NETS 2120 Final Project - Instalite

David Zhan, Yanfu Ou, Eric Zou, Zihao Zhou

Overview

The goal of the project Instalite is to build a web application emulating the functionality of Instagram. Instalite has all the features of Instagram and more – user accounts, image posts, feed recommendations, comments, image matching, chatbot, and chat – supported by advanced backend services. The project aims to demonstrate the successful integration of an optimized frontend user experience supported by the powerful backend. Instalite is designed for scalability through integration with AWS cloud services. Last but not least, security is ensured through proper user information management prior to storage. This project emphasizes scalability, security, and modular design.

Technical Description

Project InstaLite is a full-stack, Instagram-style web application developed by a team of four students. It features secure user authentication by hashing and salting passwords with Bcrypt, ensuring that raw passwords are never stored. After successfully authenticating, the server cookie stores the session ID of the authenticated user. All protected features and endpoints required a valid session ID to proceed. Additionally, it guards against accessing content by simply entering the URL. This ensures security and protection against unauthorized access.

The team carefully designed an interface for users to upload images for similarity matching. The app supports image-based matching via ChromaDB, a vector database optimized for similarity checking. All image data were pooled from S3 buckets and loaded into the ChromaDB with the actor indexer we made. We used Tensorflow and the provided model weights to generate image embeddings for each of the sample actors. When a user uploads a profile picture for themselves, those are also put through the same vectorization process, then a list of similar actor images are generated based on those similarities.

InstaLite provides an easy and intuitive chatbot to help users find relevant posts, search relevant movie facts, and discover new profiles. The chatbot is powered by Retrieval-Augmented Generation (RAG), which uses ChromaDB for context retrieval by searching the database for similarities. The context is used to feed into a GPT prompt with user questions and system instructions for natural language responses. This design helps to supplement the short-term memory nature of LLMs. In the Chatbot feature of InstaLite, the RAG prompts are passed to the ChatGPT API, which returns an answer to the user's question within the context of following our optimized instructions.

The chat feature allows real-time private and group chats. It is implemented using socket.io and AJAX polling. The chat feature allows users to invite online friends to chat; upon acceptance, a session is created. Chat history is persistent and retained across sessions. Users can leave chats anytime, and sessions are deleted when the last user exits. Group chats are supported, with new members gaining access to prior messages, as well as persisting messages across user sessions and browsers. Each group chat is unique, and duplicate group compositions are not allowed.

For the feed, posts are streamed from either the FederatedPosts Topic or the Blue Sky Topic. Any post made by anyone can be consumed and displayed on the feed for a particular user. To interact with these posts, the user can like or comment, both of which persist through both refreshes and new user sessions. Additionally, the user has the ability to post his or her own photos with descriptions and hashtags. This post would be streamed to the FederatedPosts Topic through a Kafka producer node and be available publicly for everyone to view.

The frontend is built with React, while all user interactions are routed to the backend via Express.js. The backend server is hosted on AWS EC2 alongside the frontend server. Data is stored in DynamoDB, and media files are managed in S3. Our MySQL data tables are hosted on RDS, which is linked to the EC2 instance that is hosting our backend. Federated posts are exchanged between teams through Apache Kafka. To ensure security and seamless integration of all AWS services, the team built a virtual private cloud(VPC) linking all the cloud components.

Design Decisions

We focused on a modular design to ensure scalability and ease of future development. By developing core features like user authentication, posts, likes/comments, and chat as separate modules, we made testing, debugging, and deployment easier. This modular structure also allows for independent scaling of each feature, ensuring performance bottlenecks can be addressed without affecting the entire system.

For user experience, we aimed for a minimalistic, Instagram-like interface that is both functional and easy to navigate. The layout is clean and includes a persistent navigation bar with core features easily accessible. We implemented infinite scrolling for the feed, allowing posts to load dynamically as the user scrolls, closely mimicking Instagram's smooth scrolling experience.

The chat and friend request system is powered by websockets to ensure real-time communication. This system supports private and group chats, allowing users to send messages instantly. We also implemented a robust friend request process, ensuring that both users must confirm the friendship before initiating a chat.

To optimize performance, we leveraged AWS cloud services like EC2 for hosting, RDS for MySQL databases, and S3 for media storage. These services enable scalability as user traffic increases. For real-time updates, we used websockets to ensure chat messages and feed updates are pushed instantly to users.

The image matching system uses ChromaDB and TensorFlow to process user-uploaded images and match them with similar actor profile pictures. By storing image embeddings in ChromaDB, we can efficiently retrieve the most similar images, providing users with a personalized experience. This approach was chosen for its ability to scale with a growing database of images.

For security, we implemented bcrypt for password hashing and salting, ensuring that passwords are never stored in raw form. The application also enforces strict session management, requiring authentication for protected resources and preventing unauthorized access through direct URL entry.

After the first milestone, we optimized our database schema. Initially, we had a large, centralized table that became difficult to manage. We redesigned it by splitting it into multiple tables for users, posts, comments, and likes. This improved both data organization and performance, especially when implementing advanced features like social ranking and graph-based algorithms.

Finally, the chatbot was designed to help users find relevant content and explore new profiles. It uses Retrieval-Augmented Generation (RAG) to retrieve relevant context from ChromaDB and generate natural language responses using GPT. This combination enhances the chatbot's ability to provide personalized, dynamic answers, making it more useful than traditional Q&A systems.

These decisions ensured that InstaLite would be secure, scalable, and provide an engaging user experience.

Design Changes

Over the course of building the web app we made a couple of changes.
First, before the first milestone, we had implemented each of the 4 main features completely separately. This was due to the fact that we believed integration was far away. But soon we realized that the problem was bigger than we had imagined. Our folder structures, route handling, and in general code structures were completely different. It would be a nightmare to integrate. Thus, after the TA,

Secondly, we optimized the database schema to optimize the storage of data. Initially, we had a massive table containing user information(user_id, username, hashed_password, linked_nconst), followers, friends, friend_requests, and posts. This table had over 20 columns and became a complicated mess rapidly. It was poorly organized and very confusing.

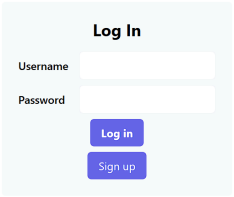
After the milestone 2 demo to the TA, the team met to redesign the database schema. We agreed to split our massive table into multiple tables. This includes a new user, follower, friend, and friend request table. Additionally, we have tables for managing likes, comments, and posts, which helped not only the organization of the data, but also in general the implementation of future features, such as the social rank and adsorption algorithms. Since we had light weight tables, we were easily able to port the "edges" of the adsorption graph into JavaRDDs due to the enhanced schema.
This new schema involved features linked by important keys such as user_id or post_id. The redesign helped to organize data more efficiently and standardized the schema.
We learned a valuable lesson on database schema organization and big system design clues.

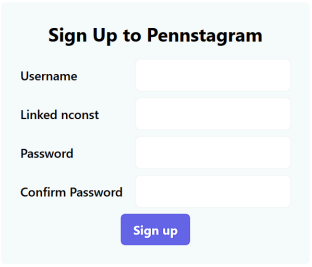
Extra Credit Features

For extra credit features, we used websockets to implement our chat feature, friend requests between users, infinite scrolling, and a very instagram-esque UI for the feed.

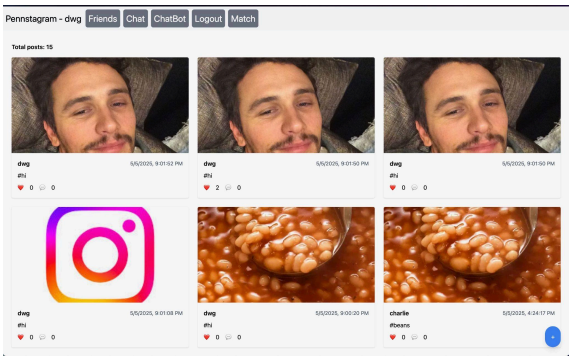
System in Action

The user experience begins at the login page, where a clean UI prompts the user for their credentials.

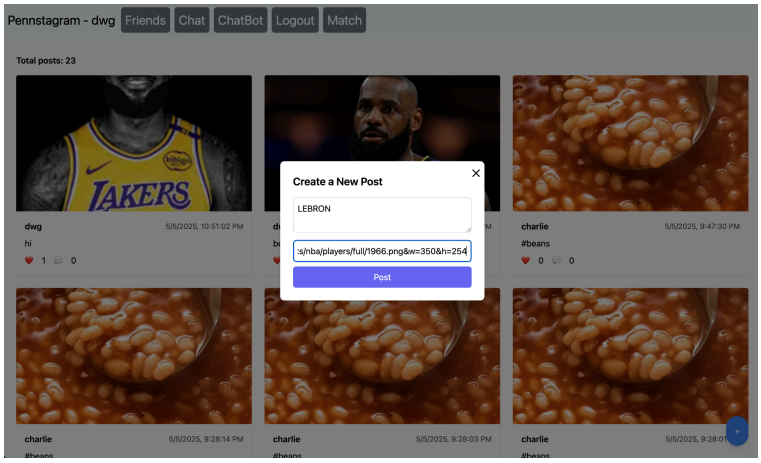




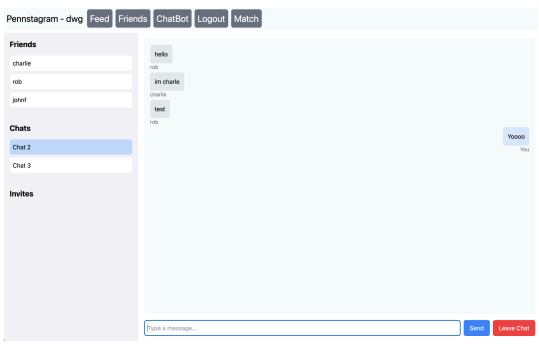
From there, our system salts and hashes the password, which is compared against the database. If the user is new, they can choose to sign up for an account.
After the user has successfully logged in, they will be welcomed to the home page. The home page displays the user's recommended feed. The feed is ranked using our implementation of the adsorption algorithm.

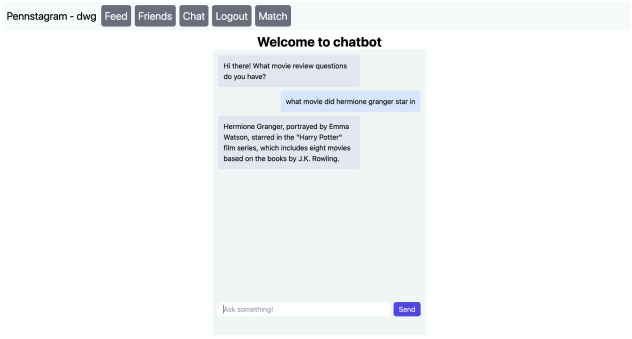


Users can post their own pictures, like other posts, and comment on the posts. These features emulate the instagram functionality. The posts are fed through the Kafka pipeline.

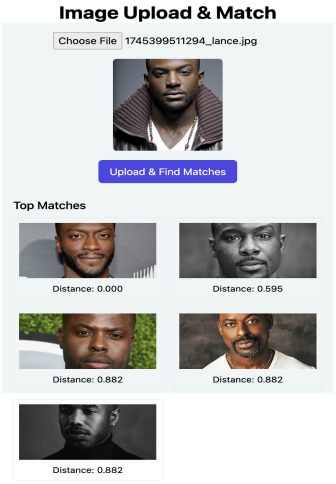


Users have the option to interact with friends online through chatting. In order to chat, the logged-in user has to send a friend request. Then, in another session, the friended user must accept the request. Upon accepting the friend request, a private message session is initiated. Then the two users will have the option to send and receive messages. Below is an example of a group chat between 4 users.
We also have the feature of being able to communicate with a chatbot for queries about movies, actors or even posts. This is an example of a conversation with a chatbot.





Below is the image upload/profile page. When you upload an image as a profile picture, your image will be vectorized into one of our image embedding, using the provided model weights. Our application will then query our chromaDB to find actors whose embeddings are most similar to you. The user can then select the profile picture from the list of 5 actors that best match their interest



Thanks for reading!!