MemeStream: A Dedicated Meme-Sharing Platform with AI-Powered Content Validation and Real-Time Engagement

Abstract. Despite the ubiquity of memes, social media often treats them as standard images. This paper presents *MemeStream*, a dedicated meme-sharing platform utilizing MVC architecture. The system integrates React 19 and ASP.NET Core 9 with PostgreSQL and SignalR. A central feature is the AI validation layer, powered by Google's Gemini API, which performs multimodal analysis to ensure content quality before publication.

The platform implements a multi-factor feed ranking model balancing recency, social ties, and engagement. User participation is quantified via *LaughScore*, a weighted metric categorizing progression based on unique interactions. SignalR facilitates real-time features, including messaging and instant notifications. This study demonstrates the effective integration of AI-driven validation, gamification, and real-time communication within a domain-specific framework.

Keywords: Memes, Social Platforms, MVC, AI Validation, Real-Time Messaging, Gamification, Feed Ranking

1 Introduction

Memes were originally nothing more than inside jokes being passed from one online circle to another. Today memes represent what is essentially the Internet's common language. Everyone uses memes; people use them to joke about something, to simply make someone laugh, or as an effective way to make a point [1, 2]. You'll hear them referenced by news anchors, see them pop up in family group chats, and every once in a while, a meme even sparks a real-world movement something the original meme creators probably never saw coming [3, 4]. Countless memes are thrown around on social media platforms such as Reddit each and everyday, and some Instagram accounts with no other content but memes attract followings that are equivalent to a news account [5]. Younger generations are particularly fond of memes and find them more appealing than plain old texting to communicate an idea [6]. However, there is much more to memes than just being funny. Research has demonstrated how memes affect political views, aid groups in coping during difficult times such as the COVID-19 pandemic, and provide a voice for communities to take action [7, 8].

Even though memes are such a big deal, most social platforms still treat them like any other image upload. There's nothing built in for how fast memes move, how often they get remixed, or the weird in jokes that make them work.

So, if you're serious about making or sharing memes, you usually have to hunt down private groups or dig through hashtags to find your people.

Frankly, missing features aren't all we have to worry about here. If a platform cannot recognize a funny meme from a random photo, then how do you keep a community's standards high? Trendy memes go by so quickly that they get lost in feeds built for news and updates, and as a result large numbers of likes do little to tell us if the meme was funny or simply an obligatory share to get some attention.

That is where MemeStream comes in. MemeStream is created specifically for meme culture. The entire process for users to upload, recommend and recognize memes were all developed with a complete understanding of what users consider to be valuable when interacting with memes in social media. MemeStream solves three main problems: Ensuring the things you upload are actually memes, recommending the right memes to appear in users' feeds and rewarding creators of popular posts that go viral.

The key aspect of MemeStream's design is its use of a Model-View-Controller (MVC) architecture. The MVC architecture maintains separate components for the data model, business logic, and user interface. This separation makes maintenance of this complex system significantly easier than other systems which include AI, WebSocket connections, and dynamic feed changes [9, 10]. This type of structure has been used for many years in developing web applications [11].

The web stack used in MemeStream is one that is well established: Entity Framework Core handles database operations with PostgreSQL, ASP.NET Core 9 is used on the backend for speed and SignalR support. On the frontend, React 19 takes care of the interface with Tailwind CSS for easy styling, React Query is used to manage server state, and Vite for fast builds.

The meme-detection system is perhaps the most important part. Before anything can be posted, the Gemini API from Google will check the uploaded content (either text, image, or both) to determine if it is a meme or not. The goal isn't to censor, but to keep quality up and give users quick feedback.

The way that the feed operates is unique as well. New posts from friends will be placed at the top of the feed. However, the algorithm will also add in posts from new creators and meme styles. Since memes don't stay relevant for long, the system puts a premium on what's new.

For recognizing creators, there is a feature called LaughScore. Each reaction from a new user will give you five points, each share will give you three points and each comment will give you two points. The setup encourages the creation of content that people engage with in a meaningful way rather than creating a high amount of fake engagement. On top of that, SignalR provides the real-time capabilities that allow users to receive read receipts, typing indicators, immediate messaging capabilities whether they are communicating privately or in a group, and notifications that immediately appear when a friend request is made, someone leaves a comment or reaction on one of your posts, etc.

There are five key contributions of this work. Firstly, it demonstrates how the architecture of MVC can be adapted to meet the requirements of various so-

cial platforms. Secondly, it demonstrates how large language models can perform content validation without requiring a customized data set. Thirdly, it introduces algorithms for feeds that are specifically designed for memes. Fourth, it introduces an engagement-based ranking system that captures the type of subjective resonance that is missed by simple vote counts. Fifth, it introduces practical examples for incorporating real-time functionality into MVC applications using SignalR.

2 Related Work

A number of disciplines study memes in some way. Cultural theorists study how memes create commonalities in the understanding of events and ways to form identity and community [1]. Political scientists study how memes can be used to initiate public dialogue around certain topics, particularly with groups that may be overlooked by traditional news outlets [4]. Platform researchers study how memes move across platforms and how various user bases and design aspects of platforms affect how memes are transmitted [5].

While content moderation researchers have done well in developing technologies to identify and remove hate speech, misinformation and other violations of social media policies, there is little discussion of identifying or ranking the positive or quality content that remains after the removal of negative content. In addition, while many social media platforms continue to promote metrics such as "likes" and "followers," these metrics do not measure whether users genuinely value a piece of content or whether they click through habit or due to social influence. Additionally, these metrics do not assess whether an individual meme is engaging.

From a technical viewpoint, most web applications use the MVC architecture to organize their application's components [10, 12]. Frameworks like SignalR makes it much easier to implement features requiring real time interaction like using WebSockets and tracking who is currently logged into an application [13]. However, the majority of the work done in web applications continues to focus on removing problematic content rather than ensuring that the content is high-quality. Large language models though, are starting to show real potential for sorting and categorizing content [14].

This is where MemeStream comes in. Instead of only focusing on what cannot be allowed, it uses AI to check if new uploads actually meet a basic standard of meme quality. Instead of copying your typical news feed algorithms, MemeStream's feed pays attention to the timing and unique patterns of memes. Its ranking system doesn't just count clicks or likes it brings forward the real effort people put into making these memes. Ultimately, MemeStream fills a gap in both research and practice by caring about what makes memes work, not just what gets them kicked off the platform.

3 Methodology

3.1 System Architecture

Have a look at Figure 1. It shows MemeStream's three-tier setup; splitting things into presentation, application, and data layers.

The architecture includes five main parts:

1. Frontend Application (Presentation Layer)
2. Backend API Server (Application Layer)
3. Database Server (Data Layer)
4. External Services (Cloudinary, Gemini API, SMTP)
5. Real-time Communication Layer (SignalR Hubs)

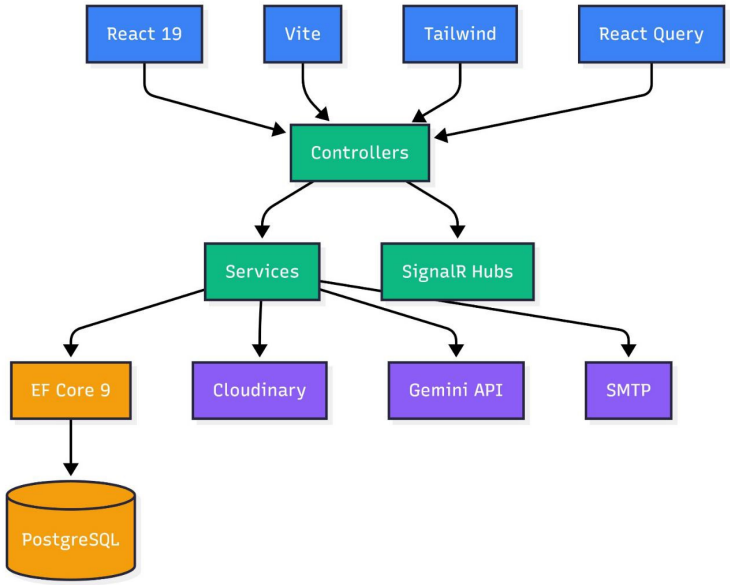

Fig. 1: MemeStream's three-tier architecture

Presentation Layer The Frontend is built using React (Version 19.1.0) along with the build tool Vite for development. It is styled using the utility first approach of Tailwind CSS & its component library called Daisy UI. This is where the user performs all actions related to Browsing Content, Uploading Memes, Sending Messages, Logging In, Managing Profiles etc...

The application relies on WebSockets to achieve Real Time Updates. Upon User Action, the Frontend will send an HTTP Request to the Backend. Authentication Tokens are stored locally on the Users Device and Attached to API Calls. All Media Files are sent directly to Cloudinary.

To manage state, the React Context API handles things that need to be global, while React Query keeps server data fresh and cached. There are also managers watching the connection status in real time.

Application Layer The backend is based on the MVC architecture that runs on ASP.NET Core 9.0 [15, 12].

There are three primary components to the API server:

- **Controllers:** Responds to HTTP requests and point them to the appropriate services. These include social interactions, messaging, notifications, meme recognition, user administration, and post processing.
- **Services:** Includes the essential business logic like EmailService (sends emails via SMTP), LaughScoreService (calculates engagement scores), NotificationService (manages alerts), MemeDetectionService (connects to Gemini API), and FeedGenerationService (implements the feed algorithm).
- **SignalR Hubs:** Communicate in real time [13]. While ChatHub handles group and private messaging, NotificationHub sends alerts to clients.

Data Layer We utilize Entity Framework Core as our ORM, which stores all data within a PostgreSQL database. Our database is structured with normalized tables to organize user information, posts, interactions, and logs about what has happened in the system.

Our database schema includes tables that cover users, posts, comments, reactions, friendships, friend requests, messages, groups, and notifications. As a way to increase performance and allow us to handle large amounts of users simultaneously, we implemented connection pooling. Additionally, we have indexed the most commonly searched columns by users.

3.2 Integration with External Services

Cloudinary Integration Cloudinary provides both media storage and delivery services for our application. Before any files are uploaded, the client-side checks the file's size (max 10MB) and type (JPEG, PNG, GIF, WebP) and authenticates the upload. After Cloudinary returns the URL of the uploaded file (which is a CDN-backed URL), the CDN-backed URL is stored in the database, allowing users to access their media immediately.

Google Gemini API Integration We use the Gemini API [14] for AI-based analysis of text and images. In order to avoid hitting rate limits and/or experiencing downtime, we implemented retry logic with an exponential backoff and also have fallback options available. The API keys are managed using environment variables.

Email Service Integration MailKit and SMTP provide email service integration for our application; MailKit and SMTP handle all aspects of email including, but not limited to, password resets, account verification, notifications and security alerts. The verification email sent to the user includes a secure token and password reset links expire after a set amount of time.

3.3 Algorithm Workflow and Validation

As depicted in Figure 2, our platform follows the same workflow when developing algorithms. Each algorithm will follow this workflow to ensure that the algorithms are developed with care, and works correctly.

Fig. 2: Algorithm development and validation process

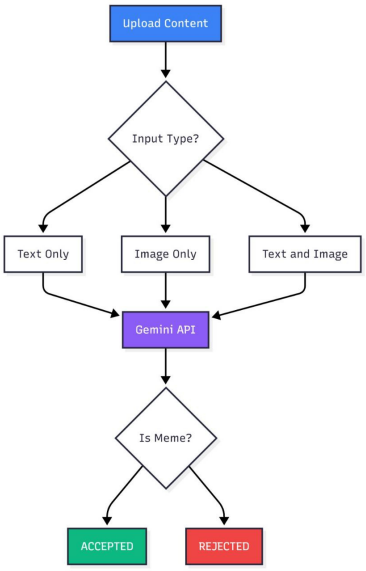

Fig. 3: Meme detection algorithm flowchart

Meme Detection Algorithm Before anything is published, this one intervenes. What is its job? Determine whether the image we are viewing is a meme. This algorithm can handle text, images, or a combination of the two.

Inputs: You can give it text, an image URL, or a language hint none of those are required, but you need at least text or an image.

Steps:

1. Decide how to analyze based on what you've got text, image, or both.
2. If there's text: figure out the language, create a context-aware prompt, send it to the Gemini API, and scan the response for humor, cultural references, and classic meme signals.
3. If there's an image: pull it from Cloudinary, convert it to base64, shoot it over to Gemini Vision, and extract any text (with OCR), spot visual elements, and look for meme templates.
4. If both text and image: run both analyses and see how well the results match.
5. Finally, everything is put together to check if the meme probability score crosses a set threshold, and make a final yes/no call.

Outputs: You get a structured answer with the classification (meme or not), confidence score (from 0.0 to 1.0), humor score (0–10), sentiment, detected language, any cultural references found, and visual analysis info if there's an image.

Take a look at Figure 3 for the full decision flow.

Scoring: The final score combines text, image, and alignment components with weighted coefficients:

$$S_{total} = w_1 \cdot S_{text} + w_2 \cdot S_{image} + w_3 \cdot S_{alignment}$$

where $w_1 + w_2 + w_3 = 1.0$.

Feed Generation Algorithm This algorithm is responsible for what users see in their feed.

Inputs: A list of posts you've already viewed, options for pagination and your User ID.

Steps:

1. First, grab posts from your friends and mix them into a larger pool, excluding the posts you have viewed.
2. Next, evaluate each post based on a number of criteria: How much engagement has occurred (using log scaled values for comments/reactions), how recently the post was created, is the post from someone you're connected with, is the post visually appealing (has a photo) and if the poster is new, add a "discovery" score.
3. After evaluating each post, the algorithm ensures you do not view an excessive amount of liked-posts or posts from the same creator again and again.
4. Finally, the posts are cached, ranked in order of score and paginated.

Outputs: A sorted list of posts for your current page, paginating information, a timestamp used to help control the caching of the posts.

Scoring formula:

$$S(p) = S_{base} + F(p) + T(p) + E(p) + M(p) + D(p)$$

where S_{base} is a base score, $F(p)$ gives extra points for friend content, $T(p)$ rewards recency (from +40 for posts under an hour old to -20 for posts over 30 days old), $E(p)$ calculates engagement as $\log(\text{engagement} + 1) \times 6$, $M(p)$ adds points for image posts, and $D(p)$ adjusts for content diversity.

LaughScore Calculation Algorithm To keep things fair, LaughScore only counts each user once per post. That way, users are not rewarded too highly for participating in multiple ways on a single post.

Inputs: - The user's ID and all their posts.

Steps:

1. For every post, count how many unique people reacted, shared, or commented.
2. Give out points like this: two for each unique commenter, three for each unique sharer, and five for every unique reactor.
3. Add up the points for each post.
4. Sort the total into one of eight tiers ranging from "No Laughs Yet" (0 points) all the way up to "Meme Legend" (over 1000 points).

Outputs: - The overall LaughScore, the tier you landed in, and a breakdown showing where your points came from.

Formula:

$$LS(u) = \sum_{p \in P(u)} (5 \cdot R_u(p) + 3 \cdot S_u(p) + 2 \cdot C_u(p))$$

where $R_u(p)$, $S_u(p)$, and $C_u(p)$ count distinct users who reacted, shared, or commented on post p. Each user contributes at most once per post per action type.

3.4 Technology Stack

Table 1 shows the technologies that were used in MemeStream.

3.5 Testing and Validation

Instead of relying on automated testing, we used a user research and manual checks to evaluate the project.

User Study Our user study was a four-week study that involved 50 participants. The user study was our primary means of validating whether or not the platform functioned as intended by actual users. Automated testing, while valuable, is limited; therefore, the user study allowed us to see how users registered, uploaded memes, communicated socially, and interacted with each other in a real world environment to better understand the true nature of the platform.

Table 1: Technology Stack Overview

Component	**Technology**
Frontend Framework	React 19.1.0 with Vite
Frontend Styling	Tailwind CSS, DaisyUI
Backend Framework	ASP.NET Core 9.0 (MVC)
Database	PostgreSQL with Entity Framework Core 9.0
Real-time Communication	SignalR 9.0.6
AI Content Analysis	Google Gemini API
Media Storage	Cloudinary
Email Service	MailKit with SMTP
Authentication	JWT Tokens
Deployment	Docker, Azure App Service

Algorithm Evaluation Based on lessons learned from our user study, we evaluated the efficacy of our algorithms using precision, recall, and F1 scores to evaluate the success of our meme-detection algorithm against evaluations made by human moderators. In addition, we measured diversity in the feed, user engagement time, and click-through-rates. Finally, we correlated the Laugh Score with user satisfaction, based on user feedback received via an exit survey.

Monitoring Performance Throughout the user study, we continuously monitored the performance of the system. To do this, we utilized a variety of system-monitoring tools to monitor things such as system resources used, response times, and the number of active users at any given time.

3.6 Overall Data Flow

The dataflow diagram is shown in Figure 4.

3.7 Ethical Considerations

We maintained ethical standards throughout the project.

User Consent and Privacy All participants provided informed consent for their participation in this study. Users were able to delete their data from the system at will; we also included an anonymized option for participant data when available, and provided our privacy statement clearly and in simple terms.

Algorithmic Fairness To ensure transparency of our AI, we utilized explainable AI methodologies throughout our research. Bias prevention was integrated into each of the models we developed and tested to determine fairness in the use of our system by all demographic categories. In addition to bias testing, we allowed users to dispute any moderation that occurred in the use of our system.

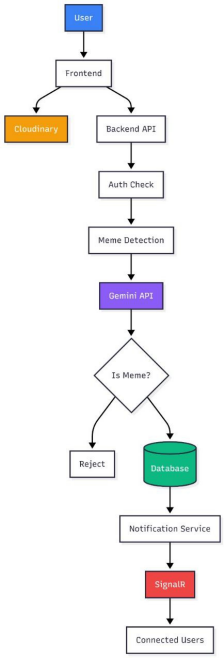

Fig. 4: System data flow diagram

Content Moderation Our content moderation was based on a dual-process model of AI moderation and human moderation. In order to develop a culturally sensitive meme-detection capability, we implemented safeguards to prevent harassment/hate speech, and we established community guidelines for our user-base.

Regulatory Compliance We complied with all applicable regulations including but not limited to: COPPA for minors, GDPR for users located within Europe, and we ensured compliance with WCAG 2.1 accessibility standards. As we are aware that ensuring compliance is an ongoing responsibility we continue to implement industry-leading security measures through regular audit processes.

4 Details of Implementation

To create a viable project, there was a need to strike a balance between ease of development, high-performance and rapid development. Many of the decisions made were based on actual trade-offs we encountered during the development phase. The frontend is implemented as a Single-Page Application (SPA) using React 19.1.0. Vite allows for rapid builds and fast reloads in development and produces very small bundles for production. Tailwind CSS and DaisyUI are being

used for styling to allow for rapid prototyping and avoid spending too much time creating custom styles for each component. The backend uses ASP.NET Core 9.0 to organize the middleware pipeline (such as authentication and error handling), services (using Dependency Injection) to keep services loosely coupled and Entity Framework Core 9.0 to map C# objects to PostgreSQL tables which provide high availability, performance and features such as native JSON support. SignalR 9.0.6 is being used for real-time features in the application. SignalR establishes WebSockets when available and will fall back to alternative transports if necessary. ChatHub is responsible for private and group messaging, including sending typing indicators and read receipts. NotificationHub is responsible for notifications related to reactions, comments, friend requests and others.

As much attention as possible has been focused on those areas of the application that have the most impact on user experience. The ability to scroll virtually through thousands of posts and still be able to view the content quickly and smoothly is a good example of this. Composite database indexes were created to speed up search functionality and common queries. Lazy loading was also enabled so images do not begin loading until the user views them. Compressing responses reduces bandwidth usage. Connection pooling is used to ensure SignalR does not consume excessive resources even under heavy traffic conditions.

Security was not overlooked either. Tokens with a valid expiration date are used for authenticating users via JWT. Role-based policies are enforced at the controller level to define permissions per role. Every bit of user input gets validated before anything happens with it. Rate limiting protects heavy endpoints like meme detection from getting spammed. To guard against XSS, we escape user-generated content before displaying it, and file uploads are tightly checked for allowed MIME types and size.

5 Evaluation and Results

5.1 Experimental Setup

We had fifty people participate who were all different ages from eighteen to thirty-five, and were from two completely different areas: internet meme communities and computer science students at the local university. Each person had their own tech background, and their own way of using the internet as well. We did not have each participant perform some artificial task, but instead let them use MemeStream however they wanted to; posting, replying, messaging, or even simply browsing through their feed as they would normally do on their personal computers.

5.2 System Performance Metrics

The platform held up well under real-world loads. You'll find the main performance stats in Table 2.

From the table we can see that performance was excellent and remained stable upto 1000 concurrent users, after which the response time began to degrade gradually.

Table 2: System Performance Metrics

Metric	Description	Value
API Response Time	Average response time for API requests	45 ms
Feed Generation Time	Time to generate personalized feed	120 ms
Real-time Message Latency	End-to-end message delivery latency	<100 ms
Meme Detection Time	Average AI analysis time per upload	1.2 s
Concurrent Users Supported	Maximum without significant degradation	1,000
Uptime	Platform availability during study	99.8%

5.3 User Engagement and Satisfaction

We were pleased to learn that participants became much more active than what we had originally anticipated. Averages of participant activity can be seen in 3.

Table 3: User Engagement Metrics (Per User, 4-Week Average)

Activity Type	Average Count
Posts Created	12.4
Reactions Given	87.2
Comments Posted	23.1
Shares Performed	15.6
Messages Sent	42.8
Friend Requests Sent	8.3
Daily Active Minutes	28.5

Exit surveys provided satisfaction ratings across several dimensions (Table 4).

Table 4: User Satisfaction Ratings (1-5 Scale)

Aspect	Description	Average Rating
Overall Satisfaction	General platform satisfaction	4.3
Meme Detection Accuracy	AI classification correctness	4.1
Feed Relevance	Content matching user interests	4.0
Real-time Messaging	Speed and reliability of chat	4.6
LaughScore System	Motivation from point system	3.8
User Interface	Ease of use and visual design	4.2
Platform Performance	Speed and responsiveness	4.4

Interview feedback showed that people liked the focused vibe of just memes. The quick, chatty pace of meme groups stood out too, and that is clear from how high they rated the real-time messaging. However, when it came to LaughScore,

opinions split. Some found the tier system motivating. Others wanted a clearer breakdown of how the points actually worked. More than a few wanted better ways to tweak and customize their own feeds.

5.4 Algorithm Effectiveness

Algorithm-wise, engagement and variety both improved when the feed wasn't just straight up chronological. Users seemed happier, and the scores reflected that. LaughScore, with its unique-user weighting, rewarded real participation, not just empty clicks.

6 Discussion and Limitations

Our findings supported the theory that meme platforms require their own specific features rather than merely copying those of larger social media platforms. A good example of this was the meme recognition system. Large language models recognized meme categories well, and did not require additional training pipelines. The multi-factor ranking mechanism also performed well, as users spent longer periods of time interacting with the platform, and reported higher satisfaction rates when the feed included fresh, relevant, and diverse content. The algorithm was able to successfully monitor new posts, while avoiding stale or redundant content by monitoring recency and diversity.

LaughScore met our expectations. The 5-3-2 point system used to evaluate comments, shares, and reactions provided an appropriate weighting of valuable interactions. Unique users per post allowed LaughScore to mitigate users' ability to game the scoring system by engaging in spammy actions. Interviews clearly showed that users desire more transparency surrounding how scores are calculated. Real-time features received high ratings as timing is everything in meme culture. The instant messaging and alert capabilities of the platform allowed users to join into conversations at the moment they went viral.

However, there were limitations. Much of the content collected was from a western perspective; therefore, memes from other cultural perspectives did not have equal representation. The four week window used to collect data was limited and as such we were unable to track longer trends. Additionally, our participant group was primarily composed of users from North America and Europe who were sourced from online communities and university circles and therefore do not represent the entire global meme community.

Costs are another issue. At scale, Gemini API charges start to pile up since every upload costs money. WebSocket reliability varies from network to network, and some company firewalls just break long-lived connections. Even with measures to mix things up, personalization can still create echo chambers. And gamification no matter how well you set it up always nudges user behavior in ways that aren't always great for the community. These are ongoing challenges that need attention.

7 Conclusion and Future Work

MemeStream shows how niche communities are able to achieve more on platforms created specifically for them rather than forcing themselves into a generic social media model. The data shows that it is successful. The meme detection system demonstrated that large language models can identify the correct type of content and not simply flag items to be removed. Users preferred to have an area developed strictly for memes, as evident in the user satisfaction ratings.

MemeStream has left a legacy of best practices that can be used by others. The MVC structure allowed for integration of WebSockets, dynamic feeds, and AI without creating clutter. Gemini worked well due to careful prompting and adequate error handling, there was no need for customized training. The feed algorithm presented a good example of a method to create a fresh, diverse feed based on time weight, relationship weight, and diversity check. LaughScore's system counting unique users and not just likes, solved the old problem of measuring reach without real resonance. Considering the complexity of this application, the stack was sufficient. The application's design was straightforward, simple to maintain, and scalable if necessary, without requiring additional complex infrastructure.

Future research should focus on detection accuracy outside of Western countries that were not included in the study, as accuracy may vary across different cultures. In this study, users expressed a desire for more customization options for their feeds (e.g., ability to limit the amount of content from friends vs. discovery). Additionally, users stated they would like LaughScore to provide better clarity regarding how their points are being generated (as this will help them understand how they are earning their points). Lastly, considering the majority of people access their memes through their phones, developing mobile applications for LaughScore is essential.

At the end of the day, the role of algorithmic accountability and community governance will be the key factor as social media platforms grow. Algorithmic decisions about ranking content affect what people see and what gets traction. It is good practice to implement processes at the time of initial use so that users may make appeals or understand the decision-making process; this should not be an afterthought.

There are many other aspects to consider beyond memes, but the issue of adapting platforms for specific types of content (e.g., crafting, academic papers, cooking classes) can be very beneficial for communities. However, such communities will require systems that allow them to continue to function in their own way, whereas mainstream social media will continue to pursue the largest possible audiences.

Acknowledgment

Thank you to all participants in our user study for their valuable feedback. We also appreciate the support from our University's Department of Computer Science and Engineering for providing us with the necessary computing resources.

References

1. Shifman, L.: Memes in Digital Culture. MIT Press (2014).
2. Wiggins, B.E., Bowers, G.B.: Memes as genre: A structurational analysis of the memescape. New Media & Society 17(11), 1886–1906 (2015).
3. Ahmed, S., Jaidka, K.: The political potency of memes: A longitudinal analysis of visual rhetoric in online political discourse. Journal of Communication 74(1), 45–67 (2024).
4. Moreno, A., Navarro, C.: Memes as a vehicle for political engagement among youth: A case study of the 2020 U.S. elections. Social Media + Society 7(2), 1–14 (2021).
5. Tommasini, D., et al.: The ecology of memes: Platform affordances and the spread of internet culture. First Monday 28(3) (2023).
6. Mihailidis, P., Viotty, S.: Spreadable spectacle in digital culture: Civic media and the new attention economy. American Behavioral Scientist 64(8), 1121–1137 (2020).
7. Flecha, R.: Memes as coping mechanisms during the COVID-19 pandemic: A global comparative study. International Journal of Environmental Research and Public Health 18(16), 8567 (2021).
8. Akram, U.: Memes, mobilization, and collective action: From online laughter to street protest. Social Movement Studies 20(5), 589–605 (2021).
9. IEEE: Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Std 1471-2000 (2001).
10. Krasner, G.E., Pope, S.T.: A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming 1(3), 26–49 (1988).
11. Bucanek, J.: Learn Objective-C for Java Developers. Apress (2009).
12. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley (2003).
13. Microsoft: ASP.NET Core SignalR Documentation. https://learn.microsoft.com/en-us/aspnet/core/signalr (2024).
14. Google: Gemini API Documentation. https://ai.google.dev/gemini-api/docs (2024).
15. Reenskaug, T.: Models–Views–Controllers. Technical note, Xerox PARC (1979).