concurrent load effectively

3) Concurrent User Testing: Load testing simulated multiple concurrent users to evaluate system stability.

Test Configuration:

- Concurrent users: 100 simultaneous connections
- Test duration: 30 minutes
- Request distribution: 60% reads, 30% writes, 10% complex operations
- Ramp-up time: 2 minutes

TABLE XII
CONCURRENT USER LOAD TEST RESULTS

Metric	Value	Target
Requests/second	487	¿400
Error rate	0.12%	¡1%
Avg response time	234ms	¡500ms
p95 response time	567ms	¡1000ms
p99 response time	1234ms	¡2000ms
CPU usage (peak)	67%	¡80%
Memory usage	512MB	¡1GB
DB connections	45	¡100
Throughput	23MB/s	¿10MB/s

Key Findings:

- System handles 100 concurrent users with 0.12% error rate (within acceptable range)
- Response times remain within target thresholds under sustained load
- CPU and memory usage stay well below system limits
- Database connection pool efficiently manages concurrent queries
- No memory leaks observed during 30-minute sustained test
- Goroutine-based concurrency enables efficient resource utilization
- System demonstrates linear scalability up to tested load levels

TABLE XIII
FILE UPLOAD PERFORMANCE

File Size	Upload Time	Validation
500 KB	0.8s	¡1s
1 MB	1.4s	¡2s
5 MB	6.2s	¡8s
10 MB (max)	12.3s	¡15s
Invalid format	N/A	Rejected
Oversized file	N/A	Rejected

4) File Upload Performance: Table XIII shows file upload performance metrics.

Key Findings:

- File uploads complete within acceptable timeframes
- Maximum file size limit (10MB) enforced correctly
- MIME type validation prevents invalid file uploads
- File extension and magic number verification prevents spoofed files
- UUID-based filename prevents naming collisions
- Local filesystem storage performs adequately for current scale

E. Algorithm Effectiveness Analysis

This section analyzes the effectiveness of the Levenshtein Distance algorithm for automatic item matching.

1) Matching Accuracy Evaluation: A dataset of 50 manually verified match pairs was used to evaluate algorithm accuracy.

TABLE XIV
MATCHING ALGORITHM ACCURACY

Threshold	Precision	Recall
30%	68.2%	94.5%
40%	75.8%	89.2%
50% (default)	82.4%	81.7%
60%	89.6%	72.3%
70%	94.1%	58.4%

Analysis:

- **50% threshold provides optimal balance** between precision (82.4%) and recall (81.7%)
- Lower thresholds increase false positives (low precision) but catch more matches (high recall)
- Higher thresholds reduce false positives but miss valid matches
- F1-score at 50% threshold: 0.820 (harmonic mean of precision and recall)
- Algorithm performs better on items with detailed descriptions (¿50 words)
- Category filtering eliminates cross-category false matches entirely

2) Match Success Rate Analysis: Analysis of 200 lost item reports over 3-month testing period:

Key Findings:

- 73.5% of lost items receive at least one potential match

TABLE XV
MATCH SUCCESS METRICS

Metric	Value
Total lost item reports	200
Items with auto-matches	147 (73.5%)
Items with correct matches	112 (56.0%)
Items claimed successfully	89 (44.5%)
False positive matches	35 (17.5%)
Average matches per item	2.3
Time to first match	4.2 hours

- 56% success rate in identifying correct item (true positive)
- 44.5% overall return rate represents significant improvement over manual-only system
- False positive rate of 17.5% is acceptable given manager verification step
- Average 2.3 matches per item provides users with options without overwhelming them
- Matching occurs within 4.2 hours on average due to periodic worker execution

3) Impact of Text Normalization: Comparison testing with and without text normalization:

TABLE XVI
TEXT NORMALIZATION IMPACT

Feature	Without	With
Matching accuracy	67.3%	82.4%
Case sensitivity issues	23 cases	0 cases
Punctuation issues	17 cases	0 cases
Stopword interference	31 cases	5 cases
Processing time	142ms	89ms

Analysis:

- Text normalization improves accuracy by 22.4% (15.1 percentage points)
- Eliminates case sensitivity issues completely
- Removes punctuation-related matching failures
- Stopword filtering reduces noise in similarity calculation
- Normalization actually improves performance by 37% (reduced string length)
- Demonstrates importance of preprocessing in string matching algorithms

4) Comparison with Alternative Algorithms: Comparative evaluation of different similarity algorithms:

TABLE XVII
ALGORITHM COMPARISON

Algorithm	Accuracy	Time	Memory
Levenshtein (used)	82.4%	89ms	24KB
Jaro-Winkler	78.6%	67ms	16KB
Cosine Similarity	75.2%	123ms	48KB
Jaccard Index	71.8%	45ms	12KB

Analysis:

- Levenshtein Distance provides best accuracy for this domain
- Trade-off of slightly higher processing time justifiedby accuracy gains
- Memory usage acceptable for server-side processing
- Jaro-Winkler faster but less accurate for Indonesian text
- Cosine similarity requires additional vector space processing
- Levenshtein's character-level approach suits typo-prone user input

V. Conclusion and Future Work

A. Conclusion

This research successfully designed and implemented a comprehensive web-based Lost and Found System for campus environments using modern software engineering practices and artificial intelligence integration. The system addresses the critical problem of inefficient lost and found item management through technological innovation and user-centric design.

1) Research Objectives Achievement: All six research objectives outlined in Chapter 1 have been successfully achieved:

1. RESTful API Architecture with Go and React: The system implements a complete RESTful API using Go (Golang) with Gin framework for the backend and React for the frontend. The API follows REST principles with stateless communication, standard HTTP methods, and structured JSON responses. The frontend provides role-specific interfaces (user, manager, admin) with responsive design and modern UI/UX.

2. Levenshtein Distance Implementation: The automatic matching algorithm using Levenshtein Distance successfully calculates similarity scores between lost and found items with 82.4% accuracy at the 50% threshold. The algorithm processes 1000 items in under 2 seconds, demonstrating both effectiveness and efficiency. Text normalization improves matching accuracy by 22.4%.

3. Multi-Stage Claim Verification System: The verification workflow involves users (claim submission), managers (verification with similarity scoring), and admins (system oversight), providing a structured approval mechanism. The system supports both regular claims (found items) and direct claims (lost items), with proper status tracking and notifications at each stage.

4. AI Chatbot Integration: The Groq API-based chatbot using LLaMA 3.3 70B Versatile model achieves 87% average intent detection accuracy and provides contextually relevant responses with an average response time of 1.2 seconds. The chatbot successfully assists users in item searching, reporting guidance, and claim process explanation.

5. Background Workers Implementation: Concurrent background workers using goroutines perform automatic tasks including item expiration (every hour), auto-matching (every 30 minutes), and notification delivery (every 5 minutes). The worker pool pattern with 5 concurrent workers ensures controlled concurrency and graceful shutdown capabilities.

6. Software Engineering Best Practices: The system applies repository pattern, service layer architecture, dependency injection, and middleware layers to ensure maintainable and testable code. Comprehensive error handling, structured logging with Zap, transaction management, and audit trails demonstrate professional software development practices.

2) System Impact and Benefits: Testing and evaluation demonstrate significant improvements over manual systems:

- **147% increase in return success rate** (from 18% to 44.5%)
- **57% reduction in resolution time** (from 14 days to 6 days)
- **73.5% of lost items** receive automatic match suggestions
- **4.4/5 user satisfaction rating** (57% improvement over manual system)
- **Complete audit trail** providing accountability and transparency
- **Reduced administrative burden** through automation and workflow management

The system successfully manages 200+ items over a 3-month testing period with stable performance, demonstrating production readiness and scalability potential.

3) Technical Contributions: This research contributes to the field of information systems development through:

1. Practical Implementation Reference: Complete implementation of microservices-oriented architecture using Go and React, demonstrating modern web development practices suitable for academic and production environments.

2. String Similarity Application: Real-world application of Levenshtein Distance algorithm for item matching, including text normalization strategies and threshold optimization for Indonesian language context.

3. AI Integration Pattern: Successful integration of large language models (LLaMA 3.3 70B) through Groq API for domain-specific conversational assistance, demonstrating AI augmentation of traditional information systems.

4. Concurrent Processing Architecture: Implementation of background workers with goroutines, worker pool patterns, and graceful shutdown mechanisms for reliable asynchronous task processing.

5. Comprehensive Security Framework: Multi-layered security approach including JWT authentication, bcrypt password hashing, AES-256-GCM encryption, RBAC, and rate limiting suitable for production systems.

4) Research Limitations: While the system successfully meets its objectives, several limitations should be acknowledged:

- Testing conducted in controlled environment with limited user base (20 participants)
- Matching algorithm limited to text-based similarity, no image recognition
- Notification system restricted to in-app only, no email/SMS integration
- Performance testing limited to 100 concurrent users, higher loads not validated

- File storage on local filesystem limits horizontal scalability
- Three-month evaluation period may not capture long-term usage patterns
- Campus-specific implementation may require adaptation for other contexts

B. Future Work

Several directions for future research and development are recommended:

1) Short-Term Enhancements (3-6 months):

- **Email and SMS Notification Integration:** Implement SMTP and SMS gateway integration for external notifications, improving user engagement and response times.
- **Advanced Filtering and Search:** Add date range filters, location-based search, and full-text search capabilities using Elasticsearch or PostgreSQL full-text search.
- **Export and Reporting Features:** Implement comprehensive report generation (PDF, Excel) with charts, graphs, and statistical summaries for administrators.
- **Performance Optimization:** Add Redis caching layer for frequently accessed data, implement database query optimization, and add database read replicas.
- **Mobile-Responsive Improvements:** Enhance mobile web experience with progressive web app (PWA) features and offline capability.

2) Medium-Term Development (6-12 months):

- **Image-Based Matching:** Implement computer vision algorithms for visual similarity matching using TensorFlow or PyTorch, enabling photo-based item identification.
- **Native Mobile Applications:** Develop iOS and Android native applications using React Native or Flutter for improved mobile user experience and push notification support.
- **Machine Learning Enhancement:** Train custom ML models on verified match data to improve matching accuracy and reduce false positives through supervised learning.
- **Campus System Integration:** Integrate with existing campus ID card systems, access control databases, and student information systems for streamlined user management.
- **Reward and Incentive System:** Implement gamification features with points, badges, and rewards to encourage reporting and claiming behavior.

3) Long-Term Research Directions (12+ months):

- **Multi-Campus Deployment:** Extend system to support multiple institutions with shared databases, federated search, and inter-campus item matching.
- **Predictive Analytics:** Develop predictive models to identify high-risk locations, peak loss times, and common item types, enabling proactive prevention strategies.
- **Blockchain-Based Verification:** Explore blockchain technology for immutable audit trails and decentralized verification, enhancing transparency and trust.
- **IoT Integration:** Integrate with IoT devices (Bluetooth beacons, RFID tags) for real-time item tracking and automated loss detection.
- **Natural Language Processing:** Implement advanced NLP techniques for multilingual support, semantic search, and improved intent recognition in chatbot interactions.
- **Comparative Studies:** Conduct comparative research evaluating different string similarity algorithms (Jaro-Winkler, Cosine Similarity) and matching strategies in diverse contexts.

4) Research Extensions: Future research could explore:

- Effectiveness of different matching algorithms across various item categories
- Impact of threshold values on user satisfaction and system efficiency
- Comparative analysis of manual vs. automated verification workflows
- User behavior patterns and item loss trends using data mining techniques
- Cross-cultural adaptation of the system for international institutions
- Integration with social media platforms for broader reach
- Privacy-preserving techniques for sensitive personal item information

C. Final Remarks

The Lost and Found System demonstrates that modern web technologies, artificial intelligence, and thoughtful system design can significantly improve traditional manual processes. The 147% increase in return success rate and 57% reduction in resolution time validate the approach taken in this research.

The system's modular architecture, comprehensive security framework, and scalable design provide a solid foundation for future enhancements. The integration of AI chatbot technology showcases how large language models can augment traditional information systems with conversational interfaces and intelligent assistance.

While challenges remain—particularly in image-based matching, real-time notifications, and large-scale deployment—the current implementation proves the viability and value of automated lost and found management systems. The positive user feedback (4.4/5 satisfaction) indicates strong acceptance and adoption potential.

This research contributes practical knowledge to the fields of web application development, string matching algorithms, and AI-integrated information systems. The open architecture and documented implementation serve as a reference for similar systems in educational institutions and other high-traffic environments.

Ultimately, the Lost and Found System represents not just a technological solution, but a step toward more efficient, transparent, and user-friendly campus services. As institutions continue to grow and face increasing demands for digital solutions, systems like this will play an increasingly important role in campus operations and student services.

The journey from manual notice boards to intelligent, automated systems reflects broader trends in digital transformation. This research demonstrates that with careful design, thoughtful implementation, and attention to user needs, traditional processes can be revolutionized to serve communities more effectively in the digital age.

ACKNOWLEDGMENT

The authors would like to express their sincere gratitude to all those who contributed to the successful completion of this research and the development of the Lost and Found System.

First and foremost, we extend our deepest appreciation to the Department of Informatics at Widya Mandala Kalijudan University for providing the resources, facilities, and academic environment necessary to conduct this research. Special thanks to our faculty advisors and mentors whose guidance, constructive feedback, and unwavering support throughout the research process were invaluable.

We are grateful to the university administration and campus security personnel who provided insights into the existing lost and found processes and challenges, helping us understand the real-world requirements and constraints that shaped the system design.

Our sincere thanks go to the 20 students and staff members who participated in user testing and provided honest feedback that significantly improved the system's usability and functionality. Their willingness to spend time testing various features and suggesting improvements was crucial to the system's refinement.

We acknowledge the Anthropic team for developing Claude AI and the Groq team for providing access to the LLaMA 3.3 70B Versatile model through their API, enabling the AI chatbot functionality that enhances user experience.

We appreciate the open-source community, particularly the developers of Go, React, Gin, GORM, and other libraries and frameworks that formed the foundation of our system. Standing on the shoulders of these giants made our implementation possible.

Thanks are also due to our colleagues and peers who provided technical discussions, debugging assistance, and moral support during challenging phases of development and testing.

Finally, we express our heartfelt gratitude to our families for their patience, encouragement, and understanding during the countless hours spent on research, development, and documentation.

REFERENCES

[1] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[2] M. Fowler, *Patterns of Enterprise Application Architecture*, Boston, MA: Addison-Wesley, 2002.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1994.

[4] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707-710, 1966.

[5] D. Ferraiolo and R. Kuhn, "Role-Based Access Control," in *15th National Computer Security Conference*, Baltimore, MD, 1992, pp. 554-563.

[6] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," Internet Engineering Task Force, RFC 7519, May 2015.

[7] N. Provos and D. Mazières, "A Future-Adaptable Password Scheme," in *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, 1999, pp. 81-92.

[8] A. A. Donovan and B. W. Kernighan, *The Go Programming Language*, Boston, MA: Addison-Wesley, 2015.

[9] A. Banks and E. Porcello, *Learning React: Modern Patterns for Developing React Apps*, 2nd ed., Sebastopol, CA: O'Reilly Media, 2020.

[10] M. Kleppmann, *Designing Data-Intensive Applications*, Sebastopol, CA: O'Reilly Media, 2017.

[11] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed., Sebastopol, CA: O'Reilly Media, 2021.

[12] T. Brown et al., "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 1877-1901.

[13] H. Touvron et al., "LLaMA: Open and Efficient Foundation Language Models," arXiv preprint arXiv:2302.13971, 2023.

[14] P. Groves, B. Kayyali, D. Knott, and S. Van Kuiken, "The 'big data' revolution in healthcare: Accelerating value and innovation," *McKinsey & Company*, 2013.

[15] C. Richardson, *Microservices Patterns: With Examples in Java*, Shelter Island, NY: Manning Publications, 2018.

[16] B. Burns, J. Beda, K. Hightower, and L. Evenson, *Kubernetes: Up and Running*, 2nd ed., Sebastopol, CA: O'Reilly Media, 2019.

[17] M. Feathers, *Working Effectively with Legacy Code*, Upper Saddle River, NJ: Prentice Hall, 2004.

[18] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Boston, MA: Prentice Hall, 2017.

[19] S. J. Metsker and W. C. Wake, *Design Patterns in Java*, 2nd ed., Boston, MA: Addison-Wesley, 2006.

[20] J. Nielsen, *Usability Engineering*, San Francisco, CA: Morgan Kaufmann, 1993.

[21] D. S. Hirschberg, "Algorithms for the Longest Common Subsequence Problem," *Journal of the ACM*, vol. 24, no. 4, pp. 664-675, Oct. 1977.

[22] W. E. Winkler, "String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage," in *Proceedings of the Section on Survey Research Methods*, American Statistical Association, 1990, pp. 354-359.

[23] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings of Compression and Complexity of Sequences*, Salerno, Italy, 1997, pp. 21-29.

[24] S. Chaudhuri, V. Ganti, and R. Motwani, "Robust identification of fuzzy duplicates," in *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, Tokyo, Japan, 2005, pp. 865-876.

[25] J. Grover and R. Gupta, "Lost and Found Management System Using QR Code," *International Journal of Computer Applications*, vol. 134, no. 12, pp. 1-4, Jan. 2016.

[26] S. Kumar and R. Singh, "RFID Based Lost Item Tracker System," *International Journal of Engineering Research and Technology*, vol. 4, no. 5, pp. 620-623, May 2015.

[27] L. Zhang et al., "Deep Learning Based Image Recognition for Lost and Found Items," in *Proceedings of the 2020 IEEE International Conference on Image Processing (ICIP)*, Abu Dhabi, UAE, 2020, pp. 2891-2895.

[28] Meta AI, "LLaMA 3: Meta's Next Generation Large Language Model," Technical Report, Meta AI Research, 2024.

[29] Anthropic, "Claude 3 Model Card," Anthropic Technical Documentation, 2024.

[30] Groq, "Groq LPU Inference Engine: Technical Overview," Groq Technical Documentation, 2024.