

NETS 2120 Spring 2025 Final Project: InstaKANN

Armaan Ahmed, Kurtis Zhang, Nathan Lee, Neha Peddinti

armaana@sas.upenn.edu, kurtisz@seas.upenn.edu, nzlee@seas.upenn.edu, peddinti@seas.upenn.edu

System Overview

Our Instalite system allows users to register profiles, interact with other users, and scroll through recommended posts on their feeds. The system includes persistent storage for posts, chat messages, and user profiles, as well as a post-ranking algorithm that generates individualized feeds upon login. We also use session management mechanisms for security.

Technical Components

I. Kafka

- A. We used Kafka to subscribe and post to the BlueSky Kafka topic provided by the course, in addition to the FederatedPosts topic that contains aggregated posts from other project teams. Within our project's route to createPost function, we use a Kafka producer to send our posts to FederatedPosts so other teams can have access. Within the consumer file, Bluesky tweets and FederatedPosts are added to our "posts" database as well. Dummy user_ids are created and used for representing posts for BlueSky and FederatedPosts.
- B. Challenges Encountered: One of the issues that we ran into is maintaining consistency within our posts database so that 1. Posts that are sent to FederatedPosts are not re-created within our database when the consumer reads our own post data while reading from the topic and 2. Ensuring we do not resend a FederatedPost back to the topic with our createPost route. To solve the first issue, we check if a post's content already exists in our database upon reading from FederatedPost, only creating a new post if it does not exist. To solve the second issue, we created an addExternalPost route (different from createPost) called when the consumer reads so that we are not re-sending posts to FederatedPosts.

II. User System

- A. Users are able to **register** through the sign up page which occurs in 4 different sections. The first section allows users to input their information, the second section allows users to upload a profile picture, the third section allows users to select a similar actor from the k-nearest embeddings vector search, and the last section allows users to select their interests using hashtags.

Welcome to Pennstagram!

Create Your Account

Username*


Email Address*

First Name*

Last Name*

Password*

Confirm Password*

Birthday*
 

Affiliation*

[Continue](#)

Welcome to Pennstagram!

Upload Your Profile Picture



[Change Picture](#)

Upload a picture to find similar actors

[Back](#)

[Continue](#)

Welcome to Pennstagram!

Choose a Celebrity Look-alike

No matches found. You're unique!

[Continue to Interests](#)

[Back](#)

Welcome to Pennstagram!

Add Your Interests

Popular hashtags:

[#penn](#) [#nets2120](#) [#upenn](#) [#cis](#) [#computer_science](#) [#philadelphia](#) [#ivy_league](#) [#tech](#)
[#coding](#) [#student](#)

Your interests:

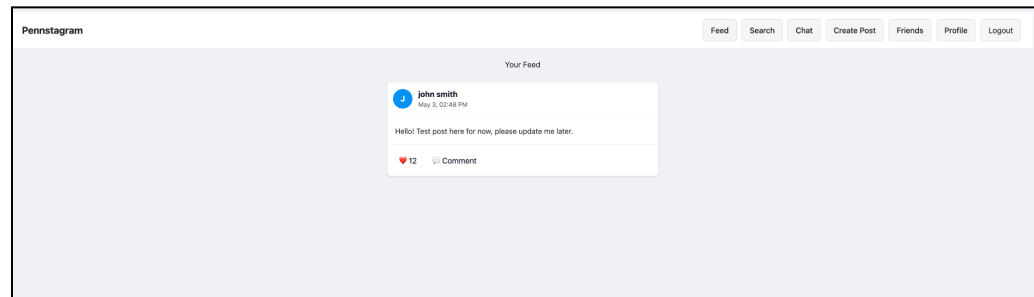
[Add](#)

[Back](#)

[Finish](#)

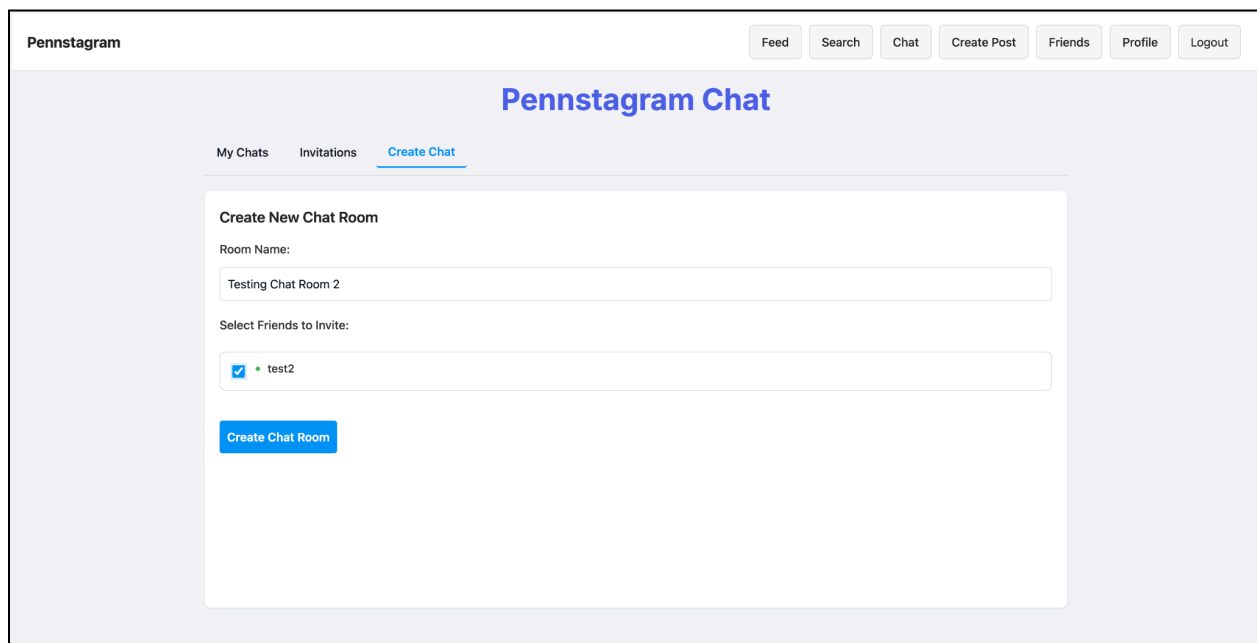
III. Feed

- A. The Feed page shows the posts that users have made by querying from Kafka as well as our own database. Posts created by users will show up here.

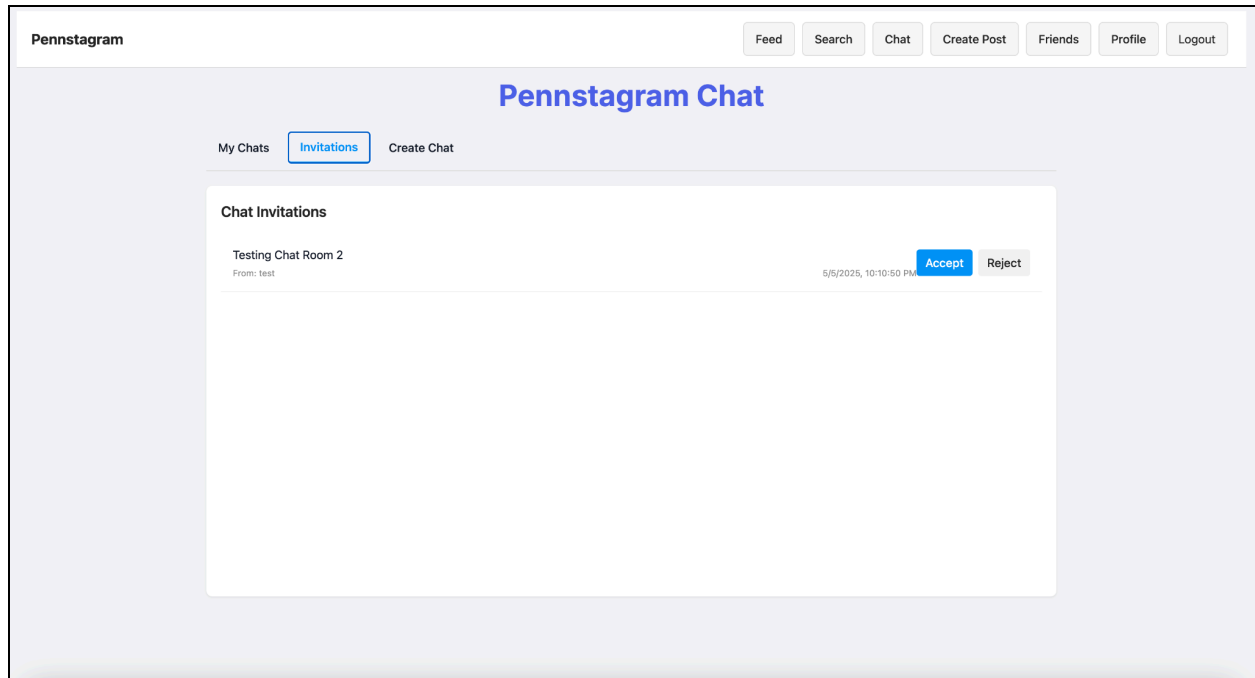


IV. Chats

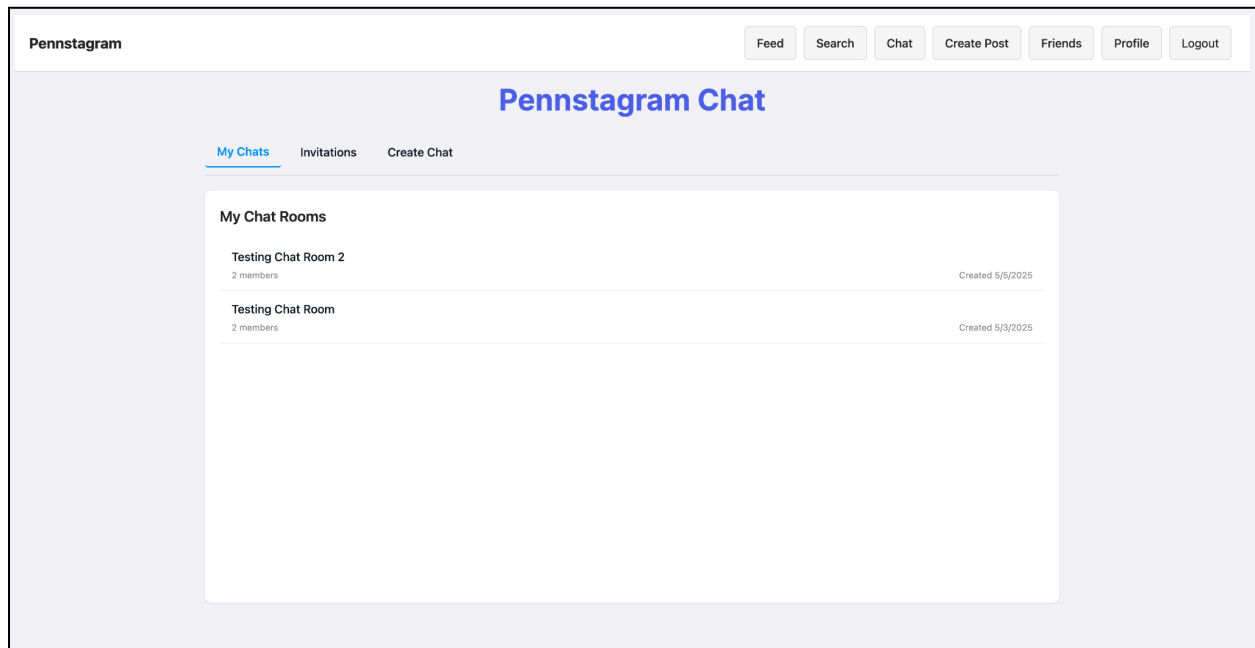
- A. On this page of our application, users are able to create chat groups and enter them in order to send real time updated messages to other users. Users can begin by creating a chat through the create chat page, and select their friends to invite to the room. This works by creating a new chat invitation in the database linking the users that are invited to the chat room id.



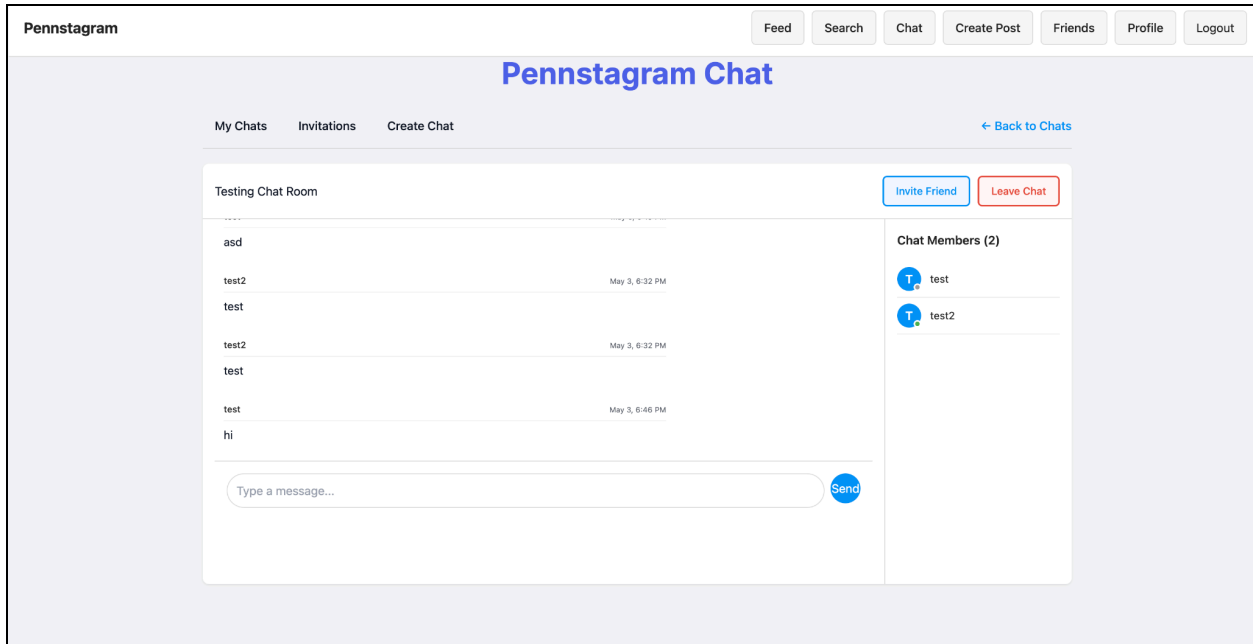
- B. Once the user pressed create chat room, the application sends invitations to the respective users. These will show up under the invitations tab for users to accept. Once accepted, the user can return to the My Chats page to enter the chat room. This works by having the users that accepted their invitation be added to the database that manages which user is in what room. These are all accessible by the user id and room id keys.



- C. Users are able to connect to the chat room through websockets and send messages to each other which will update in real time. These messages are saved to our database for chat persistence. This means that users are able to log in and always see their chat history.

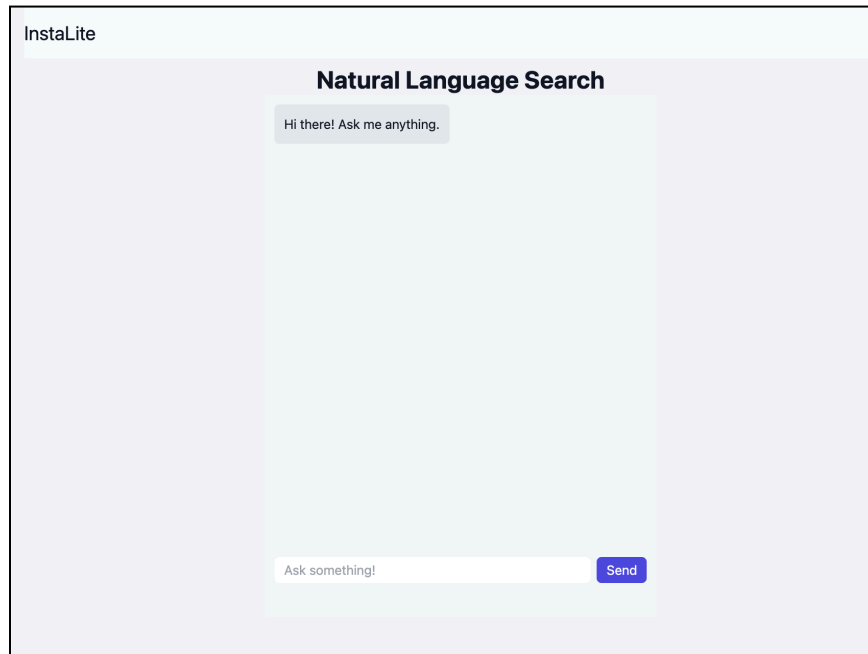


- D. Inside the chat room, users can also invite more friends to the room by pressing the invite friend button. Users can also choose to leave the room and once all users have left the room, it is removed from the database. This works by checking our database to see if the chat room has no users linked to it anymore.



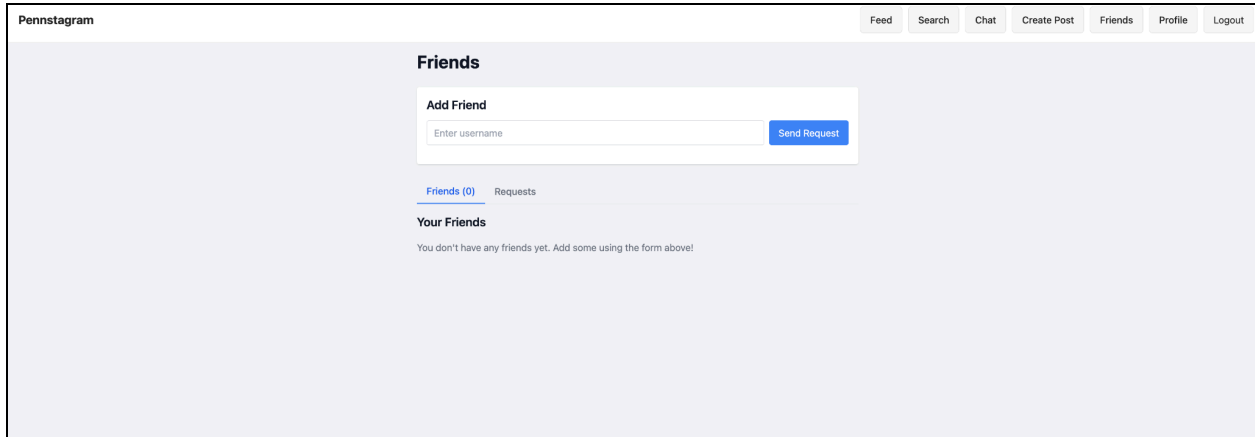
V. Natural Language Chatbot Search

- A. The OpenAI API is used in our web application to do a Natural Language search on our database to find posts relevant to the user's input query. This is based on the content of the post as well as the hashtags.

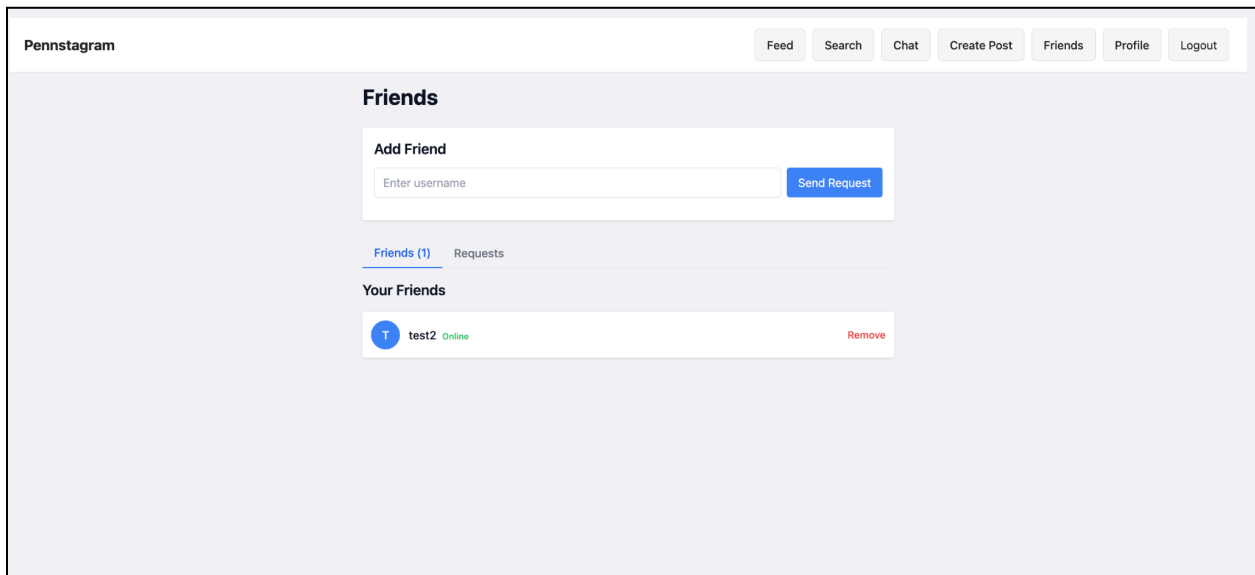


VI. Friend System

- A. Users are able to add friends by looking up their username. Then they are able to send a friend request to the user. This creates a new friend request in our friend request table and links the users together by id.

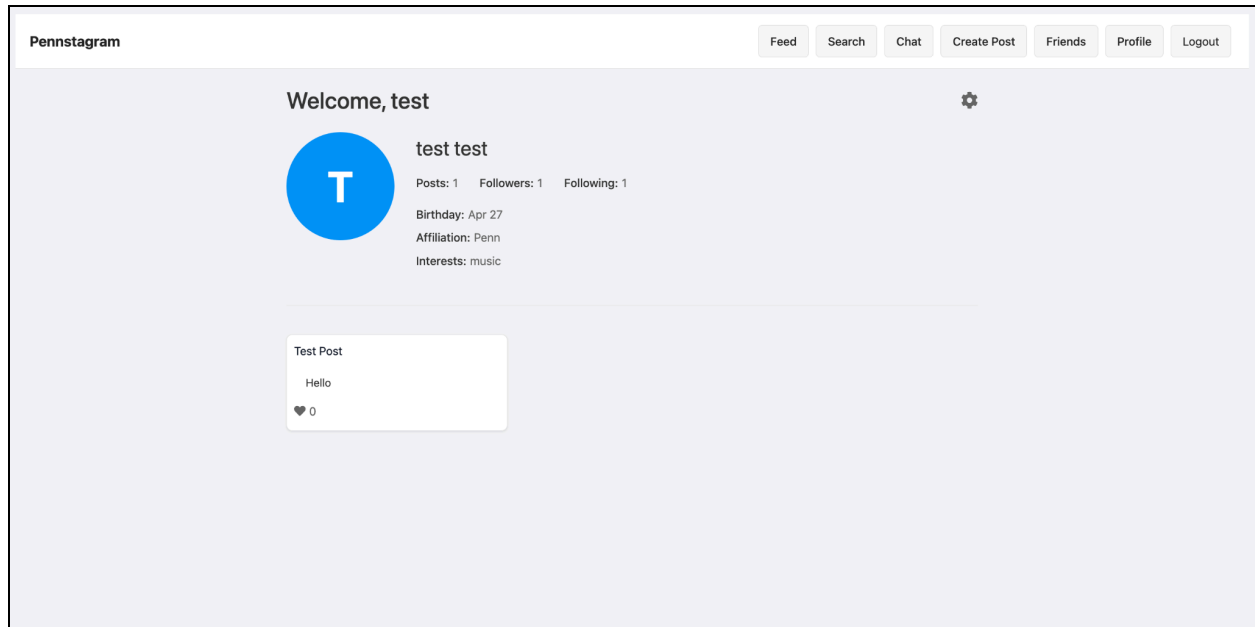


- B. When users are on the friends tab, they are connected to a websocket which allows them to see whether their other friends are online in real time. This websocket connection emits whether or not the friend is online or offline.

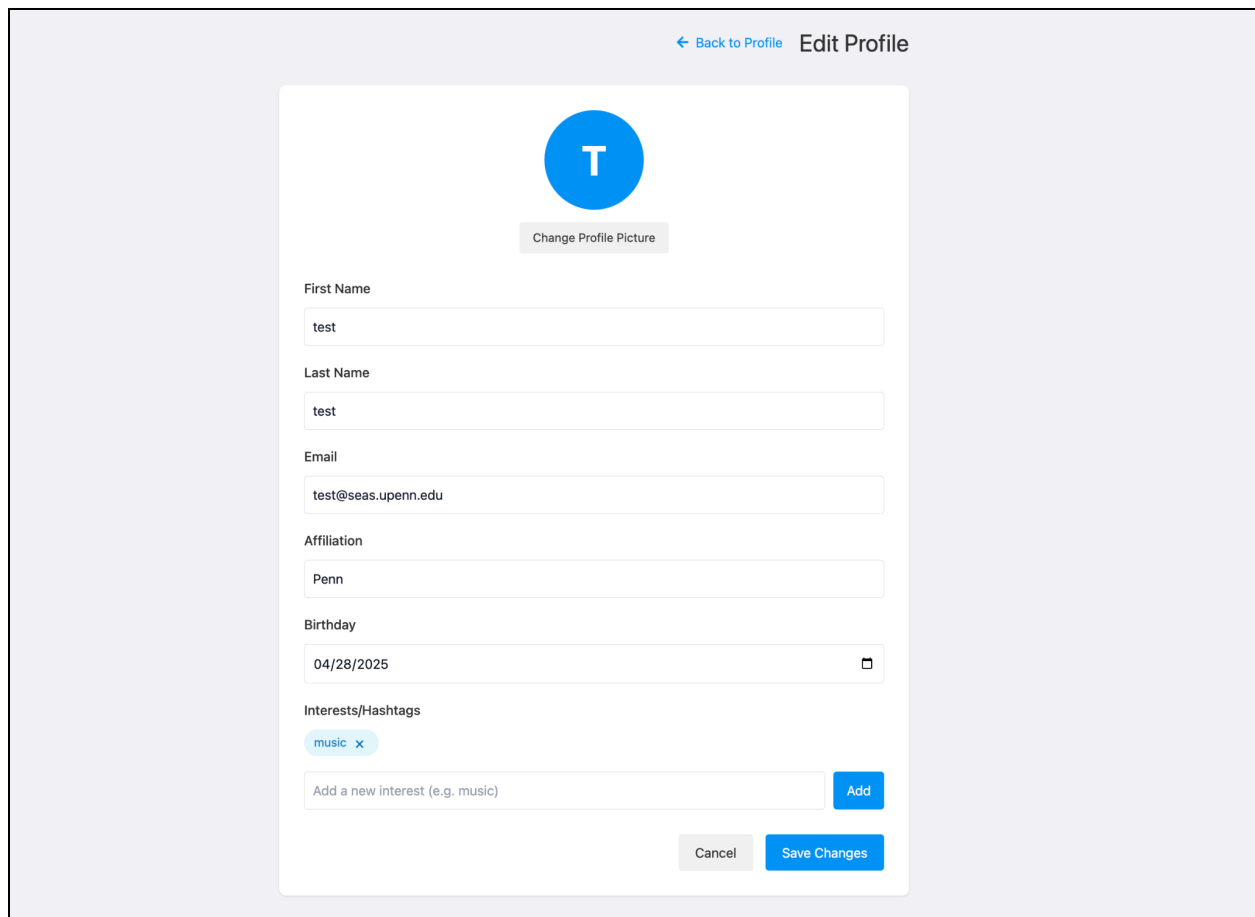


VII. Profile

- A. Users are able to go to the Profile page to view their profile. The information includes the number of posts, followers, following, birthday date, affiliation and interests. It also shows any posts that the user has made. This works by querying the information from the users table as well as joining on the posts and following table to get their posts and follower counts.



- B. Another feature on the profile page is the ability to edit the user's information which sends a POST request with all the new information to the respective tables for the user.



VIII. Ranking Algorithm on AWS EMR, and feed generation

- A. The adsorption algorithm reads from the RDS database and uses Apache Spark to create edges of a social network, weight the edges as required, and perform label propagation. The results are stored in the *post_rankings* table in RDS, where each row contains a *user ID*, *post ID*, and *weight* corresponding with how much that post is recommended for that user. We designed our output this way so that the *getFeed()* route can use MySQL to filter for a given user and query for their highest-weighted posts.
- B. We use a friends-of-friends approach to find follow recommendations, and this algorithm is also implemented as a Spark Job so it can run periodically when adsorption runs. It posts to a *recommendations* table that the backend can query from when user follow recommendations are needed.
- C. We use an automated serverless workflow with AWS Lambda and Amazon EventBridge to trigger the execution of the ranking algorithm (packaged as a JAR file and stored on S3 for persistence), which is scheduled every hour.

Design Decisions

- I. Persistent storage is implemented with S3 to store images from posts and profile pictures, and ChromaDB to store embeddings of actor photos, which are matched against the profile picture at runtime. We host the frontend and backend on EC2, and the adsorption algorithm for ranking is implemented with Apache Spark, which interacts with Apache Livy and runs on EMR.
- II. All other information is stored in RDS.
 - A. We have a *users* table that includes each user's ID (only used internally; not publicly available), other required info, and a "last_online" field that is used as a fall-back in case the Web Sockets used for chat messages fail.
 - B. We designed our schema so that the social network graph (with edges between users, hashtags, and posts) can be created efficiently for adsorption. Our *posts* table contains a comma-separated "hashtags" field. Our *likes* table keeps track of which users have liked which posts. Our *hashtags* table also maps users to a comma-separated list of hashtags they indicated interest in during registration, to create the graph for adsorption.
- III. We keep track of sessions with the *sessions* table, which maps user IDs to session IDs that are generated upon login and expire after 7 days. This is to allow a user to conveniently stay logged in on their machine if they choose to, but sessions are not indefinite due to security concerns. We also add a 64-byte random token (instead of a user ID cookie) to prevent impersonation of a user from simply setting the token to an arbitrary ID, and HTTP-only cookies to prevent scraping from malicious browser processes and/or cross-origin resource sharing attacks. That is, it is rather challenging to impersonate a user on our site.


```
Database changed
mysql> show tables;
+-----+
| Tables_in_instalite |
+-----+
| chat_invites        |
| chat_members        |
| chat_messages       |
| chat_rooms          |
| friends              |
| hashtags             |
| likes                |
| names                |
| post_rankings        |
| posts                |
| recommendations     |
| sessions             |
| users                |
+-----+
13 rows in set (0.01 sec)
```

Changes Throughout Process

- I. Many changes to our database schema were made throughout the process of this project. This included adding several tables to focus on handling the chat as well as schema changes to the users table. These changes made it easier for us to extract specific data we needed from the backend, allowing it to be more efficient in querying and processing data between tables concurrently.
- II. We decided to pursue the chat with websockets architecture. This meant that we had to change the chat's initial REST requests to instead be a connection through a websocket allowing for real time updates. This was a cool approach that meant that messages were emitted every time they were sent so all users in the room could see it. We decided to keep this architecture and also reuse it on the friends page so that the online status of all users was kept in real time.

EC Features

- I. Friend requests
 - A. Users are able to send friend requests to other users. They are able to then accept or decline a friend request from other users.
- II. Friend Status with websockets
 - A. Status is shown in real time by using websockets. This works when a user connects to the websocket and broadcasts that their status has changed to online.
- III. Chat with websockets

- A. Chat messages are sent in real time through websockets since the messages are broadcasted to all users that are in the room as soon as they are sent.
- IV. Password reset email
 - A. Users that have forgotten their passwords are able to send a password reset message to their email in order to change their passwords.