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 networks face increasing challenges due to the scale of user interaction and data generation. This paper presents the design and validation of a scalable social platform using a microservices-based architecture and a hybrid SQL/NoSQL data model. Our results show that combining polyglot persistence with container orchestration achieves high availability, performance, and modularity under concurrent workloads.**

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

I. Introduction

Social networking platforms play a central role in modern digital communication. They allow individuals, communities, and organizations to engage in real-time interaction and content sharing at unprecedented scale. However, supporting such high levels of interaction brings challenges in terms of system scalability, performance, and resilience.

Traditional monolithic systems tend to fail under high load, primarily due to their tightly coupled architecture and limited horizontal scalability. To overcome these limitations, the software industry has widely adopted microservices as an architectural pattern. Microservices decompose a system into small, independently deployable services, improving scalability, fault isolation, and continuous delivery.

In parallel, the need to manage heterogeneous data types—ranging from structured user profiles to semi-structured posts and comments—has led to the rise of polyglot persistence. This approach combines relational databases (SQL) for structured data and non-relational databases (NoSQL) for flexible and high-volume content storage.

Related research supports this approach. Fowler and Newman emphasize microservices for modular, evolvable systems. Cattell, Stonebraker, and Ahmed et al. discuss the trade-offs between consistency and scalability in SQL and NoSQL stores. Large-scale systems like those at LinkedIn and Netflix have validated the polyglot persistence model in real-world scenarios.

Building on these foundations, this paper presents a social media platform prototype inspired by Twitter (X), which integrates microservices, PostgreSQL, MongoDB, Redis caching, and Kubernetes orchestration. We present its design rationale, implementation strategy, and performance evaluation.

II. Methods and Materials

The development of the platform began with a formal problem definition from a systems engineering perspective. The central challenge was to design a backend infrastructure capable of supporting high user concurrency, dynamic content generation, and consistent performance across services. These requirements implied the need for scalability, fault isolation, and responsiveness in both computation and data storage layers.

A. System Decomposition and Service Design

An initial functional decomposition of the platform identified key domains: user authentication, account management, content publication (posts), social graph relationships (followers), and personalized feed generation. These domains were mapped to independent microservices, each with its own responsibilities, data models, and APIs. The use of microservices allowed modular development and independent scaling of components based on workload.

To support asynchronous communication and ensure loose coupling between services, RESTful APIs were adopted as the interface standard. Each service was designed to be stateless, enabling horizontal scaling and failover without reliance on shared session memory.

B. Hybrid Data Strategy and Storage Assignment

A core design decision involved choosing the appropriate database technology for each service's data model. Following the principles of polyglot persistence, we adopted a hybrid approach combining SQL and NoSQL systems. This decision was guided by three critical criteria:

1) **Data Structure**: Well-defined, structured entities such as user credentials and metadata benefit from relational models, while dynamic and semi-structured content is better suited for document-based storage.
2) **Access Patterns**: High-frequency write and read operations with flexible schema needs—such as posts and reactions—are best handled by NoSQL systems.
3) **Consistency Requirements**: Critical operations, such as authentication and identity verification, demand ACID compliance and transactional guarantees, justifying SQL storage.

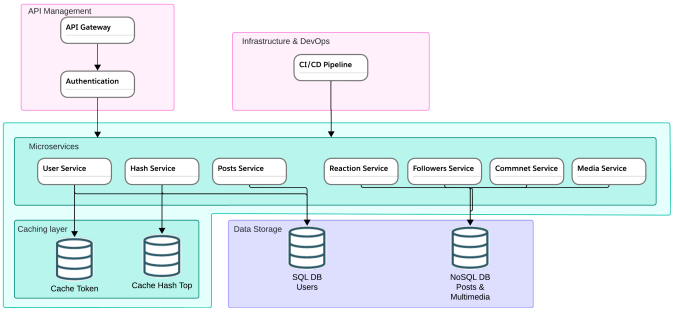

Fig. 1. System Architecture Diagram

Based on this classification:

- PostgreSQL was used to store structured data such as `Users`, `Sessions`, and `Credentials`, ensuring data integrity and relational consistency.
- MongoDB was selected for collections such as `Posts`, `Followers`, and `Reactions`, which exhibit flexible schemas and require high write throughput and scalability.

This separation was essential to avoid mismatches between the workload and storage engine characteristics. Misplacing a semi-structured, high-volume dataset in a rigid SQL schema could introduce latency and reduce scalability. Conversely, storing critical transactional data in a NoSQL store could compromise consistency and security.

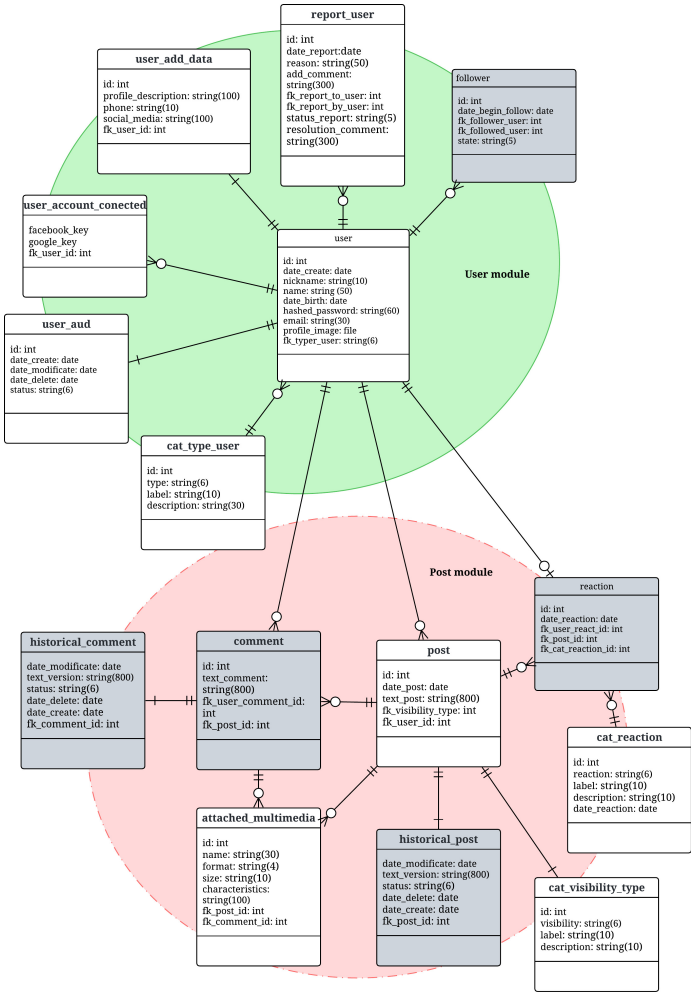

Fig. 2. Entity-Relationship Diagram with SQL and NoSQL Separation

C. Infrastructure and Orchestration

All services were containerized using Docker to ensure reproducibility and isolation. Kubernetes was employed for orchestration, enabling automatic scaling, service discovery, and fault tolerance. An API Gateway acted as the single entry point for external clients, managing routing, authentication, and rate limiting.

Redis was introduced as an in-memory cache layer to support fast retrieval of frequently accessed content such as trending posts and session tokens. This reduced latency and minimized pressure on the underlying databases.

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.

D. Data Generation and System Evaluation

In the absence of access to real-world datasets, the evaluation phase required the generation of synthetic data that could accurately reflect the expected behaviors and interaction patterns of users on a social media platform. To accomplish this, generative AI tools, such as ChatGPT, were employed to produce a variety of realistic and contextually relevant records. These included user profiles, posts, comments, reactions, and social connections (follows).

This approach allowed for the simulation of multiple scenarios, such as mass user registration, bulk content publication, and concurrent user interactions—elements typical of modern social platforms. Although the data was artificial, it maintained coherence in structure and semantics, enabling a reliable assessment of how the system would behave in production-like conditions.

The synthetic nature of the datasets proved valuable not only for validating functional correctness but also for identifying potential bottlenecks and evaluating the system's capacity to process, store, and retrieve large volumes of diverse data. In particular, the generated data covered both structured ele-

ments (ideal for SQL storage) and semi-structured or dynamic components (well-suited for NoSQL databases), providing a comprehensive test across both storage paradigms.

Two primary tools were used in the testing environment:

- **Postman**: utilized for validating API endpoints through functional and automated test suites. It ensured correct response formatting, status codes, and adherence to service contracts across the microservices architecture.
- **Apache JMeter**: employed for load and stress testing. JMeter enabled the simulation of hundreds to thousands of concurrent virtual users, allowing the team to observe system behavior under pressure and measure key metrics such as response time, throughput, and error rate.

This testing strategy enabled the evaluation of service performance, orchestration efficiency, and the impact of architectural decisions such as caching, container orchestration, and database partitioning. The data generation process, though artificial, offered a meaningful approximation of real-world behavior and was instrumental in validating the robustness, scalability, and responsiveness of the proposed system architecture.

Figures 1 and 2 illustrate the architectural overview and the division of data entities across SQL and NoSQL stores, which formed the basis for these evaluations.

III. 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.

A. Interpretation and Data Implications

The observed results provide practical insights into how backend design and data storage strategies impact system performance under concurrent load. Specifically, the superior performance of the *Concurrent Following Users* test, with its minimal latency and 100% success rate, reinforces the benefits of using NoSQL databases for operations involving high-frequency, loosely structured, and rapidly growing datasets.

In contrast, the relatively lower success rates and longer response times recorded for the user registration and post insertion tests highlight the limitations of relational databases when handling intensive concurrent writes and validations. These operations typically involve strict schema enforcement, foreign key constraints, and transactional integrity, all of which introduce processing overhead during high load scenarios.

These findings underscore the importance of thoughtful data modeling and storage assignment in distributed systems. Properly categorizing entities based on their structure and access behavior is essential—not only for meeting performance goals but also for maintaining long-term scalability. For instance, relational models remain ideal for critical, consistent, and structured data such as user credentials, while document-based storage better supports variable, high-volume interactions like post creation and social connections.

From a systems engineering perspective, these test outcomes support future architectural adjustments. Services experiencing bottlenecks may require data denormalization, indexing enhancements, or even migration to alternative storage models (e.g., transitioning from SQL to NoSQL or vice versa) depending on the specific workload. Additionally, caching strategies may be tuned to further optimize frequent query paths.

In summary, the performance metrics obtained from the load tests validate the core design decisions and offer actionable feedback to refine the distribution of data across storage engines. These results can guide future iterations of the platform as it evolves toward real-world deployment.

IV. Conclusion

This research validated the feasibility of designing a scalable social networking platform through a microservices architecture combined with a hybrid SQL/NoSQL data strategy. The system effectively leveraged relational databases for transactional consistency and NoSQL for handling high-throughput, schema-flexible data—demonstrating the strengths of polyglot persistence.

Performance evaluations under concurrent load confirmed the benefits of aligning data models with workload characteristics. Notably, NoSQL-backed services showed superior scalability, while SQL-based components highlighted the need for schema optimization under stress.

The modular architecture, containerization, and orchestration infrastructure contributed to system resilience and independent service scaling. These findings reinforce best practices in distributed system design and support the continued application of hybrid models in dynamic environments.

Future Work: Further research should explore large-scale deployments, real-time analytics integration, and the use of event-driven communication to enhance elasticity and observability.

A. Consideration of Distributed and Parallel Database Models

As the platform evolves toward higher concurrency and global scalability, incorporating distributed or parallel database models becomes a critical area of future improvement. While the current hybrid approach effectively separates workloads between SQL and NoSQL technologies, it still relies on single-instance deployments for each database engine.

Distributed databases, such as Google Spanner, CockroachDB, or Cassandra, offer horizontal partitioning (sharding), replication, and built-in fault tolerance. These characteristics allow the system to handle higher write throughput, maintain availability during node failures, and support geographic data distribution with lower latency for global users.

Parallel databases, on the other hand, allow for concurrent query execution across multiple processors or nodes, significantly reducing response time for read-intensive or analytical operations—such as feed generation, trend analysis, or content recommendation.

Integrating such models into the current architecture could enhance not only performance but also resilience and availability. For example, distributing the NoSQL collections across nodes using a partitioned keyspace could improve the handling of sudden spikes in follow or post activity. Similarly, deploying PostgreSQL with replication or partitioning mechanisms could prevent bottlenecks during account creation and authentication under peak loads.

In summary, evaluating the adoption of distributed and parallel database models represents a promising direction for extending the platform's scalability and reliability. These technologies align with the modular and service-oriented nature of the system and would support long-term growth, especially under production-level workloads an

REFERENCES

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

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

[3] R. Cattell, "Scalable SQL and NoSQL data stores," *ACM 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

[8] Netflix Tech Blog, "Polyglot Persistence," 2013. [Online]. Available: https://netflixtechblog.com

[9] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[10] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade," *ACM Queue*, vol. 14, no. 1, pp. 70–93, 2016.

[11] H. Shadid and A. Rabai, "Performance Evaluation of SQL and NoSQL Databases: A Comparative Study," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 11, pp. 45–52, 2019.

[12] D. Batista et al., "A Survey of Microservice Architecture," *IEEE Latin America Transactions*, vol. 19, no. 4, pp. 688–700, 2021.

[13] M. Armbrust et al., "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[14] J. Bonér, "Reactive Microservices Architecture: Design Principles for Distributed Systems," Lightbend, 2016. [White Paper]

[15] Redis, "Redis Documentation," [Online]. Available: https://redis.io/docs/

[16] PostgreSQL Global Development Group, "PostgreSQL Documentation," [Online]. Available: https://www.postgresql.org/docs/

[17] MongoDB Inc., "MongoDB Architecture Guide," [Online]. Available: https://www.mongodb.com/docs/manual/core/architecture/

V. APPENDICES

- Architecture Diagram
- API Documentation
- Test Scripts

VI. GLOSSARY

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

VII. LIST OF FIGURES/TABLES

LIST OF FIGURES

LIST OF TABLES