NETS 2120 Final Project Report - Instalite

Group bhbk: Bach Tran, Benjamin Xu, Kevin Lu, Nhat-ha Pham

System Overview

Instalite, inspired by Instagram, is a social media platform that utilizes a React (Next.js) frontend deployed on Vercel, data storage on Amazon RDS and S3, with Apache Kafka for streaming and Apache Spark for data processing tasks. The system supports functionalities like social graph management, chat, and dynamic post ranking, post management, hashtagging, commenting and liking.

Components (screenshots below)

1. User login
2. Profiles
3. Chats
4. Posts management
5. Friends management and recommendations
6. Search with NLP/RAG
7. Feed, including Twitter and federated posts & ranked posts.

Database design:

Our database features *user*, *posts*, *comments*, *hashtags*, *likes*, *chats*, and *chatMessages,* tables. Relationships between these tables are captured via **userChats**, **postsToHashtags**, **userFriends, friendRecommendations, postRecommendations, userHashtags, usersRelations, ChatsRelations, userChatsRelations, hashtagsRelations, postsRelations, commentsRelations,** and **likesRelations** relationships. The postRecommendations table associates users and post with a weight amount to rank them on the user's feed.

Server Design:

Our server sections InstaLite APIs into different subsets. There are subsets for posts, hashtags, search, comments, liking and chats. Each API interacts with Drizzle ORM, which interfaces over the MySQL RDS instance. It also connects to more miscellaneous sources such as ChromaDB (image search), S3 (uploading images), Pinecone (text search and RAG) and OpenAI API (for LLM). The server uses Node, Express, all in TypeScript

Client Design:

Our client uses React under the hood but supercharged by Next.js, which allows us powerful SSR benefits. With Next.js, we are able to have a performant frontend with automatic code-splitting, tree shaking optimization and server/static generation. We are also able to deploy more easily on Vercel Cloud, making the frontend available for all. We use Shadcn UI, built on top of Radix UI, to build our components. We also use tailwind css to style our components. Our components are also sections into directories similar to how we section our API subsets.

Social Graph:

We update our graph every hour, using cron. Each hour, the cron spins up a Python script. Our Python script connects to our Spark cluster, constructs the social graph from tables within our EC2 instance, loads it into Spark and finally, loads the graph data into GraphFrame. We utilize the GraphFrame method to run

our adsorption. After the adsorption, we then decompose the graph again and loaded back into the table rankedPost, where we associate user and post pair with a weight attribute, to be used for ranking

Post management:
Users can create a post with text content. They also optionally upload a graphic to the post, which will be uploaded to the S3 bucket and returned as an URL. We also scan for hashtags within the text content to attach hashtags to the post. Posts also have likes and comments, which we expose as many-to-many relations in our database. We also scan comments for hashtags to attach to the post.

Friends Management:
We maintain a table on friendships, and a table on friendship recommendations, both visible through the Friends page. Every day the cron PySpark job will refresh the friendship recommendations table with new rankings. In terms of friend management, users can add, unfriend, and are suggested to befriend related users when they search a term linked to those users.

Search:
We use ChromaDB for image searching. When users upload profile pics, we then embed the user's image and compare embeddings with those from our current Chroma vector store, along with their respective actor ids. We then query the actor ids from the past imdb database to get actor names. We also embed posts and users data whenever they are created. These are loaded into a Pinecone vector database, which is used for RAG data ingestion into our LLMs as well as normal embeddings search. We use Pinecone since it displays faster querying results compared to ChromaDB.

Kafka Twitter and Federated Post:
We run Kafka as a process within our Express server. Kafka hydrates our posts table every day, by ingesting and extracting posts data from the stream, scans it for hashtags and users and ingest it back into the database.

User authentication:
For sign up, we take in username, password, affiliation, first name, last name, DOB. We create users with usernames as our unique requirement. After signup, we prompt users to upload a profile pic. Profile pics are uploaded into S3 and then read by our face matching code to return a list of similar looking actors. We allow users to choose which actor they want to link to as well as picking a list of up to 5 hashtags they are interested in. These hashtags are saved to build the social graph. Passwords are salted and hashed into the database to ensure security using bcrypt. When logging in, we use bcrypt again to compare the hash to see if they are equal, meaning the password is correct. We then save the user into the session using express-session. We also install authentication middlewares within our routes to check whether the user sessions still persist and the user is still logged in, as additional measures for routes requiring authentication. When logging out, we flush the user session.

Chats
We create chat sessions of 2 types, either those that are only between 2 users (you and each of your friends) or those within a group (you, along with multiple other users). Each chat session keeps track of how many users it has as well as the messages it has, for persistence.. Each message tracks its owner,

content and creation time. We order messages based on dates to ensure ordering. We also use WebSocket as an efficient way to stream chat messages efficiently to users connected to a chat session. Users connect to the chat session by giving the server the chatId they want to connect to and the server will open up a websocket connection. The WebSocket will distribute messages to all chat users while also saving the chat messages back into DB. Note, users can leave a group (which notifies the group) or deny group invitation requests.

Changes & Lessons

- Adopting efficient developer paradigms Drizzle ORM and TypeScript, while taking a bit more time to set upfront, eventually introduces great developer productivity as we delve deeper into the project and more complex features
- React UI hydration can be a problem if you are dealing with constant updates and API calling across many features. Context provider can help introduce more simplicity
- Kafka streams lead to an influx of data, and we have to find ways to make the streaming seamless.

Extra Credit Features

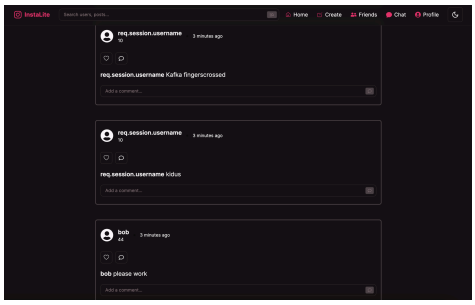
1. Server Side Rendering of data (first few posts, friends lists, users data, etc) for enhanced performance
2. Stale While Revalidation strategy: Fetching data in the background and then refreshing it once successfully fetched. This powers almost all GET requests coming from the frontend, ensuring data freshness, live updates with minimal UI lag
3. WebSocket powered Chats: Chats are built with express-ws on the backend and uses the native Websocket API on the frontend
4. Theme toggling: Utilize next-themes context provider to switch between light, dark and system theme.
5. Infinite scrolling: Fetches new data when users reach the end of the feed. This fetching is also powered by Stale While Revalidation for enhanced performance
6. Components and styling library: Uses shadcn components library, with tailwindcss styling
7. TypeScript: Utilize TypeScript throughout for enhanced productivity and typesafeness
8. ORM integration: Integrate Drizzle as an ORM and query builder for faster development, enhanced latency and data management
9. Always return valid user and post objects/links from the LLM
10. Display hashtag mentions counts on post
11. GraphFrame for social ranking
12. Context provider to share data across components easily
13. GitAction CI/CD

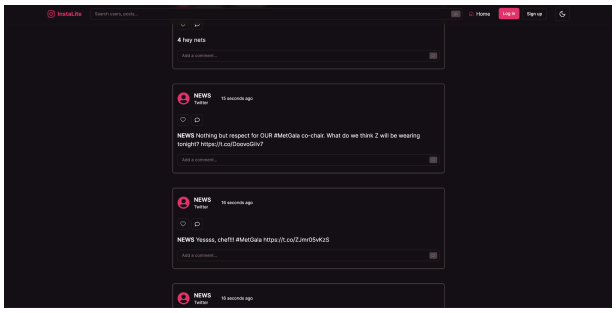
NETS 2120 Final Project Report - Instalite

Group bhbk: Bach Tran, Benjamin Xu, Kevin Lu, Nhat-ha Pham

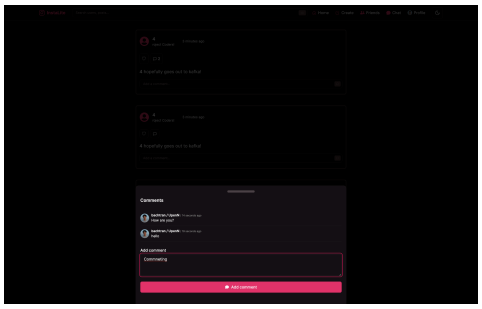
Screenshots

Feed (shown below are federated posts & Twitter, ordered by algorithm):

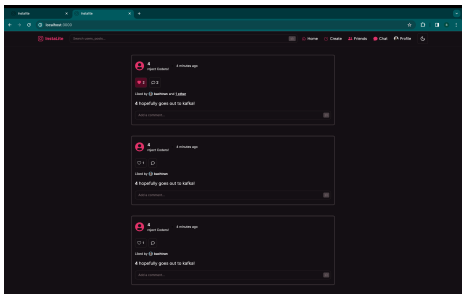




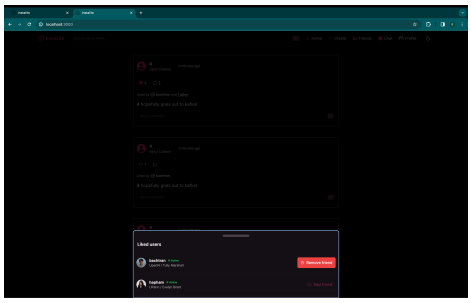
Commenting:



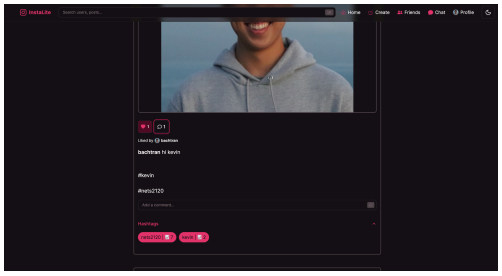
Liking (show up as highlighted heart Icon):



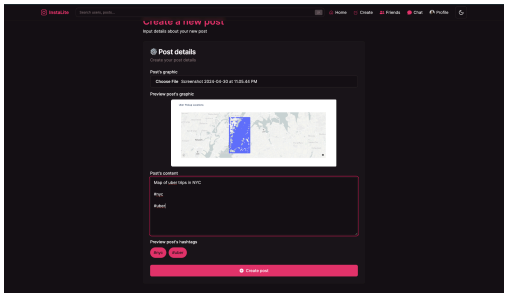
Showing likes (show up list of liked user):



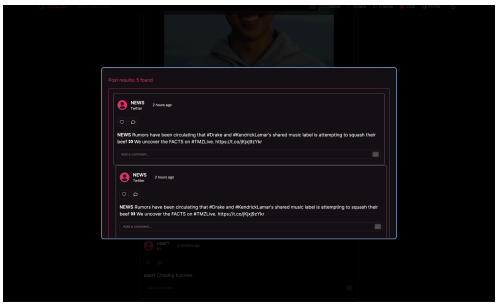
Hashtags of posts, with total mentions shown:



Post creation (can upload image):



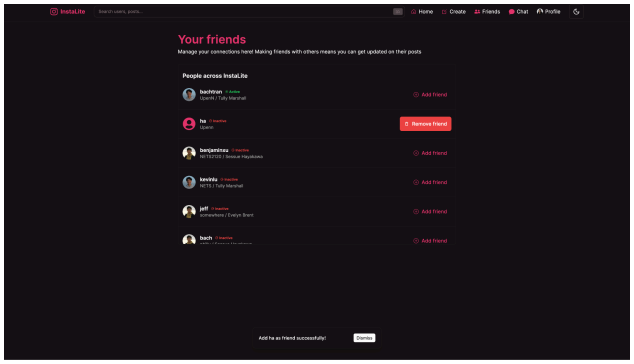
Search (Post results):



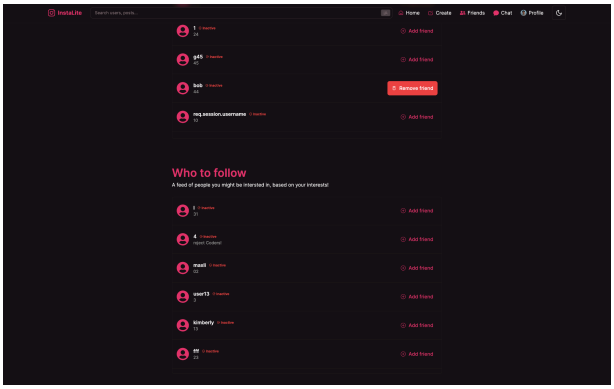
Friends management (adding/removing/recommending friends with online status)

NETS 2120 Final Project Report - Instalite

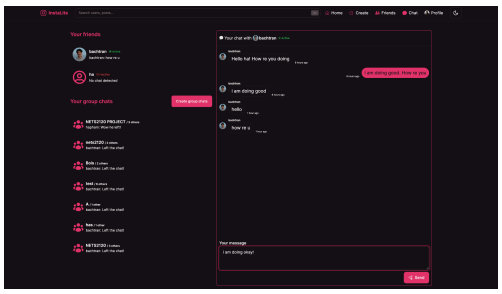
Group bhbk: Bach Tran, Benjamin Xu, Kevin Lu, Nhat-ha Pham

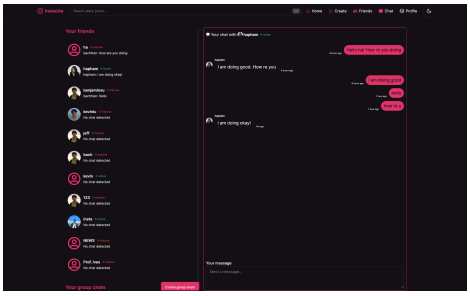


Who to follow (recommending friends to follow based on interests):

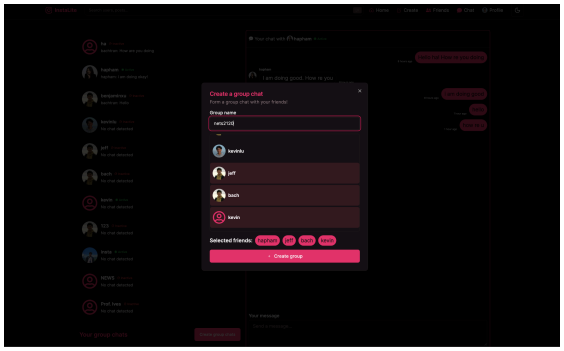


Chatting to a friend (built on websocket):

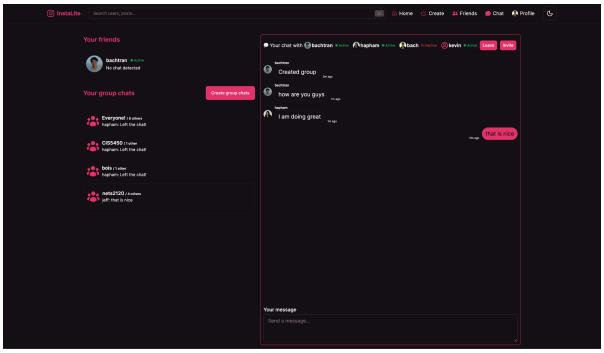




Chat group creation:



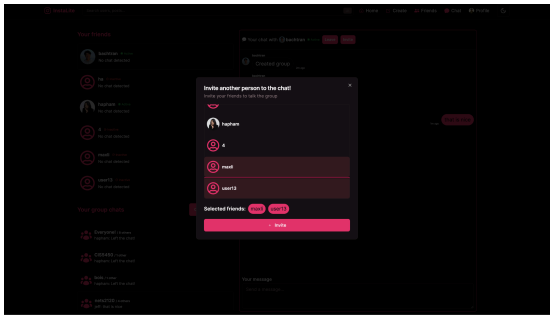
Chatting in chat groups:



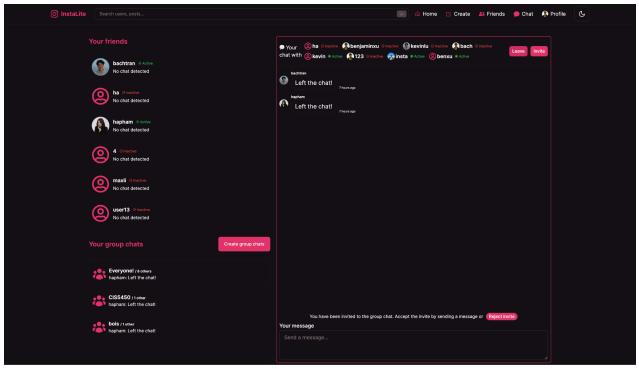
NETS 2120 Final Project Report - Instalite

Group bhbk: Bach Tran, Benjamin Xu, Kevin Lu, Nhat-ha Pham

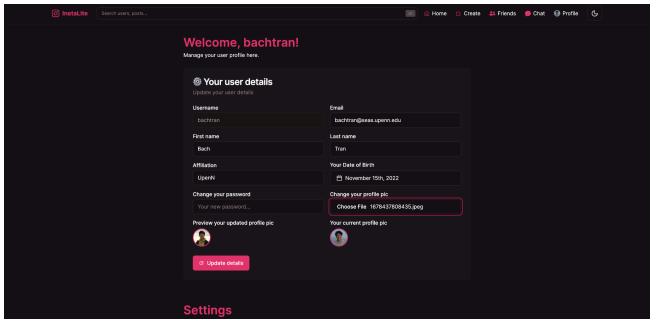
Inviting another friend to a chat group:



Leaving a chat group (left users will not be able to see the groups, unless reinvited):



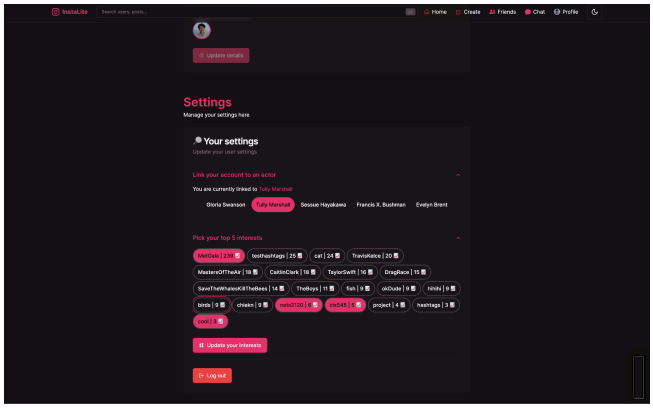
Change user profile details (including uploading new profile pics)



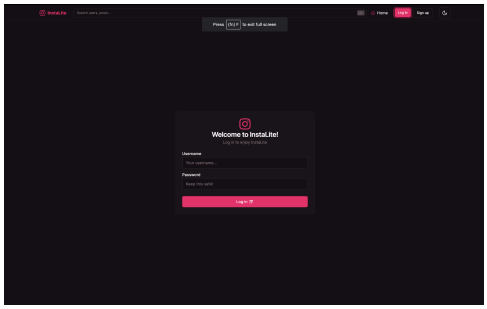
Getting user's image matches and hashtag interests:

NETS 2120 Final Project Report - Instalite

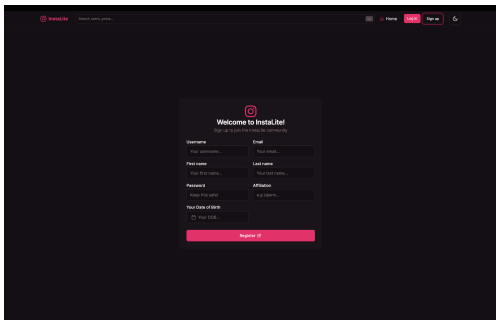
Group bhbk: Bach Tran, Benjamin Xu, Kevin Lu, Nhat-ha Pham



User log in:



User sign up:



Search (LLM RAG + user links returned):