Patchwork

Inna Gruneva, Jack Lund, Anna DeBarro, and Emily Mayer

Software Design Specification

1. System Overview

Patchwork is a web-based social media marketplace that bridges fashion inspiration sharing with peer-to-peer clothing sales. The system enables users to curate visual content on personalized boards, create item listings for clothing sales, and engage with a community of fashion enthusiasts through a following based social network.

The platform targets users interested in discovering fashion inspiration, building personal style brands, and buying/selling secondhand or unique clothing items. Users can make purchases, sell items, share content, build vision boards, and leave reviews all in one space with customizable privacy options such as posts, sales, and boards.

1.1 Core System Components

1. Authentication Service: Manages user registration, login, password security, JWT token generation, and password reset functionality
2. User Management Service: Handles user profiles, follow/unfollow relationships, follower/following lists, user search, and review management
3. Content Management Service: Processes post creation for both inspiration and item listings, image uploads, post editing/deletion, and like functionality
4. Board Management Service: Organizes user-created boards, manages privacy settings, and handles post assignments to boards
5. Feed Algorithm Service: Generates personalized content feeds based on following relationships with filtering capabilities (all/inspiration/listings)
6. Search Service: Provides user, post, and tag-based search with advanced filtering for marketplace listings
7. Data Persistence Layer: MongoDB database with collections for Users, Posts, Boards, and Sessions, plus external image storage service

These components interact through a three-tier web architecture: the Client Layer (web browser interface) communicates with the Application Layer (Node.js/Express server) via RESTful APIs, and the Application Layer interacts with the Data Layer (MongoDB) for persistence. The Authentication Services validates JWT tokens for all protected routes, while the Feed Algorithm Service queries both User Management (for following relationships) and Content Management (for posts) to generate personalized feeds.
The user interface also employs a sewing/quilting metaphor with ‘patching’ content to your ‘quilt’ board, including visual stitch lines connecting posts in profile layouts, and a sewing needle animation for loading states.

2. Software Architecture

2.1 Software Architecture

The Patchwork platform follows a three-tier web architecture with clear separation between presentation, application logic, and data persistence layers. This architecture supports scalability, maintainability, and future mobile application development.

Tier 1: Client Layer (Web Browser)

The Client Layer consists of six main interface components:

1. Authentication & Session Interface: Login/registration forms, password reset interface
2. Home Feed Display: Masonry-style grid with filter buttons and infinite scroll
3. Profile Management Interface: Three-tab layout (All Posts, Selling, Inspiration Boards)
4. Post Creation Interface: Forms for inspiration posts and item listings with image upload
5. Board Organization Interface: Board creation, post assignment, privacy controls
6. Search & Discovery Interface: User search, post search, tag exploration

Tier 2: Application Layer (Node.js/Express Server)

The Application Layer contains six service modules, each responsible for specific functionality:

1. Authentication Service
    1. Manages user registration with input validation
    2. Handles secure login with bcrypt password hashing (minimum 10 salt rounds)
    3. Generates and validates JWT tokens for session management
    4. Processes password reset requests with secure token generation
    5. Implements rate limiting to prevent brute force attacks
2. User Management Services
    1. Handles profile CRUD operations
    2. Manages follow/unfollow logic and relationship data
    3. Provides user search and discovery functionality
    4. Manages review system with rating calculations
3. Content Management Service
    1. Processes post creation for inspiration and listings
    2. Handles image upload validation and storage

    3. Manages post editing, deletion, and status updates
    4. Implements like/unlike functionality
    5. Handles tag indexing for search
4. Board Management Service
    1. Creates and organizes user boards
    2. Manages privacy settings (public/private boards)
    3. Handles post assignments to boards
5. Feed Algorithm Service
    1. Generates personalized content feeds based on following relationships
    2. Applies content filtering (all/inspiration/listings)
    3. Implements reverse-chronological sorting with pagination
    4. Provides recommended user suggestions
6. Search Service
    1. User search by username/name
    2. Post search with keyword matching
    3. Tag-based content discovery
    4. Advanced filtering for listings (price, category, size, condition)

Tier 3: Data Layer (MongoDB)

1. User Collection: Stores credentials, profiles, addresses, followers/following arrays, and reviews
2. Post Collection: Contains inspiration posts and item listings with metadata, tags, and likes
3. Board Collection: Manages user-created collections with post references
4. Session Collection: Maintain active sessions and JWT tokens
5. Image Storage Service: External service for uploaded images with URL references

Component Interactions

The Client Layer communicates with the Application Layer exclusively through RESTful API endpoints over HTTPS. All authenticated requests include JWT tokens in the Authorization header, which the Authentication Service validates before routing to other services.

Within the Application Layer, services interact through internal method calls. For example, the Feed Algorithm Service calls getUserFollowing() from the User Management Service to retrieve the user's following list, the calls getPostsbyUserIds() from the Content Management Service to fetch relevant posts. The Content Management Service interacts with the Image Storage Service for file uploads and URL retrieval.

The Application Layer interacts with the Data Layer through MongoDB's native driver using asynchronous queries. Services execute CRUD operations on collections and leverage MongoDB's aggregation pipeline for complex queries like feed generation.

Payment processing is handled exclusively through third-party services (PayPal or Stripe). The application never stores raw payment information; only encrypted payment service identifiers are saved in the Users collection. Transaction flows redirect users to secure payment portals hosted by the payment provider.

2.2 Naming Conventions

All module names are descriptive and project-specific, avoiding generic terms. Instead of Database or Backend, we use User Management Service and Content Management Service. The term client and server are used contextually to describe roles in specific service interactions, recognizing that a component can be both client and server depending on the operation.

2.3 Design Rationale

The three-tier architecture was selected for clear separation of concerns, enabling independent development and testing of each layer. This design facilitates horizontal scaling; the Application Layer can be deployed across multiple servers with load balancing, while the Data Layer can implement sharding as the user base grows.

MongoDB was chosen over relational databases because the platform's data model involves flexible, nested structures (boards containing post arrays, users with variable-length follower lists, posts with optional listing details). NoSQL's document-oriented approach naturally represents these relationships without complex JOIN operations, significantly improving query performance for feed generation and profile loading: the most frequent operations.

The modular service architecture within the Application Layer allows individual components to be updated independently. For instance, the Feed Algorithm Service can be enhanced with machine-learning based recommendations without modifying User Management or Content Management services. This decomposition follows the principle of high cohesion (related functionality grouped together) and low coupling (minimal dependencies between services).

JWT-based authentication supports stateless API design, essential for future mobile app development and horizontal scaling. JWTs can be validated cryptographically without database lookups, reducing latency for authenticated requests compared to traditional session-based authentication.

Image storage is externalized rather than stored as binary data in MongoDB to reduce document size, improve query performance, and enable efficient CDN integration for image delivery.

3. Software Modules

This section describes each major module of the Patchwork system. Each module

description includes its role, interface specifications, static model, dynamic model, and design rationale as required by IEEE Std 1016-2009.

3.1 Authentication Service

3.1.1 Module Role and Primary Function

The Authentication Service manages all user credential operations, session management, and security protocols, ensuring only authorized users access protected resources.

Primary Functions:

1. User Registration
    1. Input validation for email format, password strength, username uniqueness
    2. Password hashing using bcrypt with minimum 10 salt rounds
    3. User document creation in MongoDB Users collection
2. User Authentication
    1. Email/username and password verification
    2. JWT token generation with 24-hour expiration
    3. Token validation for subsequent requests
3. Password Management
    1. Password reset token generation and email delivery
    2. Secure password update with token validation
4. Security
    1. Rate limiting on login attempts (5 attempts per IP within 15 minutes)
    2. HTTPS enforcement for all authentication endpoints

3.1.2 Interface Specification

Public REST API Endpoints:

POST /api/auth/register

Input: { firstName: string, lastName: string, email: string, username: string, password: string }

Output: { userId: ObjectId, token: string } or { error: string }

POST /api/auth/login

Input: { emailOrUsername: string, password: string }

Output: { userId: ObjectId, token: string, userData: User } or { error: string }

POST /api/auth/logout

Input: Header: Authorization: Bearer <token>

Output: { success: boolean }

POST /api/auth/forgot-password

Input: { email: string }

Output: { success: boolean, message: string }

POST /api/auth/reset-password

Input: { resetToken: string, newPassword: string }

Output: { success: boolean }

GET /api/auth/validate-token

Input: Header: Authorization: Bearer <token>

Output: { valid: boolean, userId?: ObjectId }

Internal Service Methods:

hashPassword(password: string): Promise<string>

Generates bcrypt hash with salt rounds

comparePassword(plaintext: string, hash: string)

Promise<boolean>: Verifies password

generateJWT(userId: ObjectId): string

Creates signed JWT token with 24-hour expiration

verifyJWT(token: string): { userId: ObjectId, exp: number } | null

Validates and decodes JWT

3.1.3 Static Model

The Authentication Service maintains minimal state. Primary data structures:

JWT Payload Structure:

{ userId: ObjectId, iat: number (issued at timestamp), exp: number (expiration timestamp) }

Password Reset Token Structure:

{ resetToken: string (UUID v4), userId: ObjectId, expiresAt: Date (1 hour from creation) }

3.1.4 Dynamic Model

User Registration Sequence:

1. Client submits registration data to POST /api/auth/register
2. Service validates input format (email regex, password length ≥ 8 characters)
3. Service queries Users collection to verify email and username uniqueness
4. If unique, password is hashed using bcrypt.hash(password, 10)
5. New user document inserted into Users collection
6. JWT token generated with userId and 24-hour expiration
7. Token and userId returned to client
8. Client stores token in localStorage

Login Authentication Sequence:

9. Client submits credentials to POST /api/auth/login
10. Service queries Users collection by email or username
11. If user found, comparePassword() verifies submitted password against stored hash
12. If passwords match, JWT token generated
13. Token and user data (excluding passwordHash) returned to client
14. Client stores token for subsequent authenticated requests

3.1.5 Design Rationale

JWT tokens enable stateless authentication, supporting horizontal scaling and future mobile app development. Unlike session-based authentication requiring database lookups on every request, JWTs are validated cryptographically without additional queries. This design decision prioritizes scalability over immediate token revocation capability.

Bcrypt was selected for its adaptive cost factor; as computing power increases, the salt rounds can be increased to maintain security against brute force attacks. The 10-round minimum balances security with acceptable login response times (<500ms). Alternative hashing algorithms (SHA-256, MD5) were rejected as they are not designed for password hashing and lack salting mechanisms.

Rate limiting (5 attempts per IP within 15 minutes) prevents credential stuffing attacks while maintaining usability for legitimate users experiencing login difficulties. This threshold was chosen based on industry standards for consumer applications.

The separation of registration and login endpoints, while functionally overlapping in token generation, maintains clear API semantics and allows for different validation rules and error handling for each operation.

3.2 User Management Service

3.2.1 Module Role and Primary Function

The User Management Service handles all operations related to user profiles, social relationships (followers/following), and user discovery. This module maintains user data integrity and provides interfaces for profile customization and social network building.

Primary Functions:

1. Profile Management: Create, read, update profile information including bio, profile picture, and address
2. Social Relationships: Follow/unfollow users, retrieve follower and following lists
3. User Discovery: Search users by username or name
4. Review Management: Add reviews, retrieve reviews, calculate average ratings
5. Privacy Controls: Manage address visibility for marketplace transactions **3.2.2**

Interface Specification

Public REST API Endpoints:

GET /api/users/:userId	Retrieve user profile
PUT /api/users/:userId	Update user profile (authenticated, owner only)
POST /api/users/:userId/follow	Follow a user (authenticated)
DELETE /api/users/:userId/follow	Unfollow a user (authenticated)
GET /api/users/:userId/followers	Get follower list
GET /api/users/:userId/following	Get following list
GET /api/users/search?q=<query>	Search users by username/name
POST /api/users/:userId/reviews	Add a review (authenticated)
GET /api/users/:userId/reviews	Retrieve user reviews

Internal Service Methods:

getProfile(userId: ObjectId): Promise<UserProfile> Retrieves complete user data

updateProfile(userId: ObjectId, updates: Partial<UserProfile>): Promise<boolean>

addFollower(followerId: ObjectId, followeeId: ObjectId): Promise<boolean>

removeFollower(followerId: ObjectId, followeeId: ObjectId): Promise<boolean>

getFollowerCount(userId: ObjectId): Promise<number>

getFollowingCount(userId: ObjectId): Promise<number>

calculateAverageRating(userId: ObjectId): Promise<number>

3.2.3 Dynamic Model

Follow User Sequence:

1. Client sends POST /api/users/:userId/follow with authentication token
2. Authentication Service validates JWT token
3. User Management Service checks if follow relationship already exists
4. If not, service adds followerId to target user's followers array
5. Service adds target userId to current user's following array
6. Both updates committed to MongoDB in a single operation
7. Success response returned to client

3.2.4 Design Rationale

Follower/following arrays are stored directly in the Users collection (denormalization) rather than a separate Relationships collection. This design improves read performance for profile loading and feed generation; the most frequent operations; by eliminating JOIN equivalent aggregations. While this creates data duplication (each following relationship

appears in two documents), the simplified queries and 2x faster read performance justify the tradeoff. The alternative of a normalized Relationships collection would require complex aggregations for every profile view.

Address information includes visibility controls: sellers display city/state publicly on listings, but full addresses are revealed only to buyers after purchase completion. This privacy-first approach protects user information while enabling transaction logistics. The module ensures that API responses filter address fields based on requester identity.

Reviews are embedded as subdocuments in the Users collection rather than a separate Reviews collection to optimize profile loading. This design assumes users will not accumulate thousands of reviews, making embedding acceptable. If review volume grows significantly, this design can be refactored to use references without affecting the API contract.

3.3 Content Management Service

3.3.1 Module Role and Primary Function

The Content Management Service handles all operations for creating, editing, and managing posts; both inspiration posts and item listings. This module processes image uploads, manages post metadata, and handles the post lifecycle.

3.3.2 Interface Specification

POST /api/posts Create new post (authenticated)

GET /api/posts/:postId Retrieve single post

PUT /api/posts/:postId Edit post (authenticated, owner only)

DELETE /api/posts/:postId Delete post (authenticated, owner only)

POST /api/posts/:postId/like Like a post (authenticated)

DELETE /api/posts/:postId/like Unlike a post (authenticated)

GET /api/posts/user/:userId?type=<all|inspiration|listings> Get user posts

PUT /api/posts/:postId/status Update listing status (authenticated, owner only)

3.3.3 Design Rationale

The Posts collection uses a polymorphic document structure where the type field (inspiration or listing) determines whether optional listingDetails are present. This allows both content types in a single collection, simplifying feed generation queries that combine inspiration and marketplace content. The alternative of separate collections would require UNION-equivalent operations and complicate the feed algorithm.

Images are stored externally with URLs saved in Posts documents rather than embedding binary data in MongoDB. This reduces document size (keeping under 16MB limit), improves query performance, and enables efficient CDN integration for image delivery. The image storage service handles resizing, optimization, and caching.

3.4 Board Management Service

3.4.1 Module Role and Primary Function

The Board Management Service enables users to organize posts into themed collections with privacy controls.

3.4.2 Interface Specification

POST /api/boards Create board (authenticated)

GET /api/boards/:boardId Retrieve board with posts

PUT /api/boards/:boardId Edit board (authenticated, owner only)

DELETE /api/boards/:boardId Delete board (authenticated, owner only)

POST /api/boards/:boardId/posts/:postId Add post to board

GET /api/boards/user/:userId Get user boards

3.4.3 Design Rationale

Boards store post IDs (references) rather than embedding full post documents. This prevents data duplication and ensures updates to posts (like listing status changes) are immediately reflected across all boards containing that post. The tradeoff is an additional query to populate post details when loading a board, which is acceptable using MongoDB's efficient aggregation pipeline.

3.5 Feed Algorithm Service

3.5.1 Module Role and Primary Function

The Feed Algorithm Service generates personalized content feeds based on following relationships with content type filtering.

3.5.2 Interface Specification

GET /api/feed?filter=<all|inspiration|listings>&limit=20&offset=0 (authenticated)

GET /api/feed/recommended-users?limit=10 (authenticated)

3.5.3 Dynamic Model

Feed Generation Sequence:

1. User requests feed with filter parameter
2. Service retrieves user's following list from Users collection
3. Service queries Posts collection for posts by followed users, sorted by createdAt descending
4. Content type filter applied if specified
5. Results paginated with limit/offset
6. Post data returned with pagination metadata

3.5.4 Design Rationale

The initial algorithm uses reverse-chronological sorting for simplicity and consistent performance. This approach ensures predictable load times and avoids cold-start problems with engagement-based ranking. The architecture supports future enhancement with machine learning-based personalization without requiring structural changes to other modules.

Content filtering is implemented at the query level rather than post-processing to minimize data transfer and improve response times. By applying type filters directly in MongoDB queries using the $match aggregation stage, the service retrieves only relevant posts rather than fetching all posts and filtering in application memory.

3.6 Search Service

3.6.1 Module Role and Primary Function

The Search Service provides comprehensive search capabilities across users, posts, and tags with advanced filtering for marketplace listings.

3.6.2 Interface Specification

GET /api/search/users?q=<query>

GET /api/search/posts?q=<query>&type=<all|inspiration|listings>
GET /api/search/tags?tag=<tagName>

GET /api/search/advanced?category=<cat>&minPrice=<n>&maxPrice=<n>&size=<size>

3.6.3 Design Rationale

MongoDB text indexes on username, caption, and itemName fields enable efficient full-text search without requiring external search infrastructure. This keeps the initial implementation simple while providing adequate search performance for the expected user base (under 100,000 users). As the platform scales beyond this threshold, the architecture supports migration to Elasticsearch or Algolia without affecting API contracts.

4. Alternative Designs

During the design process, several architectural alternatives were considered and evaluated against requirements for scalability, development timeline, and security. This section documents the major alternatives and explains why they were not selected.

4.1 Relational Database vs. MongoDB

Alternative Considered:
PostgreSQL with normalized relational schema using separate tables for Users, Posts, Boards, BoardPosts (junction), Followers (junction), and Reviews.

Evaluation:

Advantages:

Stronger data consistency guarantees with ACID transactions, established ORM tools (Sequelize, TypeORM), team familiarity with SQL, better support for complex analytical queries.

Disadvantages:

Complex JOIN operations required for feed generation (querying posts from multiple followed users with user data), rigid schema limits flexibility for evolving post types, slower write performance for high-frequency operations (likes, follows), difficulty representing variable-length arrays (followers, tags).

Decision:

MongoDB was selected for superior read performance in feed generation (3-5x faster in benchmarks) and flexibility to store heterogeneous post types in a single collection. The document-oriented model naturally represents nested structures like profiles with embedded addresses and arrays of followers. The tradeoff is weaker consistency guarantees, which is acceptable for a social media application where eventual consistency is sufficient.

4.2 Session-Based vs. Token-Based Authentication

Alternative Considered:

Traditional session-based authentication using Redis for server-side session storage with HTTP-only session cookies.

Evaluation:

Advantages:

Server-controlled session invalidation enables immediate logout and security breach response, smaller cookie size (just session ID vs. full JWT), protection against XSS attacks with HTTP-only cookies, ability to track active sessions.

Disadvantages:

Requires Redis infrastructure and maintenance, database lookup on every authenticated request adds latency, complicates horizontal scaling (requires session replication or sticky sessions), less suitable for future mobile apps (which prefer stateless APIs), single point of failure if Redis goes down.

Decision:

JWT-based authentication supports stateless API design, enabling horizontal scaling and future mobile app integration. The tradeoff of inability to immediately revoke tokens is mitigated by 24-hour expiration and is acceptable given the low-risk nature of the application (no financial transactions handled directly). If a security breach occurs, compromised accounts can be flagged in the database and checked during token validation.

4.3 Embedded vs. Referenced Post Data in Boards

Alternative Considered:

Embedding complete post documents within board documents instead of storing only post

IDs with references.

Evaluation:

Advantages:

Single query retrieves all board data including full post details, faster read performance for board viewing, no need for population/joins.

Disadvantages:

Data duplication causes consistency problems (e.g., listing status changes would require updating every board containing that post), rapidly growing document sizes as boards accumulate posts (average board has 50-200 posts), potential to exceed MongoDB's 16MB document size limit for power users with large boards, increased storage costs, complex update logic.

Decision:

Referenced design maintains data consistency and avoids document size limitations. The additional query to populate post details is acceptable using MongoDB's efficient $lookup aggregation pipeline, which adds only 50-100ms to board loading time. This design prioritizes data integrity over read performance optimization.

4.4 Microservices vs. Monolithic Architecture

Alternative Considered:

Decomposing the application into independent microservices (Authentication Service, User Service, Post Service, Feed Service, Search Service) running in separate containers with their own databases.

Evaluation:

Advantages:

Independent deployment and scaling of services, technology diversity (different services could use different databases), fault isolation, easier team division.

Disadvantages:

Significantly increased operational complexity, need for service mesh or API gateway, distributed transaction challenges, higher latency for cross-service communication, more difficult debugging, overkill for initial user base (<10,000 users).

Decision:

Monolithic architecture (with modular services) was chosen for initial launch to minimize operational complexity and development time. The modular service design within the monolith maintains logical separation and enables future migration to microservices if needed. This approach follows the principle of start monolithic, evolve to microservices.

5. Database Design

The Patchwork database uses MongoDB's document-oriented NoSQL model. While MongoDB does not enforce third normal form in the traditional relational sense, the schema is designed to minimize data redundancy and ensure logical data organization following NoSQL best practices and database normalization principles where applicable.

5.1 Collections Schema

Users Collection:

{ _id: ObjectId, username: String (unique, indexed), email: String (unique, indexed), passwordHash: String, firstName: String, lastName: String, profilePicture: String (URL), bio: String (max 500 chars), address: { street, city, state, zipCode, country }, paymentInfo: { paypalEmail: String (encrypted) }, followers: [ObjectId] (indexed), following: [ObjectId] (indexed), reviews: [{ rating: Number (1-5), reviewerId: ObjectId, comment: String, date: Date }], createdAt: Date, updatedAt: Date }

Posts Collection:

{ _id: ObjectId, userId: ObjectId (indexed), type: String (enum: inspiration | listing, indexed), images: [String] (URLs), caption: String (text indexed), tags: [String] (indexed), likes: [ObjectId], listingDetails: { itemName: String (text indexed), category: String (indexed), description: String, price: Number, size: String, condition: String, status: String (enum: available | sold | pending, indexed) }, boards: [ObjectId], createdAt: Date (indexed descending), updatedAt: Date }

Boards Collection:

{ _id: ObjectId, userId: ObjectId (indexed), name: String, description: String, isPublic: Boolean (indexed), posts: [ObjectId], coverImage: String (URL), createdAt: Date, updatedAt: Date }

Sessions Collection:

{ _id: ObjectId, userId: ObjectId (indexed), token: String (indexed), expiresAt: Date (TTL index for automatic deletion), createdAt: Date }

Index Strategy:

1. Users Collection
    1. username: unique index for login and uniqueness checks
    2. email: unique index for login and uniqueness checks
    3. followers: multi-key index for follower queries
    4. following: multi-key index for feed generation
2. Posts Collection
    1. userId: index for retrieving user's posts
    2. type: index for filtering by content type
    3. tags: multi-key index for tag-based discovery
    4. listingDetails.category: index for marketplace filtering

5. listingDetails.status: index for sold/available filtering
6. createdAt: descending index for chronological sorting in feeds
7. Text index: combined on caption and listingDetails.itemName for full-text search

3. Boards Collection
    1. userId: index for retrieving user's boards
    2. isPublic: index for filtering public/private boards
4. Sessions Collection
    1. userId: index for retrieving user sessions
    2. token: index for token validation
    3. expiresAt: TTL index (time-to-live) for automatic deletion of expired sessions

5.2 Normalization and Data Integrity

While MongoDB does not enforce relational normalization, the schema follows normalization principles where beneficial:

1. Atomic values: Each field contains indivisible data (e.g., email, username)
2. Referential integrity: Post userId references valid Users._id; Boards.posts[] references valid Posts._id
3. Denormalization exceptions: Follower/following arrays are intentionally denormalized for query performance; reviews are embedded for profile loading efficiency
4. Update anomalies minimized: Post data is referenced (not embedded) in Boards to prevent inconsistencies when posts are updated

5.3 Data Migration and Backup

Daily automated backups using MongoDB Atlas backup service with 7-day retention and point-in-time recovery for data loss scenarios. Migration strategy for schema evolution uses MongoDB's flexible schema; new fields are added incrementally, and application code handles both old and new document formats during transition periods. Deprecated fields remain until all documents are migrated using background batch jobs.

5.4 Scalability Considerations

Initial deployment uses a single MongoDB instance (replica set for high availability). As user base grows beyond 100,000 users, horizontal scaling will be implemented through:

1. Sharding on userId for Users and Posts collections to distribute data across multiple servers
2. Read replicas for geographically distributed users
3. Caching layer (Redis) for frequently accessed data (user profiles, feed results)
4. CDN integration for image delivery

6. References

IEEE Std 1016-2009. IEEE Standard for Information Technology; Systems Design; Software Design Descriptions. https://ieeexplore.ieee.org/document/5167255

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. Commun. ACM, 15(12), 1053-1058.

MongoDB Documentation. MongoDB Manual. https://docs.mongodb.com/manual/

Express.js Documentation. https://expressjs.com/

JWT Introduction. JSON Web Tokens. https://jwt.io/introduction

Bcrypt Documentation. https://www.npmjs.com/package/bcrypt

Class Diagram. In Wikipedia. https://en.wikipedia.org/wiki/Class_diagram
Sequence Diagram. In Wikipedia. https://en.wikipedia.org/wiki/Sequence_diagram

7. Acknowledgments

This document was created using the Software Design Specification template provided in the course materials. Technical references consulted include MongoDB documentation for NoSQL database design best practices, Express.js framework documentation for API design patterns, and IEEE software engineering standards for design documentation requirements.

Additional guidance on authentication security was obtained from OWASP (Open Web Application Security Project) recommendations for password hashing and token-based authentication.