

Feature Design and Implementation

AI Model Choice

At the start of the project, we decided to use an AI model that would run within our machine. We believed this would be a good learning opportunity, as in previous projects, we only used APIs from OpenAI. Therefore, we discarded any paid option such as OpenAI or any other tool that would require a simple API call.

After some research, we decided to use GPT4All, as it is free, has multiple models to choose from, and has a JavaScript package that would help integrate with our current code base.

For the model itself, we opted for "mistral-7b-openorca.gguf2.Q4_0.gguf", trained by Mistral AI. The decision to choose this model was simple and direct as GPT4All advertised it as the best overall fast chat model in their catalogue.

More information about GPT4All and our model can be found on this [link](#).

Model Usage

The model will analyze and decide if the message sent by the users contains inappropriate content and give a brief justification for why the message was flagged. Based on the AI output, the code will perform the necessary actions to filter the message for other users, notify the sender why their message was inappropriate, and prevent the user from sending messages if they were flagged inappropriate multiple times.

Implementation / AI Integration

Changes in DB Schema

To keep track of how many inappropriate messages each user sent, we initialized a new attribute, `inapContentAttempts`, for all users in the DB. This attribute will be incremented whenever the AI flags a user's message as inappropriate.

GPT4All Class

Similar to what is done in Database.js, we decided to create a new class responsible for our feature's AI functionality. It is a simple class whose only attribute is the model. It has two functions: `loadModel`, which loads the model whenever we initialize an object of the class, and the function `checkIfMessageInappropriate`, which is called when there is a need to analyze the user messages.

checkIfMessageInappropriate

The query used for prompting the model specifies very explicitly what the model's output should look like so we can parse it and obtain useful information from it.

In case the query is not inappropriate, it should only output "No." However, if the query is considered inappropriate, it should also include a justification in a format like "Yes. <Justification>."

The model's output is parsed, and the function returns an object with a boolean field, `isInappropriate`, and, in case of detection, a `justification` field.

Rest API: Check If Message Inappropriate

In order to follow project requirements, we created a post API under `/api/check-message` that receives a message string, processes it using the GPT4All class and returns the object received from `checkIfMessageInappropriate` function.

Modifications on WebSocket Backend

The primary logical modifications for implementing this feature involved WebSocket communications, as every user message is checked on send. Therefore, we decided to implement a type within the objects sent by the server on the "message" socket trigger/event. We set 3 possible types for those objects:

- "chat": Just a standard message
- "inappropriate-alert": Notify the user that its content was flagged
- "restricted-alert": Notify the user its account is prohibited from sending messages

On every message sent by a user, we first check if the user can still send messages; if their profile has already been restricted, the user will receive a "restricted-alert" and the message will not be broadcast to other users.

If a user is allowed to send messages, the AI API `/api/check-message` be called to decide if the content is inappropriate. If AI does not flag the content, the message will be forwarded to all other users as usual. Otherwise, we will update the db incrementing the number of inappropriate messages in the sender's profile, send an ""inappropriate-alert" to the sender, and, if it is its final violation, an "restricted-alert". We also modify the content of the message, flagging it as inappropriate, and broadcast it to other users so they know that the sender was inappropriate and why the message was flagged.

Major backend changes happened on `GPT4All.js` file, and `server.js` (starting on lines 114 and 345)

Modifications WebSocket of Front-End

The modifications in the front end aim to notify the user if their message was flagged or their profile was restricted. The modifications also properly updated the sender UI, as the sent messages are included in the UI before they are processed in the back end.

For the object with the type "chat," the previous actions performed by the "message" trigger were performed as this is simply a new message on the chat.

If an "inappropriate-alert" is received, a pop-up notifying the user that its message was flagged is shown, and the flagged message is updated with the filtered content. If a "restricted-alert" is received, a pop-up alerting the user that their profile is restricted is shown, and the input for sending a message is disabled. Since the message is not broadcasted on the backend in the case of a restricted user, the restricted user cannot override this restriction of sending messages by modifying the attributes of the HTML tags via DevTools.

Since our implementation still allows restricted users to read chats, if a restricted users log in and try to send a message, the message will not be broadcasted to other users, a "restricted-alert" will be received, the input box will be disabled, and the sender UI will be updated not to contain that message, as the check for the sender permission to send messages happens after the sender UI has already been updated with the new message.

Major front-end changes happened on `app.js` file, around `class Room` line 363 and `main` function line 473