Design and Implementation of a Scalable Social Network Platform Using Microservices - Technical Report

Johan S. Ebratt, Paola B. Cuellar
Systems Engineering Curricular Project
Universidad Distrital Francisco José de Caldas
Emails: jdebratts@udistrital.edu.co, pacuellarb@udistrital.edu.co

***Abstract*—Modern social media platforms face significant challenges related to scalability, security, and content relevance. In response, this work proposes the design and implementation a modular social network system built upon a microservices architecture, integrating cloud storage, identity verification, and interest-driven user interaction. As a result, the project defines a scalable and modular platform supported by a robust distributed database architecture that ensures high availability, consistent and reliable data management, and the flexibility to grow with user demand.**

***Index Terms*—Social networks, Microservices architecture, Distributed databases, Cloud infrastructure, Identity verification, Scalable systems.**

I. Introduction

Social media has revolutionized communication, providing platforms where individuals, organizations, and governments engage in real-time information sharing. With billions of daily active users generating immense volumes of structured and unstructured data, ensuring system scalability, performance, and data consistency becomes a complex challenge. Traditional monolithic applications often fall short in handling this scale due to their inflexible structure and limited resource isolation.

The current project aims to design and implement a scalable and modular social media platform based on the architectural model of X (formerly Twitter). By employing microservices, the platform promotes separation of concerns, continuous deployment, and independent scaling of services. Additionally, combining SQL and NoSQL databases allows the system to benefit from both strong consistency and horizontal scalability. SQL is primarily utilized for critical and structured data such as user profiles and relationships, while NoSQL databases like MongoDB handle high-throughput operations such as feed generation and post interactions.

This hybrid approach is particularly important for balancing read-heavy workloads and write-intensive scenarios. The use of container orchestration with Kubernetes and caching strategies via Redis further enhances performance and reliability. This report outlines the architectural design, experimental validation, and performance analysis of the system, providing insights for future applications of similar models in real-world distributed environments.

II. Literature Review

Several researchers and industry leaders have highlighted the limitations of monolithic architectures in the context of scalable web applications. Fowler [1] and Newman [2] stress the importance of microservices for achieving decoupled, maintainable systems. These architectural styles have gained popularity in social platforms due to their modular nature and ease of horizontal scaling.

The use of hybrid databases is another extensively explored area. Cattell [3] discusses the evolution of scalable NoSQL systems, noting their efficacy in managing semi-structured data and large-scale workloads. Stonebraker and Cattell [4] contrast relational and non-relational databases, arguing that the best performance is often achieved by choosing the right tool for the right data type.

In the context of social networks, Ahmed et al. [5] explore performance bottlenecks and the trade-offs between consistency and availability. They emphasize the need for distributed design patterns and data sharding. Brewer's CAP theorem [6] forms the theoretical foundation of such trade-offs, which are directly addressed in this project.

Furthermore, companies like LinkedIn and Netflix have published white papers on their use of polyglot persistence—employing different types of databases for different services [7][8]. These real-world implementations validate the hybrid model used in this project. Research also suggests that content moderation and recommendation systems benefit from scalable NoSQL models due to their flexible schemas and integration with machine learning pipelines.

III. Background

The development of scalable and resilient applications in the modern digital landscape requires a strong foundation in distributed systems. This project embraces key architectural principles such as fault tolerance, asynchronous communication, horizontal scalability, and service decoupling, which are essential for supporting dynamic workloads and ensuring continuous availability.

Foundational Concepts

The system design is guided by the principle of separation of concerns, achieved through a microservices architecture. Each service encapsulates a specific domain — such as users, posts, or authentication — and interacts with others through stateless communication using RESTful APIs. Asynchronous data flows and eventual consistency are preferred in non-critical paths to reduce latency and improve responsiveness.

Security and authentication are handled using OAuth 2.0 and JWT (JSON Web Tokens), enabling secure, stateless sessions across services. Each request includes a signed token that validates the user's identity and authorization scope without relying on centralized session storage.

Infrastructure and Orchestration

All backend services are containerized using Docker, allowing consistent deployment across environments. Kubernetes is used for orchestration, providing automated service discovery, load balancing, self-healing, and horizontal scaling. These features simplify infrastructure management and enable the system to scale dynamically according to demand.

An API Gateway serves as the single entry point for the frontend, enabling routing, authentication, and rate limiting at the edge. This abstraction shields internal services from direct exposure and centralizes cross-cutting concerns.

Frontend Integration

The frontend is built using React and communicates exclusively through the API Gateway. It acts as a client for the distributed backend, consuming data through well-defined endpoints. This decoupling of frontend and backend layers ensures that changes in the UI or logic can evolve independently, supporting a modular and iterative development process.

Data Management Strategy

A hybrid persistence strategy is employed, reflecting the different characteristics of the data handled by the platform. Structured data — such as user profiles, authentication metadata, and follower relationships — is stored in PostgreSQL, leveraging its strong relational model and transactional guarantees. Conversely, unstructured and dynamic data — such as posts, comments, and media attachments — is stored in MongoDB, taking advantage of its schema flexibility and document-oriented model.

To improve performance and reduce query pressure on primary databases, Redis is used as a high-throughput in-memory cache. It stores frequently accessed information, such as trending content and user sessions, contributing to faster response times and improved user experience.

Motivation for Architectural Decisions

This architecture mirrors the proven design patterns of large-scale social platforms, where modularity, fault isolation, and elastic scalability are crucial. By clearly separating responsibilities across services and databases, the system reduces internal coupling, facilitates independent scaling, and simplifies maintenance and deployment workflows. These qualities are essential to meet the requirements of modern web applications, particularly those operating in high-concurrency environments.

IV. Objectives

The main objective of this project is to design and implement a scalable, modular social networking platform, using a microservices-based architecture and a hybrid data persistence strategy. The project aims to validate how such an architecture can improve system scalability, maintainability, and performance in environments with high user interaction and dynamic content generation.

General Objective

To architect and develop a distributed social media application that demonstrates modular service design, effective integration of heterogeneous databases, and resilient infrastructure capable of handling concurrent access and data-intensive operations.

Specific Objectives

- **Design and implement RESTful APIs** for core services including user management, post publishing, and personalized feed generation, enabling independent service evolution and deployment.
- **Integrate relational and non-relational databases**, leveraging PostgreSQL for structured user-related data and MongoDB for flexible storage of social interactions and multimedia content.
- **Evaluate and benchmark the performance** of read and write operations across services and storage layers, identifying potential bottlenecks and optimizing resource usage.
- **Implement caching mechanisms** using Redis to reduce latency in frequently accessed data and improve overall response times for end users.
- **Orchestrate services using containerization technologies** such as Docker and Kubernetes, to ensure scalability, fault isolation, and continuous delivery through automated deployments.
- **Simulate high-load scenarios** to assess system resilience, monitoring performance indicators such as throughput, latency, and fault recovery across the distributed infrastructure.

A. Scope

This project focuses on the architectural design and backend implementation of a scalable social networking platform, emphasizing modularity, performance, and maintainability. The scope is limited to core backend services and infrastructure components required to simulate a realistic yet controlled environment for experimentation and validation.

Included in Scope:

- **Authentication and User Management:** Secure user registration, login, and token-based session handling using OAuth 2.0 and JWT mechanisms.
- **Post Operations:** Functionality to create, delete, and retrieve user-generated posts, including support for structured and unstructured content storage.

- **Content Feed Generation:** Algorithmic generation of personalized content feeds based on user relationships and recent activity.
- **Media Storage and Caching:** Management of attached media files and implementation of caching strategies to enhance retrieval performance for frequent queries.
- **Containerization and Orchestration:** Deployment of backend services using Docker containers, with orchestration and scaling managed via Kubernetes.

Excluded from Scope:

- **Deployment in Production Environments:** The system is not intended for live deployment with real users; testing is conducted under simulated conditions.
- **Mobile Interfaces:** The implementation and design of native mobile applications or mobile-specific user interfaces are beyond the current scope.
- **Integration with External Advertising or Analytics Services:** The project excludes any integration with advertising platforms, user tracking systems, or third-party analytics solutions.

V. Assumptions

Several key assumptions were made during the development of this project, which guided the technical and architectural decisions. These assumptions helped define the system's scope, anticipate operational requirements, and ensure the feasibility of the proposed design. The main assumptions considered are as follows:

1) **High concurrency and continuous availability**
It was assumed that the platform would need to handle a large volume of real-time data flows, with many users connected simultaneously. As a result, the system was designed to be available 24/7, maintaining consistent performance and avoiding single points of failure.
2) **Importance of hashtag usage**
The use of hashtags was considered an essential feature for organizing and searching content, similar to other popular social networks. Therefore, the system needed to efficiently manage hashtags, including indexing and enabling fast search capabilities based on them.
3) **Support for multiple content types**
It was assumed that users would share not only text but also multimedia formats such as images, videos, audio, and GIFs. This diversity required the implementation of efficient mechanisms for storage, retrieval, and processing of different content types.
4) **High service availability**
It was assumed that all platform microservices must remain constantly available. This led to the design of a resilient architecture, with decoupled services, load balancers, and fault-tolerant mechanisms to ensure continuous system operation at all times.

VI. Limitations

Despite careful planning and a solid architectural foundation, several limitations affected the development and evaluation of this project. Recognizing these constraints is essential to accurately interpret the results and understand the scope of the implementation. The main limitations identified are:

1) **Limited data availability and continuity of services**
Although the system was designed to handle real-time data flows and ensure high availability, the actual environment did not allow continuous access to real-world data or sustained operation of all services. This restricts the ability to validate the system under realistic conditions.
2) **Scalability planning constraints**
While the architecture considers future scalability (both horizontal and vertical), technological limitations and the available infrastructure prevented the implementation and testing of actual scaling mechanisms.
3) **Use of simulated data**
The data used during testing and validation were artificially generated using tools such as ChatGPT. As a result, these datasets do not accurately reflect real-world user behavior, data diversity, or system load, which may affect the generalizability of the findings.
4) **Time constraints**
The limited time allocated for this project restricted the depth of development. Certain advanced features, performance optimizations, and real-world integrations could not be fully implemented or tested.
5) **Lack of expertise with high-load systems**
The team faced challenges due to limited prior experience with technologies for managing high volumes of data and ensuring system availability at scale. This influenced architectural decisions and constrained the range of solutions considered.

Limitation	**Description**
Data availability	Real-time or real-world data was not accessible; synthetic data was used instead.
Service continuity	Continuous operation of services could not be maintained in a real-world environment.
Scalability planning	Horizontal and vertical scalability were considered, but not implemented due to infrastructure and technology constraints.
Synthetic data use	Data was generated using tools like ChatGPT and does not reflect actual user behavior.
Time constraints	The project timeframe limited the depth and scope of development and testing.
Technology expertise	Limited experience with high-throughput systems affected architectural decisions and solution space.

TABLE I
Summary of Project Limitations

VII. Methodology

The development methodology for this project followed an iterative, research-driven, and exploratory approach. The first phase consisted of a technical investigation into how modern social networking platforms manage their internal architecture and data flow. A key insight from this analysis was the widespread adoption of hybrid data storage systems, where relational (SQL) and non-relational (NoSQL) databases are combined to maximize efficiency, scalability, and reliability.

A. Data Modeling and Storage Design

The need to separate different types of data storage arose from the nature of the data handled by the platform. Relational databases (SQL) are highly efficient when managing structured data with well-defined schemas and relationships — such as user profiles, authentication data, and account metadata. These systems provide strong consistency and transactional support, making them ideal for core user information.

However, social media platforms also deal with high-volume, unstructured, and semi-structured data, including posts, media files, comments, reactions, and follower relationships. This type of data often exhibits unpredictable growth patterns and complex access requirements. For instance, a single user might have millions of followers or interact with thousands of posts, creating scalability bottlenecks for traditional SQL systems.

To address this, the architecture adopted a polyglot persistence model, where:

- **User and authentication data** were stored in a PostgreSQL database due to their highly structured nature and need for ACID compliance.
- **Posts, media attachments, and follower relationships** were stored in a NoSQL database (such as MongoDB), which allows horizontal scalability, schema flexibility, and optimized reads for document-based access patterns.

This hybrid design is common among large-scale technology companies such as Meta, Twitter, and LinkedIn. These organizations leverage SQL for transactional consistency and NoSQL for elasticity and performance under massive loads. The approach allows them to maintain fast user experiences while handling billions of records in near real-time.

B. Database Iteration Process

With the hybrid model defined, several iterations of the database schema were performed. Each iteration aimed to refine the data model, ensuring entities were appropriately assigned to the correct storage system based on access frequency, mutation patterns, and expected growth. The result of this process is illustrated in Figure 1, which shows the architectural blueprint of the microservices system and storage layout.

Following this, the data entities were modeled and visually represented in the entity-relationship (ER) diagram shown in Figure 2. Entities assigned to NoSQL storage were grouped based on their semi-structured nature and need for flexible schema design. The separation not only enhanced performance but also reduced the coupling between modules, thus improving maintainability and scalability.

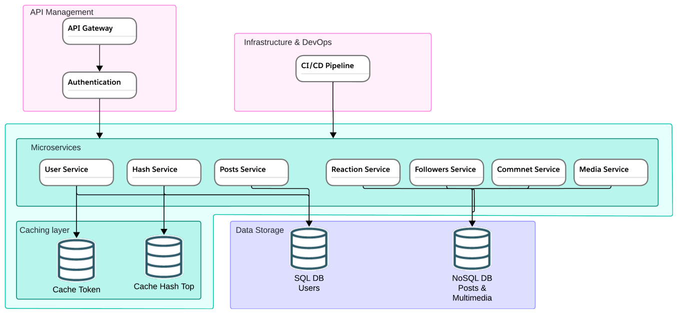


Fig. 1. System Architecture Diagram

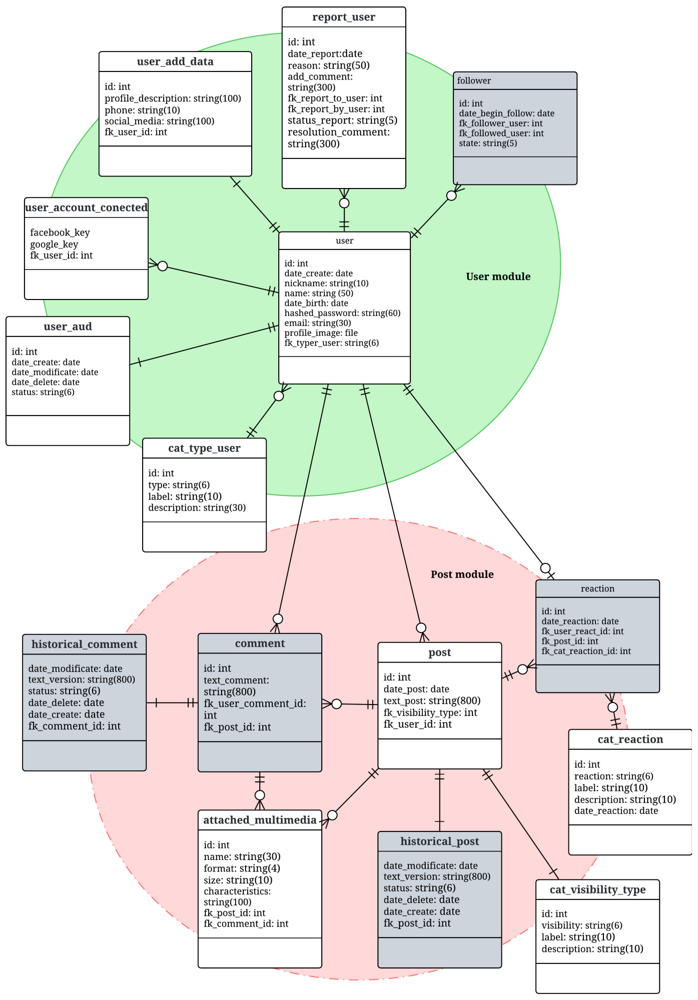


Fig. 2. Entity-Relationship Diagram with SQL and NoSQL Separation

C. Data Generation and System Evaluation

Due to the absence of real-world data, synthetic datasets were generated to simulate the behavior of the platform. This was achieved through the use of generative AI tools, such as ChatGPT, which provided randomized but contextually coherent user information, posts, comments, and reactions.

Although artificial, these datasets allowed for comprehensive testing of system functionalities and interaction patterns.

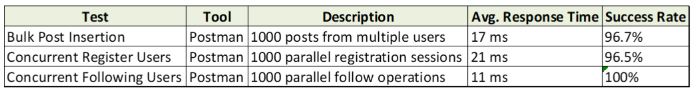
To evaluate system stability and performance, two main tools were used:

- **Postman**: Used to validate the behavior of APIs through manual and automated test suites.
- **Apache JMeter**: Used to simulate concurrent users and bulk data injection to measure response time, throughput, and resource usage under stress conditions.

The results provided insights into service latency, database bottlenecks, and potential improvements in service orchestration and caching strategies. This methodology ensured the system could be validated under simulated conditions approximating real-world use.

VIII. Results

To evaluate the performance and scalability of the system under concurrent operations, three load test scenarios were executed using **Postman Runner**: bulk post insertion, concurrent user registration, and concurrent follow actions. Each test simulated 1000 parallel operations. The average response time, total duration, and success rate were recorded. Table **??** summarizes the key metrics for each test.



Test	Tool	Description	Avg. Response Time	Success Rate
Bulk Post Insertion	Postman	1000 posts from multiple users	17 ms	96.7%
Concurrent Register Users	Postman	1000 parallel registration sessions	21 ms	96.5%
Concurrent Following Users	Postman	1000 parallel follow operations	11 ms	100%

Fig. 3. Success rate comparison across tests.

The *Concurrent Following Users* test yielded the most efficient results, with the lowest average response time and a perfect success rate. In contrast, both the *Bulk Post Insertion* and *Concurrent Register Users* tests showed slightly lower success rates and longer response times, likely due to the complexity of operations such as database inserts and validation logic inherent in relational models.

These findings suggest that the component handling user follow relationships is more optimized for parallel workloads, possibly supported by a more lightweight or efficient backend implementation (e.g., NoSQL or in-memory operations). Further analysis is recommended to assess scalability under increased load (e.g., 5000 concurrent users) and to identify potential bottlenecks in the registration and post services.

IX. Discussion

The results of the performance evaluation confirm the importance of a well-designed data distribution strategy in systems that handle high-concurrency workloads, such as social networking platforms. Specifically, the test on Concurrent Following Users—which likely relied on a NoSQL structure—demonstrated superior performance in terms of both speed and success rate. This supports the rationale behind employing document-oriented databases (e.g., MongoDB) for handling dynamic and unstructured data that requires high throughput and flexible schema design.

In contrast, operations involving user registration and post insertion, which were handled via SQL-based systems like PostgreSQL, showed slightly lower success rates and higher average response times. While relational databases offer strong consistency and transactional integrity—crucial for core data such as authentication and user metadata—their rigid schemas and limited horizontal scalability can introduce latency and contention under concurrent access.

These findings highlight a key design principle reinforced by literature and industry practice: using the right type of database for the right data domain. Structured data benefits from relational models, while interactions and fast-growing content streams are better managed with NoSQL systems.

Additionally, the results underscore the significance of having a well-structured data model. Poorly normalized tables, lack of indexing, or excessive relational joins can severely hinder performance, especially under stress. Conversely, clear separation of data responsibilities across services and databases (as implemented in this project's microservices architecture) can improve not only scalability but also maintainability and deployment flexibility.

One limitation observed in the experiments is the use of synthetic data and limited infrastructure, which may not capture the full complexity of real-world user behavior. Moreover, while tests were performed with up to 1000 concurrent operations, future evaluations should consider higher concurrency levels (e.g., 5000+ users) to further validate the system's elasticity.

In summary, the performance differences observed between the SQL and NoSQL-backed services confirm the effectiveness of the polyglot persistence model adopted. They also emphasize the need for careful data modeling and distribution strategies in any scalable cloud-native system. These decisions significantly influence not only system responsiveness but also resilience and adaptability in production environments.

X. Conclusion

This project has demonstrated the technical viability and architectural soundness of designing a scalable social networking platform using microservices and a hybrid database strategy. The integration of SQL and NoSQL technologies allowed the system to handle different data types and access patterns effectively, balancing consistency, performance, and flexibility.

From a data management perspective, the relational database (PostgreSQL) ensured strong consistency and transactional integrity for critical information such as user profiles, authentication metadata, and structured relationships. Meanwhile, the use of a document-based NoSQL database (MongoDB) for storing posts, interactions, and follow data enabled horizontal scalability and better performance in scenarios involving high concurrency and large volumes of unstructured or semi-structured data.

The experimental validation through stress testing confirmed that NoSQL-backed services performed significantly better under load, particularly in the test involving concurrent user follow actions, which achieved the lowest response time and highest success rate. This underscores the importance of matching the data model to the workload type—a principle that is central to polyglot persistence strategies. Conversely, the slightly reduced performance observed in services relying on SQL highlights the need for careful schema design, indexing, and query optimization in relational systems when operating under scale.

The microservices architecture, combined with containerization via Docker and orchestration through Kubernetes, provided modularity, fault isolation, and the ability to scale services independently. These characteristics are essential for modern cloud-native applications, particularly those that demand continuous availability and support for dynamic workloads. Furthermore, the implementation of Redis caching layers contributed to reducing response times and alleviating database pressure for frequently accessed resources.

The results obtained reinforce several best practices in systems engineering: the need for architectural decoupling, the importance of workload-specific storage strategies, and the value of synthetic data in simulation environments when real-world data is not available. The iterative process of refining the database schema and service interactions proved critical to achieving the performance objectives and maintaining system flexibility.

Recommendations for Future Work: Future development should aim to test the platform under higher concurrency levels (e.g., 5000 or more users), integrate real-time analytics and content recommendation pipelines, and deploy the solution in a production-like environment to validate its robustness. Additionally, expanding the system to support mobile platforms, implementing security audits, and introducing event-driven messaging (e.g., Kafka) could further improve scalability and resilience. Finally, a deeper exploration into dynamic schema evolution and auto-scaling policies would support continued growth and adaptability of the platform.

References

[1] M. Fowler, *Microservices: a definition of this new architectural term*, 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html [Accessed: May 12, 2025]

[2] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, O'Reilly Media, 2015.

[3] R. Cattell, "Scalable SQL and NoSQL data stores," *SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011.

[4] M. Stonebraker and R. Cattell, "10 rules for scalable performance in 'simple operation' datastores," *Communications of the ACM*, vol. 54, no. 6, pp. 72–80, 2011.

[5] A. Ahmed, M. A. Hossain, and K. Andersson, "Scalable data models for social networks," *IEEE Access*, vol. 6, pp. 58694–58705, 2018.

[6] E. Brewer, "CAP twelve years later: How the 'rules' have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[7] LinkedIn Engineering, *How LinkedIn uses NoSQL databases*, 2020. [Online]. Available: https://engineering.linkedin.com [Accessed: May 12, 2025]

[8] Netflix Tech Blog, *Polyglot Persistence*, 2013. [Online]. Available: https://netflixtechblog.com [Accessed: May 12, 2025]

XI. Appendices

- Architecture Diagram
- API Documentation
- Test Scripts

XII. Glossary

- **SQL:** Structured Query Language
- **NoSQL:** Non-relational database systems
- **API:** Application Programming Interface
- **JWT:** JSON Web Token
- **CI/CD:** Continuous Integration / Continuous Deployment

XIII. List of Figures/Tables

List of Figures

List of Tables