

# InstaLite Technical Report

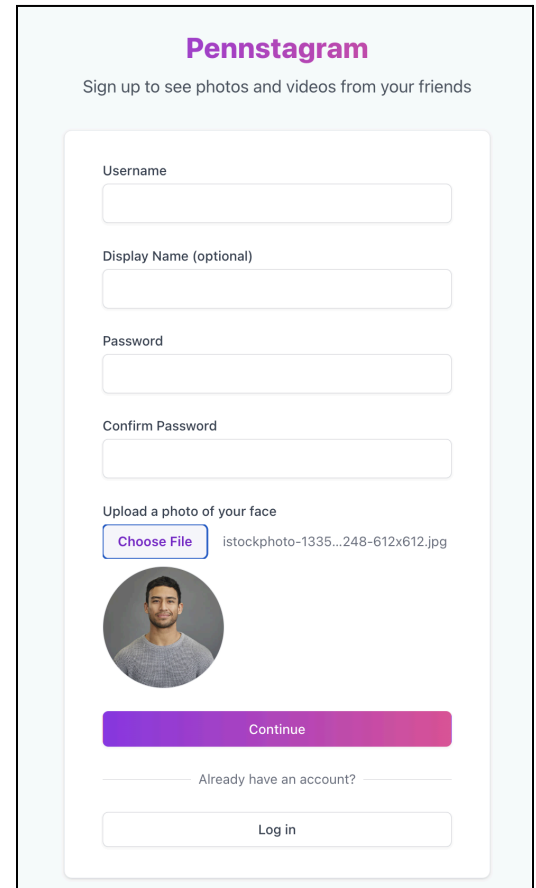
*Rishabh Mandayam, Roshan Bellary, Vedant Gaur, Vincent Lin*

## Introduction

InstaLite is a next-generation social media platform inspired by Instagram, developed with a focus on modern web technologies, scalability, and a smooth user experience. The platform is built using a contemporary tech stack: a React and TypeScript frontend, a Node.js backend powered by Express, a relational MySQL database, and deep integration with AWS services for deployment, scaling, and storage.

What sets InstaLite apart is not only its replication of core social media features such as user authentication, post creation, and liking/commenting but also its advanced functionality. These include real-time chat (implemented via WebSockets), facial recognition for actor matching, and an intelligent content ranking system powered by graph-based machine learning algorithms. These components required a high degree of architectural planning and performance optimization.

This report provides a comprehensive overview of the design decisions, system architecture, and challenges we encountered, along with the solutions we devised to build a scalable, responsive, and technically rich platform.

A mockup of a sign-up form for a social media platform named "Pennstagram". The form is set against a light blue background. At the top, the name "Pennstagram" is written in a bold, purple font. Below it, a tagline "Sign up to see photos and videos from your friends" is displayed in a smaller, grey font. The form itself is a white rounded rectangle containing several input fields: "Username", "Display Name (optional)", "Password", and "Confirm Password". Each field has a corresponding text input box. Below the password fields, there is a section for profile picture upload, labeled "Upload a photo of your face". It includes a "Choose File" button and a preview of a photo with the filename "istockphoto-1335...248-612x612.jpg". Below the photo is a circular profile picture placeholder showing a man's face. At the bottom of the form is a large, rounded "Continue" button with a purple-to-pink gradient. Below the button, there is a link "Already have an account?" and a "Log in" button.

## Technical Components and Implementation Details

### Database Design

We selected MySQL as our primary database over alternatives like DynamoDB due to the need for managing complex relational data, such as many-to-many relationships and self-referencing structures. This allowed for more efficient queries and maintainable data models.

## Chat System Schema:

The chat feature uses three key tables: `chat_sessions`, `chat_participants`, and `chat_messages`. This setup supports both one-on-one and group chats. To allow users to leave conversations without deleting message history, the `left_at` field in `chat_participants` acts as a "soft exit" flag.

## Post Metadata Separation:

Instead of embedding likes and hashtags directly into the posts table, we created `post_likes` and `post_hashtags` junction tables. This significantly improves the ability to filter posts by user engagement or hashtags, which is critical for feeds and analytics.

## Comment Threading with Self-Referencing Posts:

The posts table includes a `parent_post` foreign key, enabling us to support nested comments while keeping the content model unified. Comments are essentially posts with a reference to another post.

## Face Embedding & Actor Matching Pipeline

To enable actor-face matching, we built a sophisticated pipeline that integrates computer vision models with vector databases:

### 1. Image Preprocessing:

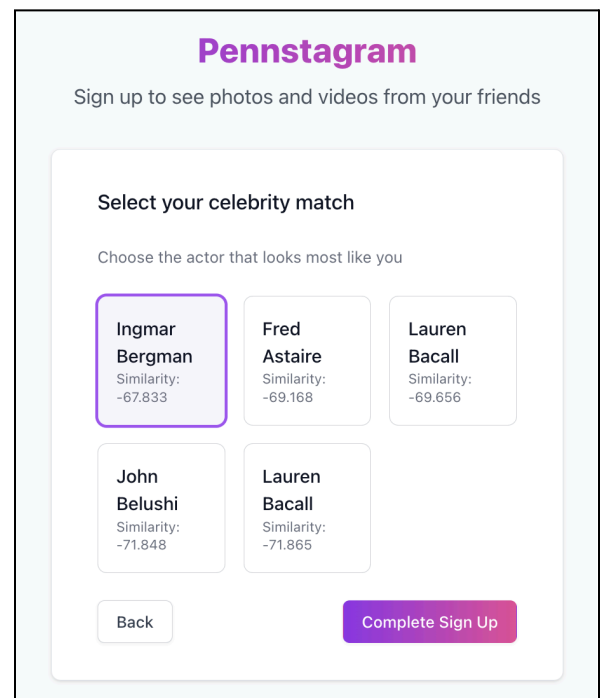
Uploaded images are resized, normalized, and preprocessed using `TensorFlow.js` and `face-api.js`. These transformations ensure that our models produce consistent and accurate facial embeddings.

### 2. Vector Storage:

We chose to store 128-dimensional raw face embeddings (not compressed) to maintain high accuracy. These vectors are stored in ChromaDB, which is optimized for high-speed similarity search operations.

### 3. Similarity Matching:

We employed cosine similarity for comparing embeddings. A dynamic threshold filters out low-confidence matches, significantly reducing false positives.



## Real-Time Chat Architecture (EC: Extra Credit)

As part of the Extra Credit (EC), we implemented real-time chat using WebSockets. This feature supports both direct and group messaging and presented several unique technical challenges.

### *Connection Management:*

We built a custom in-memory map (userId → socketId) to track active WebSocket connections. A heartbeat mechanism monitors client connectivity, ensuring we detect and clean up stale sessions manually, rather than relying solely on Socket.io defaults.

### *Three-Step Chat Invitation Protocol:*

Chat invitations follow a structured protocol: (1) invite is sent, (2) the recipient responds, and (3) the session is formally established. This prevents users from being added to chats without consent.

### *Duplicate Group Prevention:*

To avoid redundant group chats, we created a server-side validator that compares sorted user ID lists to determine if a group with identical participants already exists.

### *Ordering and Delivery Guarantees:*

To ensure proper message ordering even under network delay, we rely on database timestamps. The client re-sorts messages on receipt to maintain chronological order.

## Federation and Ranking System

### *Kafka Integration for Federated Content*

InstaLite includes a federated messaging system using Apache Kafka, which allows it to share and receive content from other instances.

### *Message Format Conversion:*

Internal posts are converted to and from a federated JSON format, allowing cross-platform compatibility. The converter handles attachments and metadata gracefully.

### *Resilient Kafka Operations:*

All Kafka interactions are enclosed in error-handling wrappers, isolating faults and preventing crashes in the main app flow.

### *Content Deduplication:*

A hashing mechanism compares content fingerprints to avoid duplicate posts from federated sources.

## Content Ranking via Adsorption Algorithm

To prioritize high-value content in the feed, we implemented a graph-based Adsorption algorithm using Apache Spark.

### *Graph Construction:*

Relationship data (user-to-user, user-to-post, user-to-hashtag) is pulled from MySQL and converted into an in-memory graph structure.

### *Weight Distribution Logic:*

We distribute edge weights with a 0.3 (user-hashtag), 0.4 (user-post), and 0.3 (user-user) ratio. Normalization ensures the weights for each node's outbound edges sum to 1.

### *Efficient Execution:*

Rather than running Spark jobs inline, we use Apache Livy and a REST controller that triggers jobs asynchronously, freeing our backend from blocking.

### *Batch Database Writes:*

Computed rankings are written in batches, cutting write load by over 85% compared to single-row inserts.

## Challenges and Lessons Learned

### Database Schema Evolution

The database schema evolved significantly throughout development:

1. Initial Design: Focused on core entities with minimal relational structure.
2. First Refactor: Added junction tables to resolve performance issues.
3. Second Refactor: Introduced soft deletion patterns and timestamping across relationships for full lifecycle tracking.

*Lesson: Plan for flexibility from the start; evolving schema mid-project is costly but sometimes necessary.*

### WebSocket Scaling

1. Initial implementations were single-server and failed under load when scaled horizontally.

- a. Issue: In-memory WebSocket maps could not be shared across instances.
- b. Solution: Introduced Redis as a socket.io adapter, allowing distributed state tracking.

*Lesson: Real-time systems must be built with horizontal scalability in mind from day one.*

## Image Processing Performance Bottlenecks

1. Original Setup: Image embeddings were processed synchronously.
  - a. Problem: Slow response times and CPU overload.
  - b. Fix: Migrated processing to a worker queue with background execution.

*Lesson: Offload CPU-heavy tasks immediately when scaling.*

## ChromaDB Integration Complexity

1. Unexpected Learning Curve: ChromaDB's vector format and connection model were more complex than expected.
2. Resolution: Built a custom wrapper with connection pooling and error handling.

*Lesson: Allocate time for deep integration when adopting specialized tools.*

## Conclusion

InstaLite represents a complete, robust, and scalable social platform designed with modern engineering practices. From federated content sharing and real-time communication to face recognition and ranking algorithms, each feature reflects thoughtful design and problem-solving.

Our use of a hybrid authentication model, a scalable WebSocket chat protocol (EC), advanced database patterns, and machine learning techniques highlight our commitment to quality, innovation, and performance.

The experience taught us valuable lessons about system design, architectural flexibility, and production-grade scalability that we will carry forward into future development efforts.