MU Connect

Software Design Specifications for

Mobile application - Social Networking of MU students and alumni

<u>MU Connect</u>

Group Name: Prime Developers

Niyathi Vasasali	SE23LCSE005	se23lcse005@mahindrauniversity.edu.in
Varshini Babers	SE22UCSE320	se22ucse320@mahindrauniversity.edu.in
Sadhika Deva	SE22UCSE230	Se22ucse230@mahindrauniversity.edu.in
Divya Moduga	SE22UCSE084	Se23ucse084@mahindrauniversity.edu.in
Sathvick Hitesh	SE22UCSE111	Se22ucse111@mahindrauniversity.edu.in

Instructor:	*Dr. Vijay Rao Duddu*
Course:	Software Engineering
Lab Section:	*CS3 and CS4*
Teaching Assistant:	*Professor Nartakannai K*
Date:	07/04/2025

Document Information

Title: Software Design Specifications for MU Connect	
	Document Version No: 1
	Document Version Date: 08/04/2025
Prepared By: Our Team	Preparation Date: 02/04/2025

Version History

Ver. No.	Ver. Date	Revised by	Description	Filename
1.0	02/04/2025	Prime Developers	Initial Version of Document	Software Design Specifications Document
2.0	08/04/2025	Prime Developers	Final Version of Document	Software Design Specifications Document

Table of Contents

1 Introduction

1.1 Purpose

The purpose of this Software Design Document (SDD), is to provide a detailed architectural and design blueprint for MU Connect — a mobile application that facilitates social and professional networking between Mahindra University students and its alumni. It is intended for developers, testers, project managers, and university stakeholders to understand the design choices and technical structure of the application

1.2 Scope

This document covers the design of both the frontend and backend components, data models, UI/UX considerations, external interfaces, and quality attributes like scalability, security, and performance. MU Connect will support core functionalities such as user authentication, profile management, messaging, job postings, events, and forums

1.3 Definitions, Acronyms, and Abbreviations

- MU – Mahindra University
- SRS – Software Requirements Specification
- MVC – Model View Controller
- API – Application Programming Interface
- UI/UX – User Interface / User Experience
- JWT – JSON Web Token
- OAuth 2.0 – Authorization protocol for secure login

1.4 References

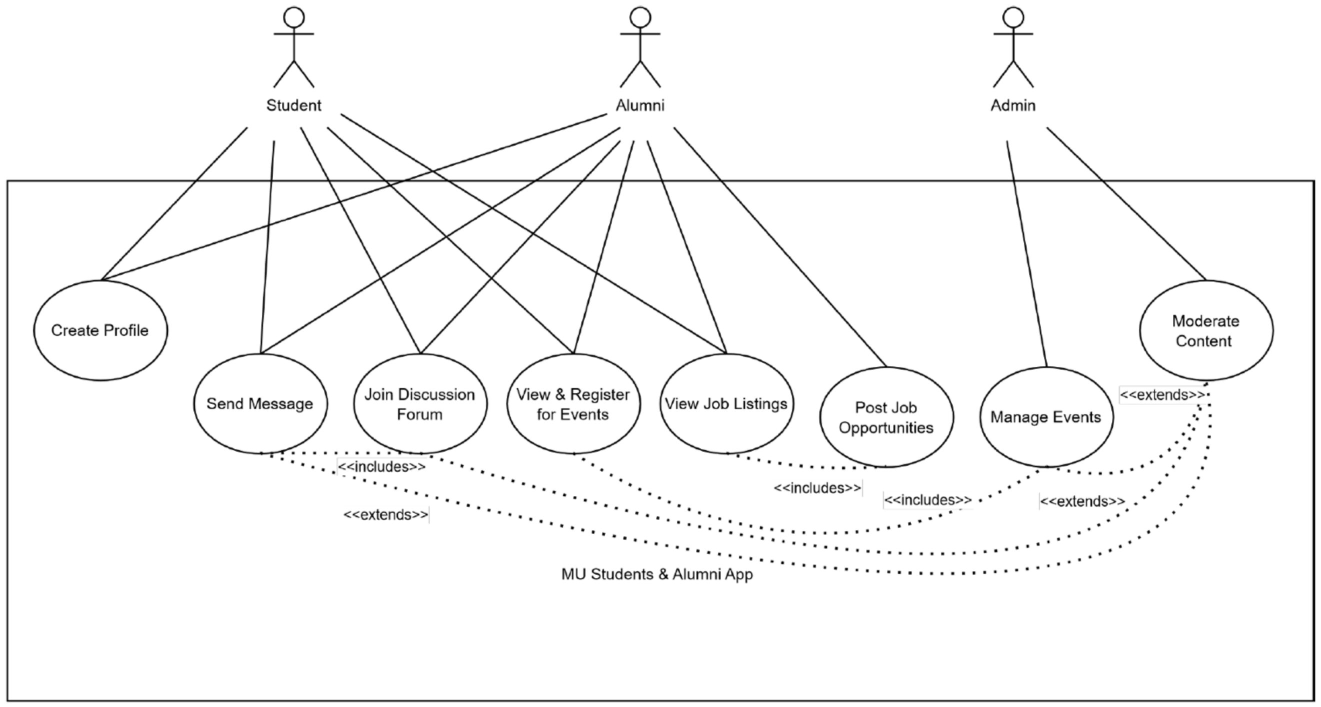
[This subsection should provide a complete list of all documents referenced elsewhere in the Software Design Specifications. Each document should be identified via an URL link to the source document library if available. If not, specify the sources from which the references can be obtained. This information may be provided by reference to an appendix or to another document.]

- Firebase Documentation
- PostgreSQL Documentation
- Node.js Documentation
- OAuth 2.0 Specification
- JWT Guide
- React Native Documentation
- AWS Architecture Best Practices

2 Use Case View

This section outlines the primary use cases that are central to the functionality and design of MU Connect. These use cases cover major features such as messaging, content posting, job opportunities, events, and profile management. They also span across multiple system layers — frontend UI, backend services, database access, and notifications.

Below is the system-level Use Case Diagram for MU Connect:



2.1 Use Cases

Use Case 1: Create Profile

- Actors: Student, Alumni
- Description: Enables users to input academic, professional, and personal information to create a visible profile within the app.
- Usage Steps:
  1. Navigate to Profile tab.
  2. Enter details: name, department, graduation year, interests, etc.

3. Upload a profile picture.
4. Save profile.

Use Case 2: Send Message

- Actors: Student, Alumni
- Description: Allows users to engage in one-on-one or group real-time messaging.
- Usage Steps:
    1. Select a user or group.
    2. Type a message.
    3. Hit send.
    4. Recipient receives notification (via Firebase or WebSockets).

Use Case 3: Join Discussion Forum

- Actors: Student, Alumni
- Description: Enables users to view and participate in themed academic or professional forums.
- Usage Steps:
    1. Enter a discussion topic.
    2. Read existing threads.
    3. Reply or create a new thread.
    4. Optionally subscribe to updates.

Use Case 4: View & Register for Events

- Actors: Student, Alumni
- Description: Lets users explore, RSVP for, and receive updates about university-hosted or alumni-organized events.

- Usage Steps:
    1. Browse the Events section.
    2. Select an event.
    3. Tap “Register” to RSVP.
    4. Receive calendar entry and reminders.

Use Case 5: View Job Listings

- Actors: Student, Alumni
- Description: Provides access to job opportunities shared by other alumni or the university.
- Usage Steps:
    1. Open “Jobs” tab.
    2. Browse job listings.
    3. Apply or bookmark opportunities.

Use Case 6: Post Job Opportunities

- Actors: Alumni
- Description: Allows alumni to post job openings for current students or fellow alumni.
- Usage Steps:
    1. Navigate to “Post Job” section.
    2. Fill in job details.
    3. Submit listing.

Use Case 7: Manage Events

- Actors: Admin

- Description: Admins create, update, or remove events from the system.
- Usage Steps:
    1. Log in with admin credentials.
    2. Access event dashboard.
    3. Add/edit/delete events.

Use Case 8: Moderate Content

- Actors: Admin
- Description: Enables moderators to flag or remove inappropriate posts, messages, or forum replies.
- Usage Steps:
    a. Review reported content.
    b. Take actions such as delete, warn, or suspend user.
    c. Log moderation actions.

3 Design Overview

The MU Connect app follows a multi-layered architecture aligned with high-level design requirements outlined in the SRS and diagrams. The system is split into four key layers:

Frontend Layer (Client-Side) – React Native/Flutter for mobile, React/Next.js for web

Backend Layer (Server-Side) – Node.js, Express, or Django

Database Layer – PostgreSQL, MongoDB, Firebase

Infrastructure Layer – Hosting (AWS/Firebase), CDN, CI/CD pipelines

This decomposition ensures separation of concerns, scalability, maintainability, and security. The system adheres to the Model-View-Controller (MVC) paradigm, supports RESTful API communication, and integrates real-time services using WebSockets or Firebase

3.1 Design Goals and Constraints

Goals:

- Ensure secure and role-based user interactions (Student, Alumni, Admin)
- Enable real-time communication (Chat, Notifications)
- Support job sharing, event participation, and forum discussions
- Maintain platform responsiveness and cross-device compatibility

Constraints:

- Must support both Android and iOS devices
- Use COMET methodology for object modeling
- Use PostgreSQL or MongoDB with scalable schema
- Adhere to Mahindra University's authentication policies
- Use OAuth 2.0 and JWT for login and session management
- Integrate cloud infrastructure like Firebase/AWS for hosting and notifications

3.2 Design Assumptions

- Users have active internet connections for real-time operations.
- Firebase/APNs are available for push notifications.
- Mobile users will have basic device capabilities (camera, media access).
- Database will support both structured and semi-structured content.
- The app will scale from 5–10 users in initial testing to 1000+ active users in future deployment.

3.3 Significant Design Packages

Layer	Package	Responsibilities
Frontend Layer	ui.react / mobile.flutter	Renders UI, handles input, sends API calls
Backend Layer	auth, chat, feed, forum, events, admin	Core business logic and user role handling
Database Layer	userDB, chatDB, eventDB, postsDB	Manages persistent user, message, and event data
Infrastructure Layer	notifications	Cloud hosting, build and deployment

3.4 Dependent External Interfaces

The table below lists the public interfaces this design requires from other modules or applications.

External Application and Interface Name	Module Using the Interface	Functionality/ Interface Description
Firebase Cloud Messaging	Notifications module	Sends push notifications(chat, events)
Oauth provider(ex. Google, MU login page)	Auth service	Secure login with verified email
AWS SES/ SendGrid	Email notification service	Send alerts, verification emails
Algolia/ Elasticsearch	Search service	Implements user and job search
APNs (IOS)	Notifications module	iOS-based push delivery

3.5 Implemented Application External Interfaces (and SOA web services)

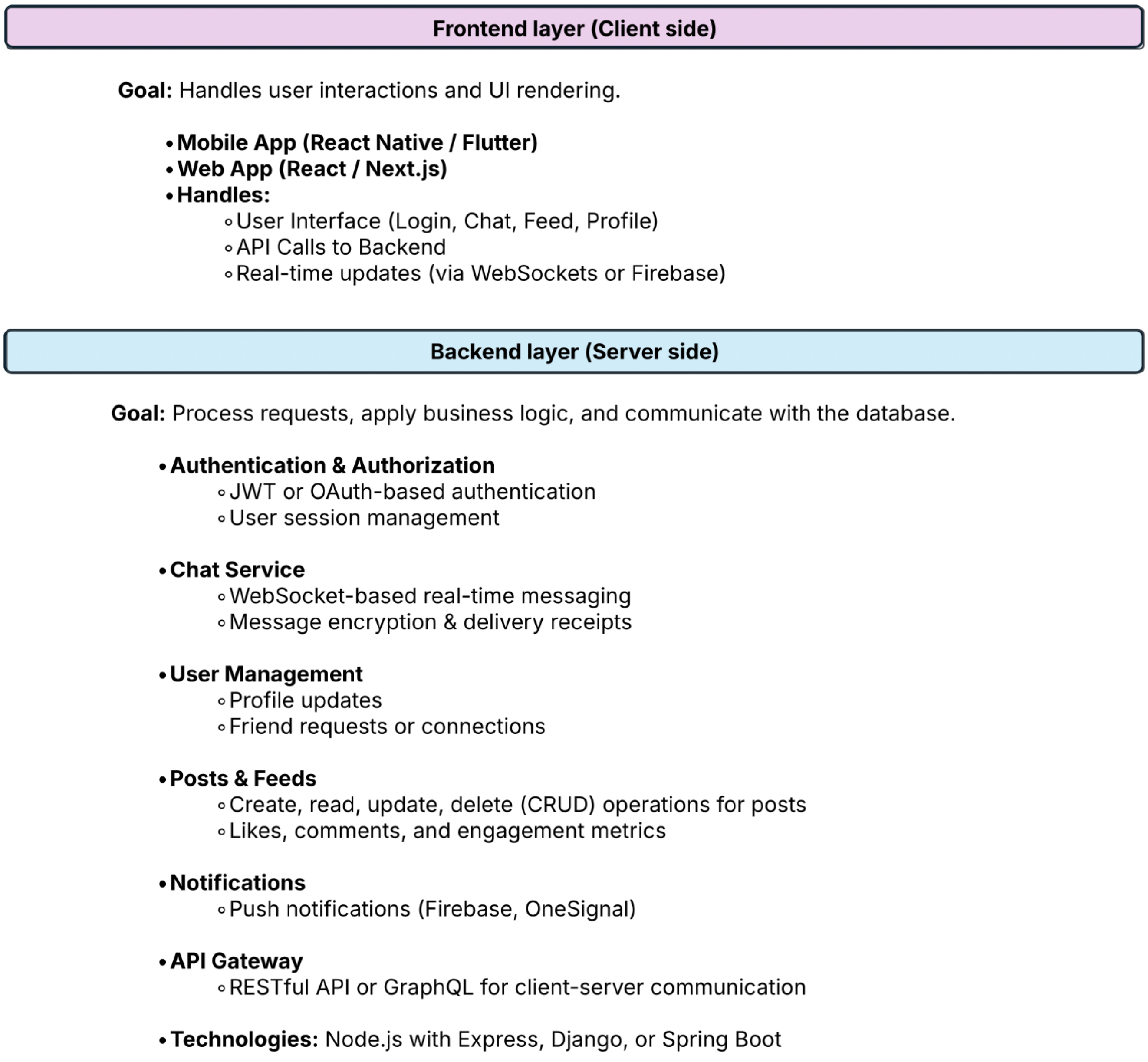
The table below lists the implementation of public interfaces this design makes available for other applications.

Interface Name	Module Implementing It	Functionality
/api/auth	Auth service	Login, logout, token generation
/api/users	Profile module	Profile creation, editing
/api/posts	Feed service	CRUD operations on posts
/api/messages	Chat service	WebSocket messaging
/api/events	Event manager	Event browsing and RSVP
/api/moderation	Admin service	Post moderation, user suspension

4 Logical View

The software architecture of MU Connect is layered and modular, with separation between the presentation, business logic, and data access layers. It uses the Model-View-Controller (MVC) pattern to organize frontend, backend, and data services. Below is a breakdown of the design model and how it supports the key use cases through structured module collaboration.

4.1 Design Model



Frontend layer (Client side)

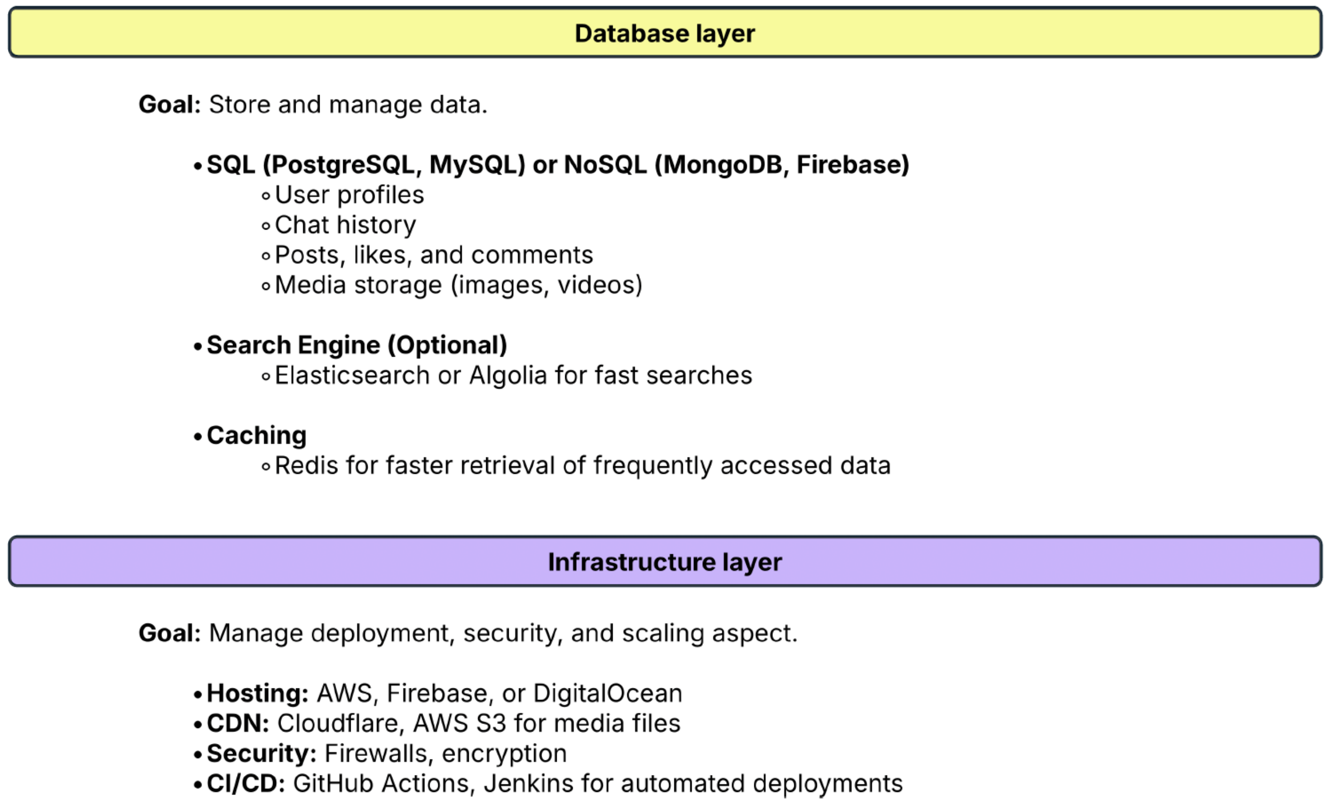
Goal: Handles user interactions and UI rendering.

- **Mobile App (React Native / Flutter)**
- **Web App (React / Next.js)**
- **Handles:**
    - User Interface (Login, Chat, Feed, Profile)
    - API Calls to Backend
    - Real-time updates (via WebSockets or Firebase)

Backend layer (Server side)

Goal: Process requests, apply business logic, and communicate with the database.

- **Authentication & Authorization**
    - JWT or OAuth-based authentication
    - User session management
- **Chat Service**
    - WebSocket-based real-time messaging
    - Message encryption & delivery receipts
- **User Management**
    - Profile updates
    - Friend requests or connections
- **Posts & Feeds**
    - Create, read, update, delete (CRUD) operations for posts
    - Likes, comments, and engagement metrics
- **Notifications**
    - Push notifications (Firebase, OneSignal)
- **API Gateway**
    - RESTful API or GraphQL for client-server communication
- **Technologies:** Node.js with Express, Django, or Spring Boot



Database layer

Goal: Store and manage data.

- **SQL (PostgreSQL, MySQL) or NoSQL (MongoDB, Firebase)**
  - User profiles
  - Chat history
  - Posts, likes, and comments
  - Media storage (images, videos)

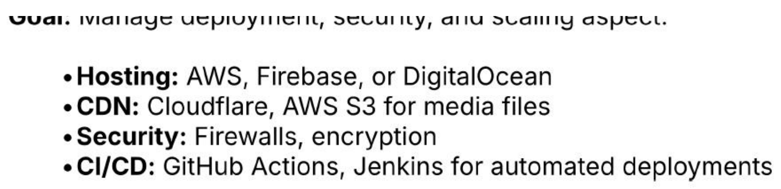
- **Search Engine (Optional)**
  - Elasticsearch or Algolia for fast searches

- **Caching**
  - Redis for faster retrieval of frequently accessed data

Infrastructure layer

Goal: Manage deployment, security, and scaling aspect.

- **Hosting:** AWS, Firebase, or DigitalOcean
- **CDN:** Cloudflare, AWS S3 for media files
- **Security:** Firewalls, encryption
- **CI/CD:** GitHub Actions, Jenkins for automated deployments



Goal: Manage deployment, security, and scaling aspect.

- **Hosting:** AWS, Firebase, or DigitalOcean
- **CDN:** Cloudflare, AWS S3 for media files
- **Security:** Firewalls, encryption
- **CI/CD:** GitHub Actions, Jenkins for automated deployments

Use Case: User Registration & Authentication

High-Level Flow:

1. AuthScreen collects credentials
2. Sends request to AuthController
3. Controller verifies via OAuth → issues JWT
4. JWT returned to client and stored for future requests

High level to low level sequence (module interaction):

[User] → AuthScreen → AuthController → OAuth Provider → AuthController → UserModel → AuthController → [Client]

Use Case: Send Message

High-Level Flow:

1. ChatScreen opens WebSocket connection

2. User sends message to MessageController via WebSocket
3. Message is stored in MessageModel
4. PushService triggers real-time notification

Module interaction:

[User] → ChatScreen → MessageController (WebSocket) → MessageModel → PushService → [Recipient]

Use Case: Post Job Opportunity

High-Level Flow:

1. Alumni opens JobBoardScreen
2. Enters job info, sends to PostController
3. Stored in PostModel
4. Appears in all users' feeds

Module interaction:

[Alumni] → JobBoardScreen → PostController → PostModel → [All Users]

Use Case: Moderate Content

High-Level Flow:

1. AdminPanel flags post via AdminController
2. PostModel queried for post status
3. Post is either hidden or deleted
4. Optional: User notified via PushService

Module interaction:

[Admin] → AdminPanel → AdminController → PostModel → PushService (optional)

5 Data View

[This section describes the persistent data storage perspective of the system. This section is may not apply if there is little or no persistent data, or the translation between the Design Model and the Domain Model is trivial]

5.1 Domain Model

[Provide the entity model that represents the persistent data and relationships between them. The entities in this model are typically domain objects or data transfer objects.]

Entities:

- User
  - ID, Name, Email, Role (Student/Alumni/Admin), GraduationYear, ProfilePicture
- Post
  - ID, AuthorID, Content, Timestamp, Likes, Comments
- Message
  - ID, SenderID, ReceiverID/GroupID, Content, Timestamp, ReadStatus
- Event
  - ID, Title, Description, DateTime, RSVPList
- Job
  - ID, PostedByID, JobTitle, Description, Location, ApplyLink
- ForumThread
  - ID, Topic, CreatedBy, Replies[]
- Notification
  - ID, RecipientID, Type, Message, Timestamp, Seen

Relationships:

- A User can have many Posts, Messages, Jobs, and Events.
- A ForumThread belongs to one User, and has many Replies.
- Notifications are tied to Users and system events.

5.2 Data Model (persistent data view)

5.2.1 Data Dictionary

User

- user_id: unique ID
- name: full name
- email: MU or verified email
- role: Student / Alumni / Admin
- graduation_year: year of graduation
- profile_picture: profile image URL

Post

- post_id: unique ID
- author_id: who created the post
- content: text of the post
- media_url: attached image/video (optional)
- timestamp: when it was posted
- likes_count: number of likes

- comments_count: number of comments

Message

- message_id: unique ID
- sender_id: who sent the message
- receiver_id: who received it
- content: message text
- timestamp: when it was sent
- is_read: read or not (true/false)

Event

- event_id: unique ID
- title: event name
- description: what it’s about
- datetime: date and time
- location: where it’s happening
- rsvp_list: who’s attending

Job

- job_id: unique ID
- posted_by: which alumni posted it
- job_title: title of the role
- description: job details
- location: where the job is
- apply_link: application URL

Forum Thread

- thread_id: unique ID
- topic: discussion topic
- created_by: who started it
- created_at: when it started
- replies: list of replies

Notification

- notification_id: unique ID
- recipient_id: who gets the notification
- type: chat, event, job, etc.
- message: notification text
- timestamp: when it was sent
- is_seen: seen or not (true/false)

6 Exception Handling

[This section should describe exceptions that are defined within the application, the circumstances in which they can be thrown and handled, how the exceptions are logged, and the expected follow-up action needed.]

AuthException

- Trigger: Wrong login info or expired session
- Action: Show login error, return 401

DatabaseWriteException

- Trigger: Failed to save or update data
- Action: Retry once, log the error

NotFoundException

- Trigger: Trying to fetch something that doesn't exist
- Action: Show "Not Found" message, return 404

UnauthorizedAccess

- Trigger: User tries to access something without permission
- Action: Show access denied message, return 403

WebSocketDisconnect

- Trigger: Chat connection breaks
- Action: Try reconnecting and show "disconnected" alert

FileUploadError

- Trigger: Media file too big or wrong format
- Action: Show error message, reject upload

ValidationError

- Trigger: User enters invalid data (like empty name or wrong format)
- Action: Show input error, block submission

7 Configurable Parameters

[This section may be more applicable to J2EE-based systems than Ruby on Rails applications. It should describe parameters used by the application that are configurable. For simple configuration, the name and definition/usage of each parameter should be provided. For complex configuration, the XML schema associated with the configuration should be provided. For each configurable parameter, indication should be made as to which parameters are dynamic (can be changed without restarting the application).]

JWT_EXPIRY_TIME
- What it does: Sets how long login tokens last
- Can change during runtime: Yes

MAX_UPLOAD_SIZE_MB
- What it does: Limits media upload size
- Can change during runtime: Yes

DATABASE_URL
- What it does: Database connection address
- Can change during runtime: No

REDIS_CACHE_TIMEOUT

- What it does: Time to keep data in cache
- Can change during runtime: Yes

FIREBASE_API_KEY

- What it does: Used for push notifications
- Can change during runtime: No

ENABLE_SEARCH

- What it does: Turns search feature (Algolia/Elasticsearch) on/off
- Can change during runtime: Yes

ALLOW_SIGNUPS

- What it does: Toggles new user registration
- Can change during runtime: Yes

8 Quality of Service

[This section describes aspects of the design related to application availability, security, performance, and monitoring and control in production.]

8.1 Availability

[This section should reference the availability business requirement for the application. It should highlight any aspects of the design specifically intended to support availability at the required level. It should describe any aspects of the design that can impact availability (mass update, data loading, housekeeping or periodic maintenance that requires downtime).]

- The app is designed to stay online almost all the time — 99.9% uptime.
- It can handle more users by automatically scaling up when needed.
- Updates are done without shutting things down — using a safe deployment method (Blue/Green).

8.2 Security and Authorization

[This section should reference the business requirements related to feature and data access security and authorization, and describe the authorization features and qualifier design intended to implement the requirements. This section should also describe any application specific design related to authorization outside of the Authorization framework, and how setup and management of the user access is exposed to those responsible for user administration.]

- Users log in securely using OAuth 2.0 (e.g., Google or MU login).
- Every session is protected with JWT tokens.
- Only the right users can access certain features (like Admin tools).
- Data is encrypted when stored and while moving between server and app.

8.3 Load and Performance Implications

[In the system requirements document for the application, load projections and performance requirements are provided. This section should describe the implications of load and performance requirements on the detailed design components (expected business transaction execution rate, message processing rate, database table growth projections, etc.). Sufficient detail should be provided to support the creation of load and performance test plans.]

- Chat and real-time features are built to support hundreds of users at once.
- The system uses caching (like Redis) to speed things up.
- Media like images and videos are served through a CDN, reducing load on the servers.

8.4 Monitoring and Control

[This section should describe controllable processes implemented by the application (message handlers, daemons). This section should also describe measurable values that the application will publish for monitoring from these processes.]

- The app is constantly monitored using tools like Firebase Console or AWS CloudWatch.
- If anything breaks, developers get alerts right away.
- Admins have tools to moderate content and manage users.