Assignment Report on:

"Architectural Design for Facebook"

Submitted to:

Mridha Md. Nafis Fuad
Lecturer
Institute of Information Technology
University of Dhaka

Submitted by:

Pal Swadhin **(**BSSE **1302**)
Mostafizur Rahman (BSSE **1320**)
Rony Majumder (BSSE **1325**)
Ibne Bin Rafid **(**BSSE **1330**)
Fareya Azam (BSSE **1331**)

Course- **Software Design (SE-606)**
Institute of Information Technology

Submitted on 16th January, 2025

Table of Contents

Architectural Design for Facebook: microservices and distributed designs

1. High-Level Modules

- **User Management**: Handles registration, authentication, and profile management.
- **Content Management**: Deals with posts, photos, videos, comments, and likes.
- **Social Graph Management**: Manages user connections (friends, followers).
- **Feed Management**: Personalized news feed generation.
- **Notification System**: Real-time alerts for likes, comments, friend requests, etc.
- **Messaging**: Instant messaging and group chat.
- **Search**: Global search for users, pages, groups, and posts.
- **Ads System**: Ad creation, targeting, and analytics.
- **Analytics**: Insights for user behavior and content performance.
- **Admin Tools**: Moderation and compliance enforcement.

2. Key Microservices

Each module above can be broken into smaller microservices:

2.1 User Management

- **Authentication Service**: OAuth, session management, password recovery.
- **User Profile Service**: CRUD operations for user details.
- **Privacy Settings Service**: Controls user permissions and privacy settings.

2.2 Content Management

- **Post Service**: Create, read, update, and delete (CRUD) posts.
- **Media Service**: Handles uploads and storage of images and videos.
- **Comment Service**: Manages comments on posts.
- **Reaction Service**: Like, love, or any reaction type.

2.3 Social Graph Management

- **Friendship Service**: Manage friend requests and relationships.
- **Follow Service**: Handles follower-following relationships.

2.4 Feed Management

- **Feed Aggregation Service**: Collects posts for feeds.
- **Recommendation Engine**: Suggests content based on user behavior.
- **Ranking Service**: Prioritizes posts in feeds.

2.5 Notification System

- **Push Notification Service**: Sends notifications to devices.
- **Inbox Notification Service**: Stores notifications in user accounts.

2.6 Messaging

- **Real-time Chat Service**: WebSocket-based instant messaging.
- **Group Messaging Service**: Manages multi-user chat.

2.7 Search

- **Search Indexing Service**: Builds indexes for fast searching.
- **Search Query Service**: Handles user queries.

2.8 Ads System

- **Ad Targeting Service**: Matches ads to user demographics.
- **Billing Service**: Manages advertiser payments.

2.9 Analytics

- **Event Logging Service**: Tracks user interactions.
- **Behavior Analytics Service**: Analyzes user data.

2.10 Admin Tools

- **Moderation Service**: Flags inappropriate content.
- **Reporting Service**: Manages user-reported issues.

3. Distributed Systems Design

3.1 Databases

- **User Database**: Stores user profiles (e.g., PostgreSQL, MySQL).
- **Content Database**: Stores posts, comments, and media metadata.
- **Social Graph Database**: Graph database (e.g., Neo4j) for relationships.
- **Feed Cache**: Redis or Memcached for feed generation.
- **Search Index**: Elasticsearch for full-text search.

3.2 Storage

- **Media Storage**: S3-compatible storage (e.g., MinIO) for photos and videos.
- **Backup System**: Ensures data reliability.

3.3 Message Queue

- **Kafka**: For asynchronous tasks like feed updates and notifications.

3.4 Real-Time Systems

- **WebSockets**: For live notifications and messaging.
- **Pub/Sub**: Real-time feed updates (e.g., Google Pub/Sub).

3.5 Load Balancing

- **Reverse Proxy**: Nginx or HAProxy for distributing requests.
- **Global Load Balancer**: Ensures availability across regions.

3.6 Caching

- **Edge Caching**: CDNs for faster content delivery.
- **Application Caching**: Redis for reducing database hits.

4. Component Interaction

4.1 Frontend

- React, Vue, or Angular for UI.
- GraphQL or REST APIs to interact with the backend.

4.2 Backend

- Node.js, Python (Flask/Django), or Java (Spring Boot) for services.
- APIs exposed for microservices communication.

4.3 Microservices Communication

- **Internal APIs**: REST or gRPC.
- **Event-driven Communication**: Using Kafka or RabbitMQ.

5. Deployment

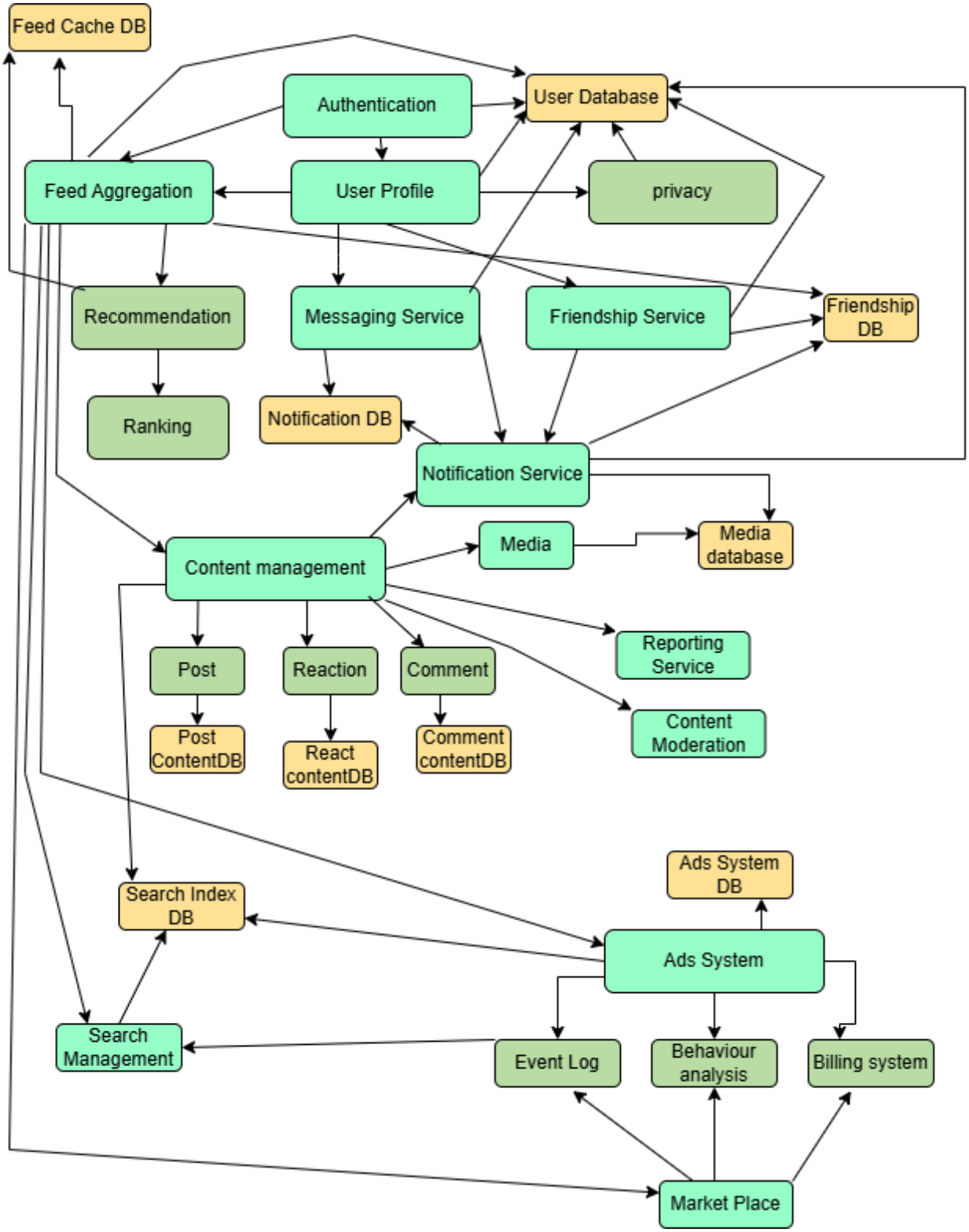
5.1 Containerization

- Docker containers for microservices.
- Kubernetes for orchestration.

5.2 Monitoring and Observability

- Prometheus and Grafana for metrics.
- ELK Stack for logging.

Connected Micro Services



Several databases are shared across multiple components in the system architecture, requiring efficient replication strategies to ensure data consistency and availability. For instance:

- The **User Database**, used by Authentication, User Profile, Privacy, and Friendship Service, leverages **Event-Driven Data Synchronization** to propagate user-related updates in near real-time while maintaining loose coupling between services.
- Content-related databases, such as the **Post Content DB**, **React Content DB**, and **Comment Content DB**, utilize **Change Data Capture (CDC)** to stream updates to dependent services like Feed Aggregation, ensuring the timely synchronization of posts, reactions, and comments.
- The **Media Database**, a shared resource, employs **API-Based Data Sharing** to avoid redundant storage and provide centralized access for services like Media and Reporting.
- To support low-latency operations, the **Feed Cache DB** and **Friendship DB** also use **Event-Driven Data Synchronization**, keeping feeds and notifications aligned with user interactions.
- For high-demand operations like those involving the **Ads System DB**, **Distributed Databases** ensure fault tolerance, scalability, and consistency across Ads-related components.

These replication strategies deliver a robust, scalable, and high-performance system that maintains consistency across its distributed microservices architecture.

Database and Components Table

#	Database	Connected Components	Replication Strategy	Reason
1	User Database	Authentication, User Profile, Privacy, Friendship Service	Event-Driven Data Synchronization	Decouples services and ensures eventual consistency for user-related data changes.
2	Feed Cache DB	Feed Aggregation, Recommendation, Ranking	Event-Driven Data Synchronization	Provides low-latency feed updates using cached data.
3	Notification DB	Notification Service, Messaging Service	Event-Driven Data Synchronization	Enables real-time notifications and message updates.
4	Media Database	Media Service, Reporting Service	API-Based Data Sharing	Avoids duplication of large files while maintaining centralized access for connected services.
5	Post Content DB	Content Management (Post Service)	Change Data Capture (CDC)	Streams updates to dependent services for real-time synchronization of posts.
6	React Content DB	Content Management (Reaction Service)	Change Data Capture (CDC)	Ensures reaction updates are propagated to related components like Feed and Notifications.
7	Comment Content DB	Content Management (Comment Service)	Change Data Capture (CDC)	Keeps comment-related data synchronized with Feed Cache and Search Index DB.
8	Search Index DB	Search Management, Content Management	Change Data Capture (CDC)	Maintains accurate and up-to-date search results for users and content.
9	Friendship DB	Friendship Service, Feed Aggregation, Notification Service	Event-Driven Data Synchronization	Ensures updates to friendships propagate to feeds and notifications for a seamless user experience.
10	Ads System DB	Ads System, Behavior Analysis, Event Log, Marketplace	Distributed Databases	Handles high data volumes and ensures consistency across Ads-related services for scalability.

Figure: The figure of how the databases, components, replication strategies, and reasons align with different microservices.

Replication and Synchronization Strategies

1. User Database

- **Connected Components**: Authentication, User Profile, Privacy, Friendship Service
- **Replication Strategy**: **Event-Driven Data Synchronization**
    - Changes to user data (e.g., profile updates) are published as events.
    - Services like Friendship or Feed Aggregation subscribe to user-related events for synchronization.

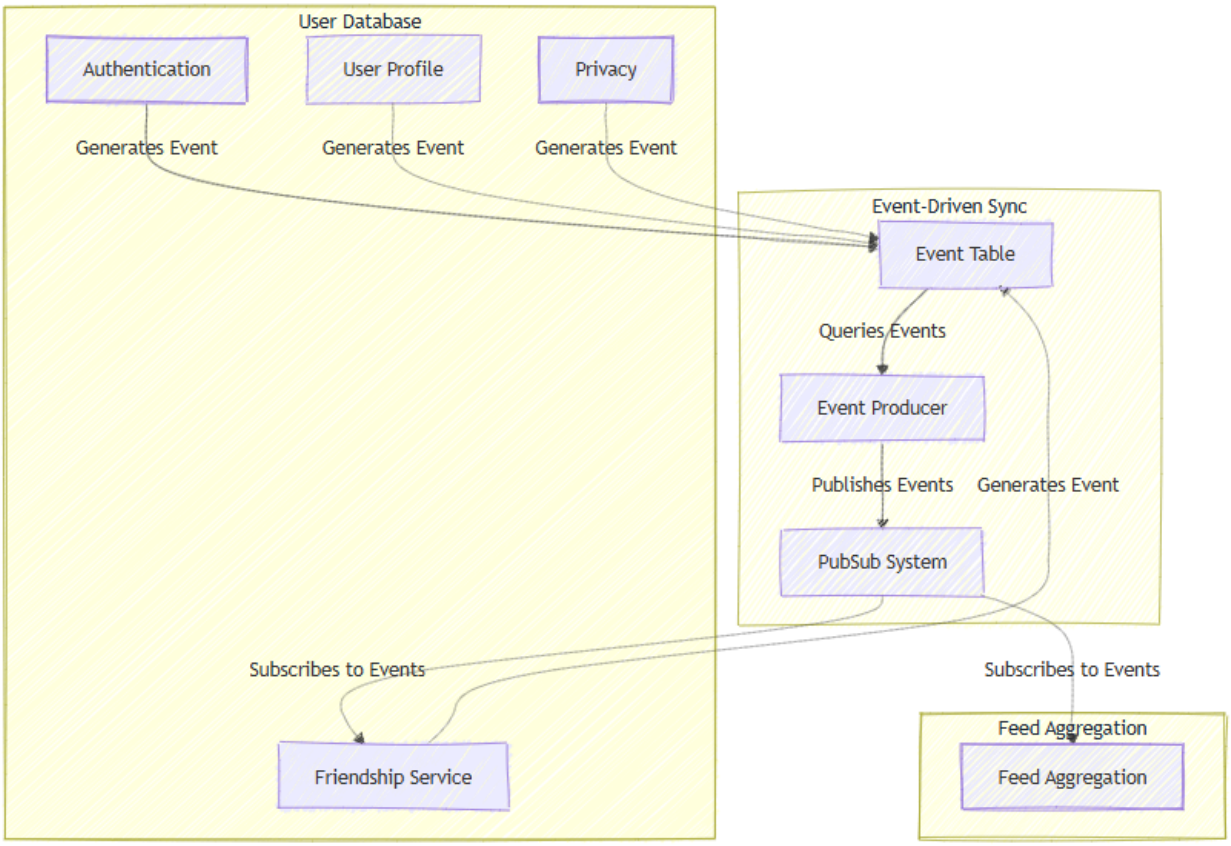


Figure: User Database Replication and Synchronization using event Event-Driven Data Synchronization

9. Friendship DB

- **Connected Components**: Friendship Service, Feed Aggregation, Notification Service
- **Replication Strategy**: **Event-Driven Data Synchronization**
    - Friendship updates (e.g., new connections) are published to Feed Aggregation and Notification systems.

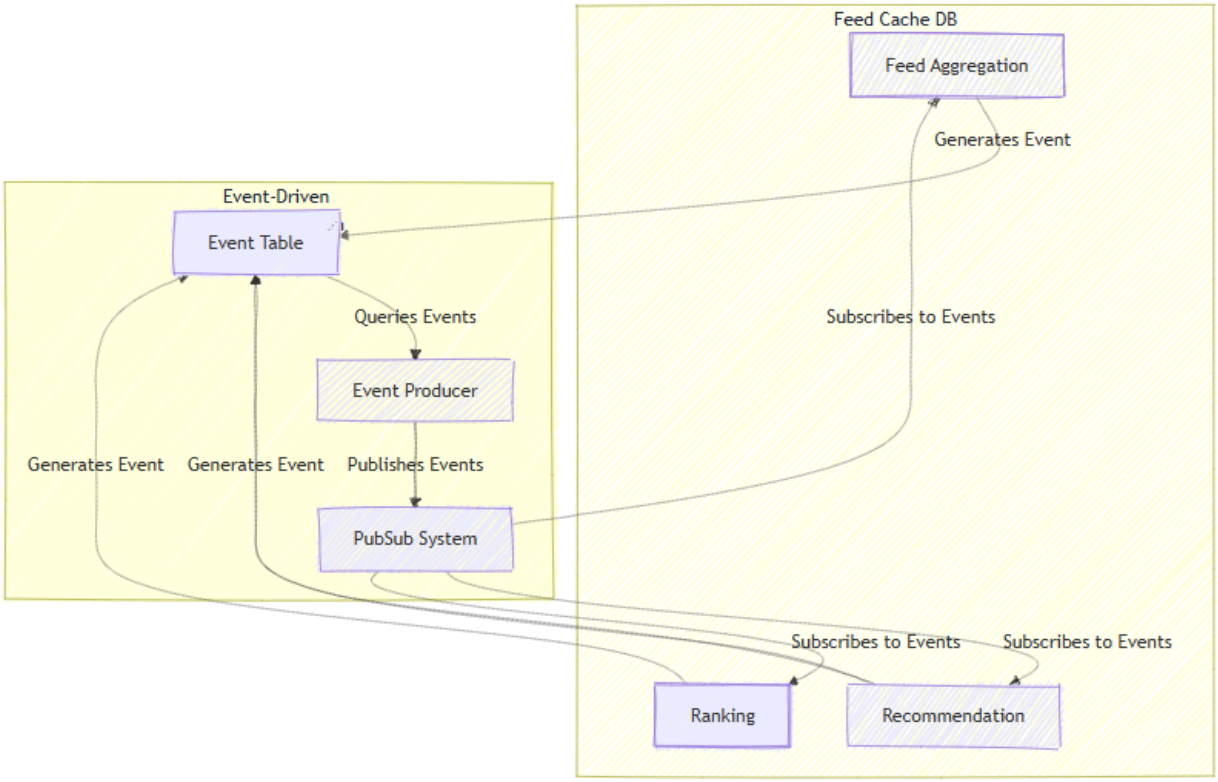


Figure: Friendship DB Replication and Synchronization using event Event-Driven Data Synchronization

2 & 3 . Notification DB & Feed Cache DB

Notification DB

- **Connected Components**: Notification Service, Messaging Service
- **Replication Strategy**: **Event-Driven Data Synchronization**
    - Notifications (e.g., new messages, friend requests) are generated via events.
    - Services like Messaging update their local data stores asynchronously.

Feed Cache DB

- **Connected Components**: Feed Aggregation, Recommendation, Ranking
- **Replication Strategy**: **Event-Driven Data Synchronization**
    - Feed changes due to new posts, reactions, or comments are processed asynchronously.
    - The cache is updated in near real-time for low-latency feed delivery.

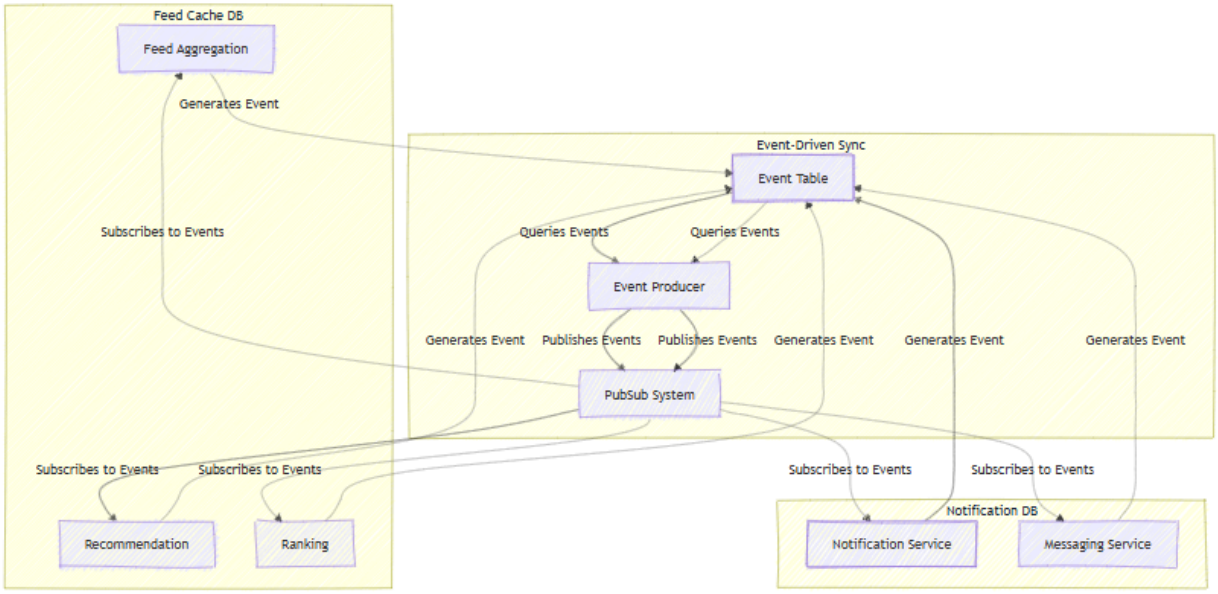


Figure: Notification DB & Friendship DB Replication and Synchronization using event Event-Driven Data Synchronization

4. Media Database

- **Connected Components**: Media Service, Reporting Service
- **Replication Strategy**: **API-Based Data Sharing**
    - Media is stored centrally, and metadata is shared via APIs.
    - This avoids large file duplication while maintaining access to connected services.

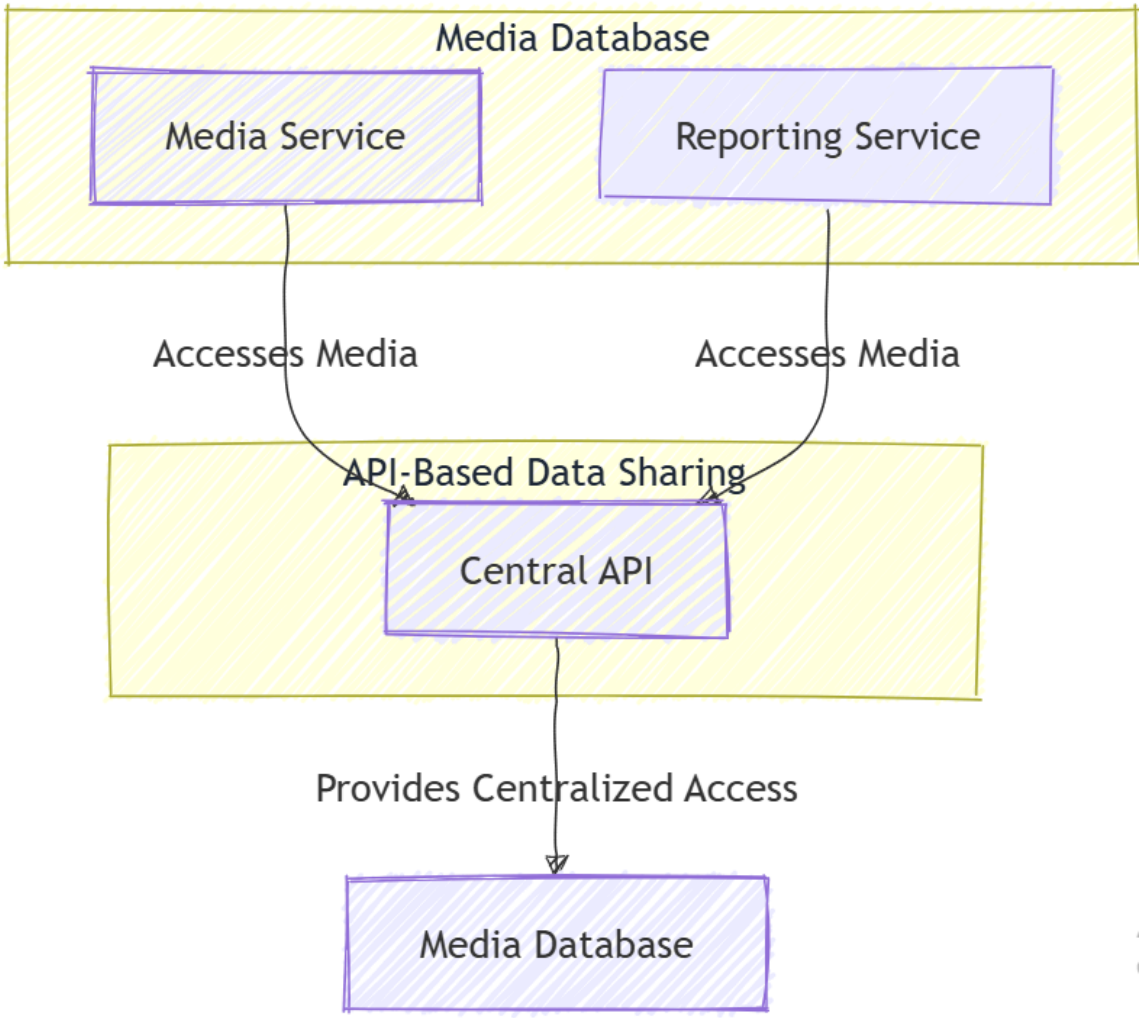


Figure: Media Database Replication and Synchronization using API-Based Data Sharing

6. React Content DB

- **Connected Components**: Content Management (Reaction Service)
- **Replication Strategy**: **Change Data Capture (CDC)**
    - Reaction updates are captured and synchronized with Feed Cache DB and Notification DB.

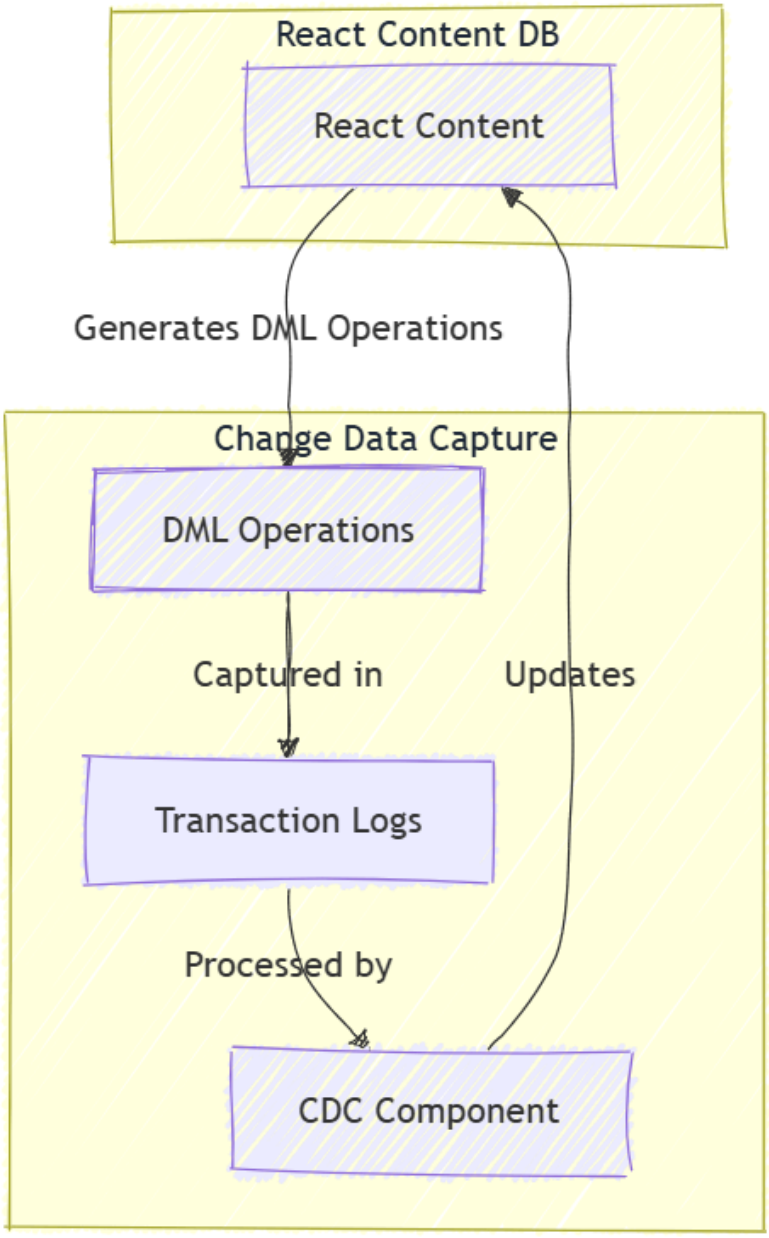


Figure: React Content DB Replication and Synchronization using Change Data Capture (CDC)

5. Post Content DB

- **Connected Components**: Content Management (Post Service)
- **Replication Strategy**: **Change Data Capture (CDC)**
    - Changes to posts are streamed to Search Index DB and Feed Cache DB.
    - Tools like Debezium ensure updates are propagated to dependent systems.

7. Comment Content DB

- **Connected Components**: Content Management (Comment Service)
- **Replication Strategy**: **Change Data Capture (CDC)**
    - Similar to the React Content DB, comment changes propagate to dependent services.

8. Search Index DB

- **Connected Components**: Search Management, Content Management
- **Replication Strategy**: **Change Data Capture (CDC)**
    - Full-text search indexes are updated based on content changes (posts, comments, reactions).

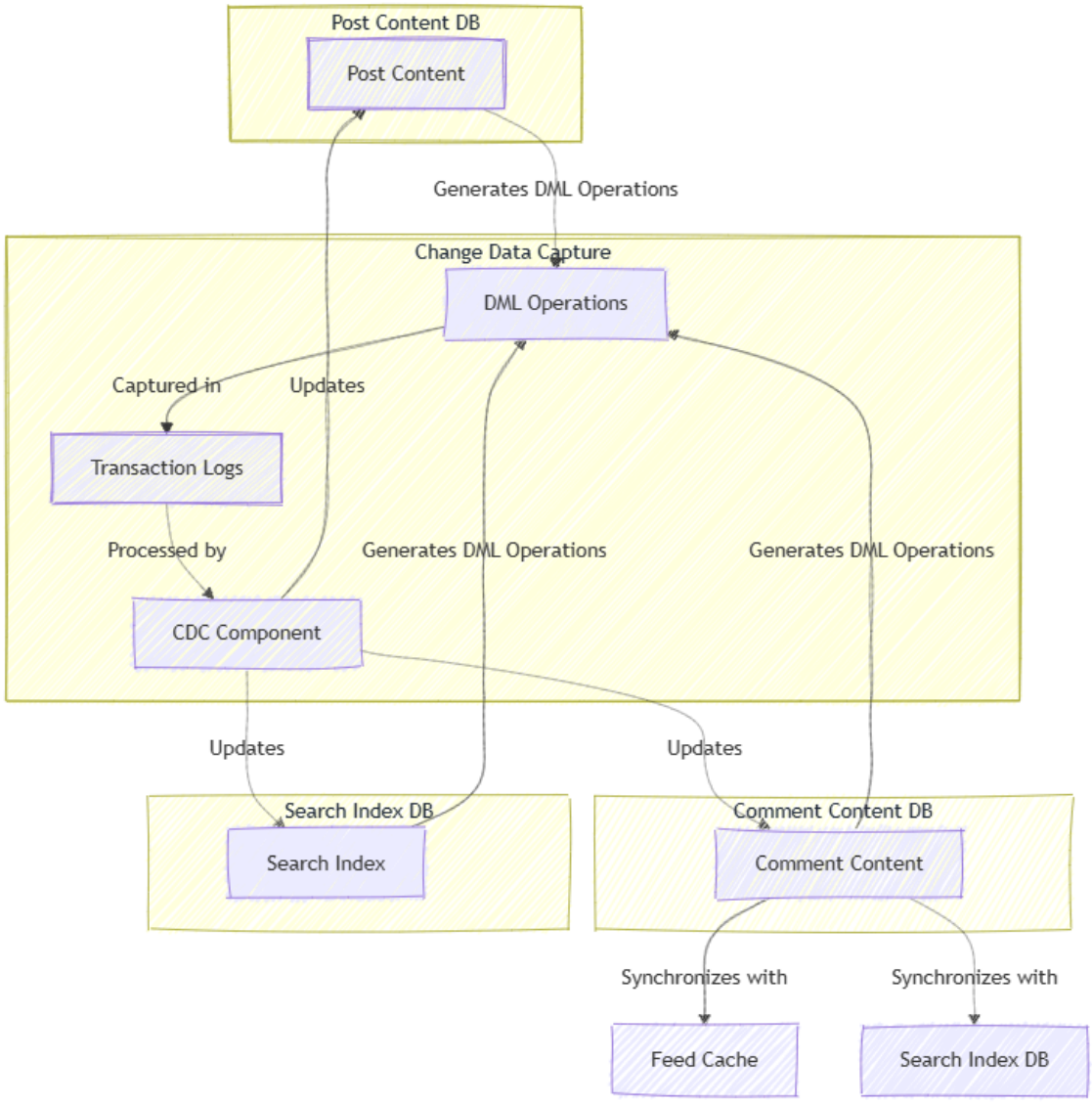


Figure: ***Post Content DB***, ***Comment Content DB*** *&* ***Search Index DB*** *Replication and Synchronization using Change Data Capture (CDC)*

10. Ads System DB

- **Connected Components**: Ads System, Behavior Analysis, Event Log, Marketplace
- **Replication Strategy**: **Distributed Databases**
    - A distributed database like Cassandra ensures scalability and low-latency access across multiple services.

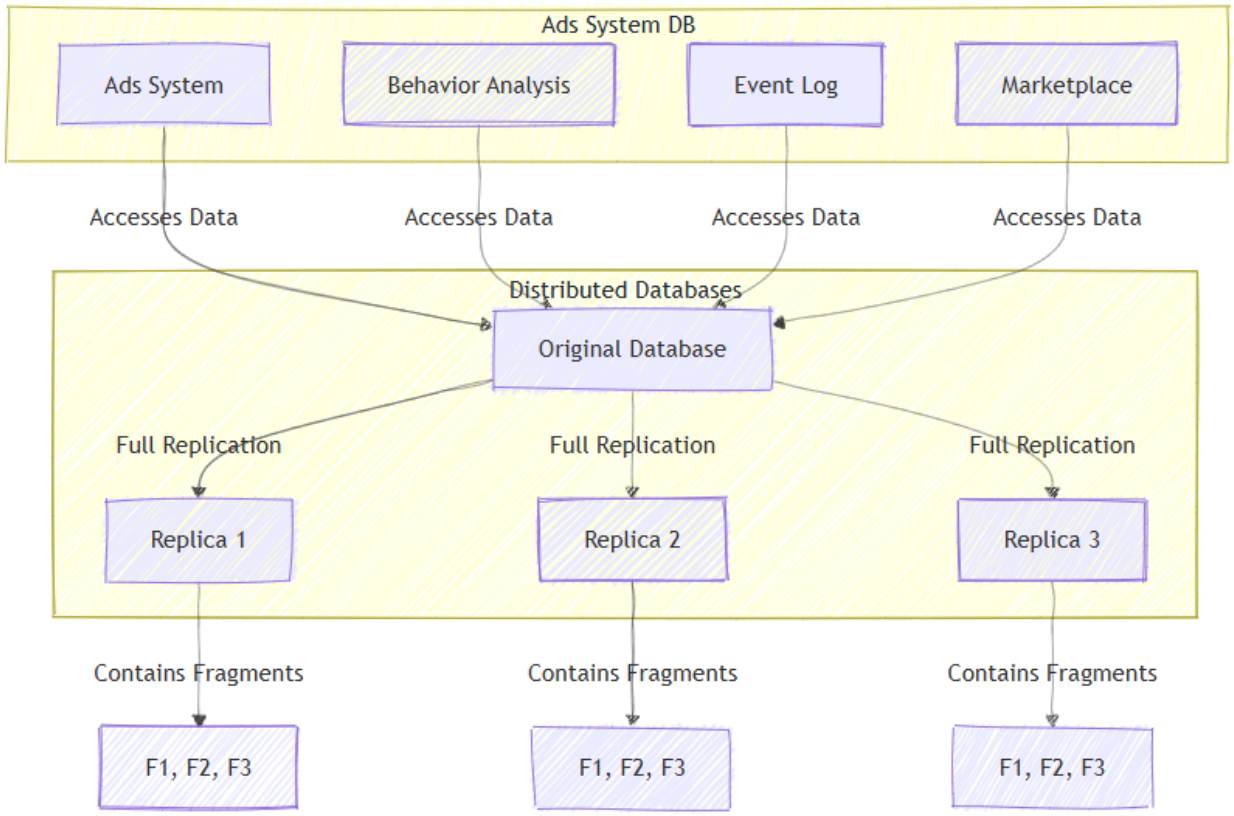


Figure: Ads System DB Replication and Synchronization using Distributed Databases

Global Load Balancer

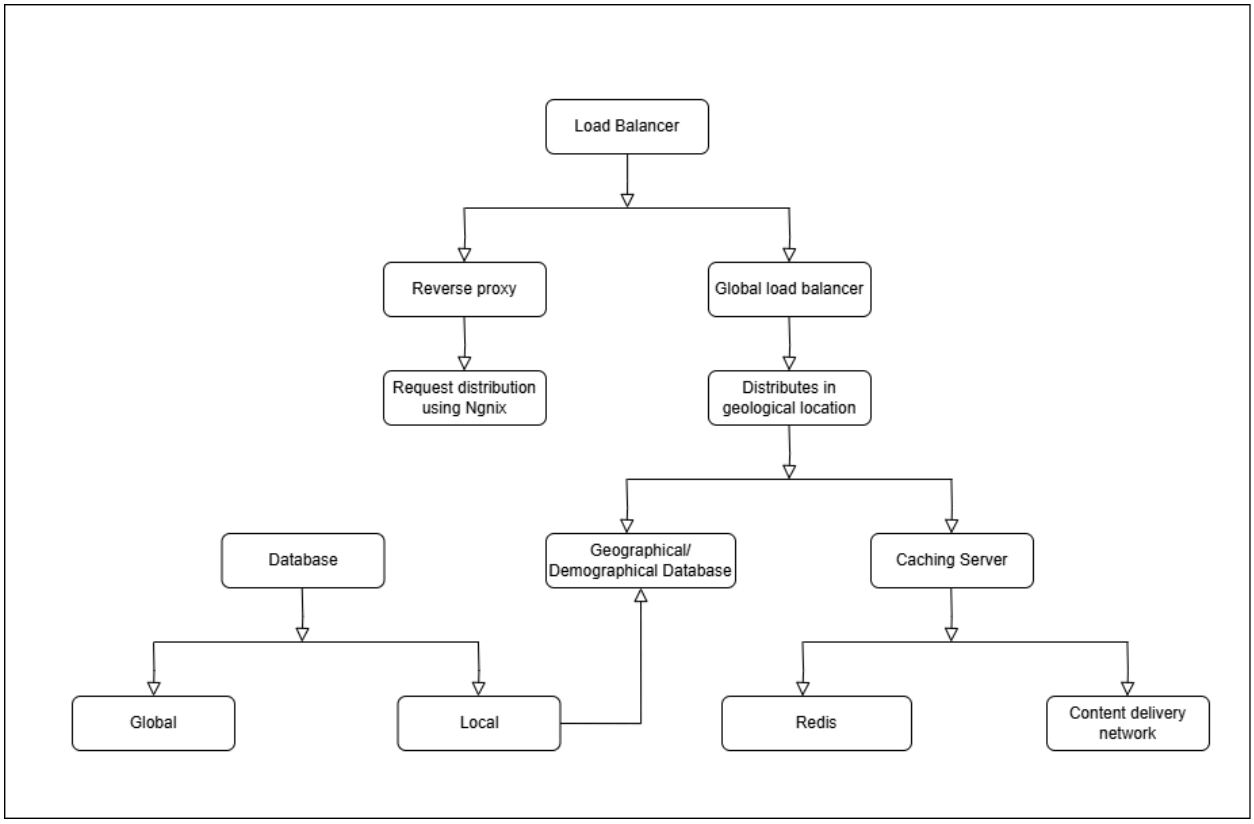


Fig: Load Balancer and Traffic Management System

This diagram illustrates a multi-layered architecture for handling web traffic efficiently:

1. **Load Balancer**: The primary component that manages incoming traffic.
2. **Reverse Proxy**: Uses Nginx to distribute requests to local databases and services.
3. **Global Load Balancer**: Distributes traffic based on geographical location.
4. **Geographical/Demographic Database**: Differentiates between global and local databases for efficient routing.
5. **Caching Server**: Uses Redis and Content Delivery Network (CDN) for faster data retrieval and reduced latency.

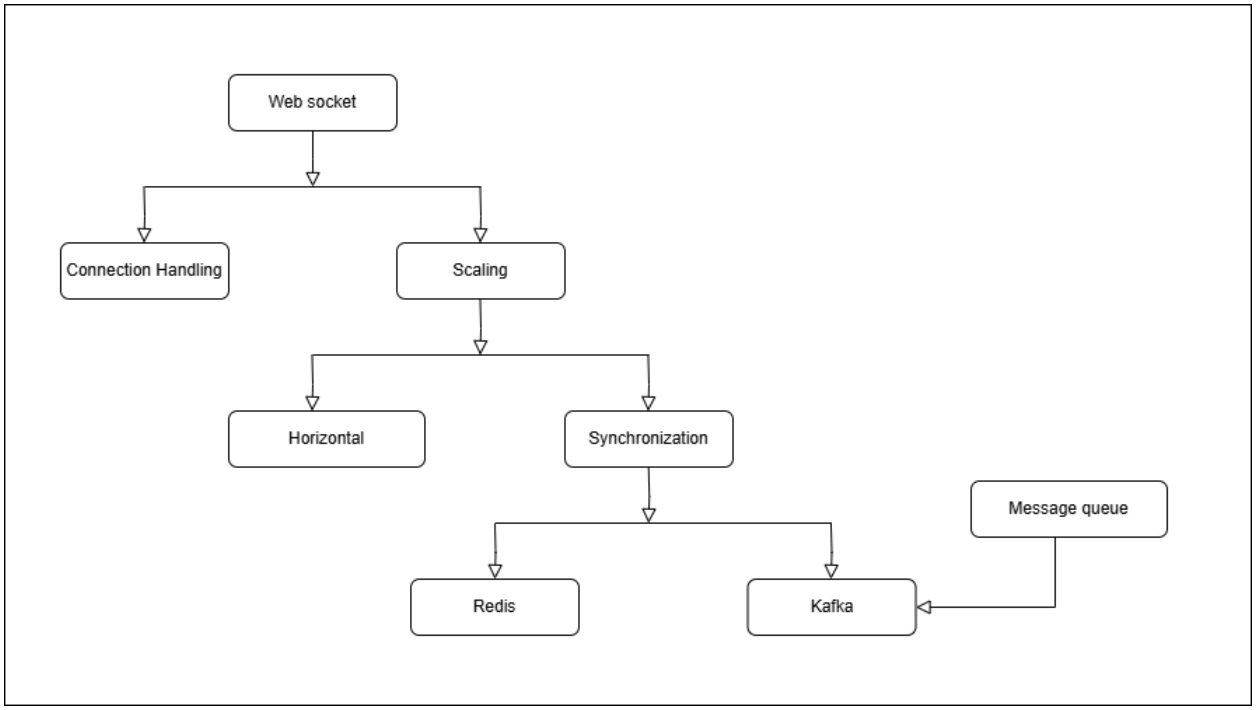
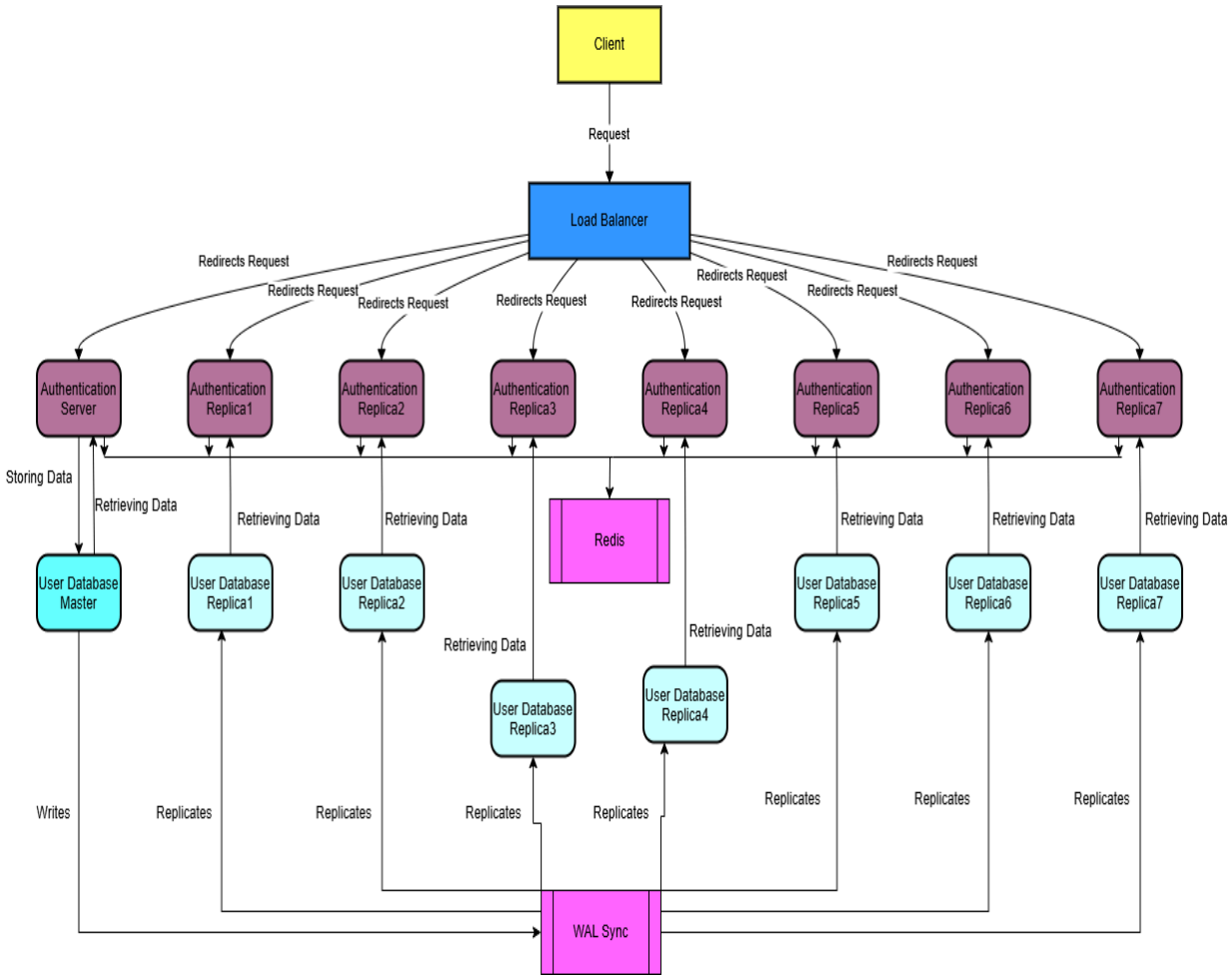


Fig: WebSocket Connection, Scaling, and Synchronization Workflow

This diagram illustrates a WebSocket-based architecture for handling real-time communication and scaling efficiently:

1. **WebSocket**: The core component managing real-time communication.
2. **Connection Handling**: Manages WebSocket connections from clients.
3. **Scaling**: Ensures the system can handle increasing demand:
    - **Horizontal Scaling**: Adds more servers or instances to distribute the load.
    - **Synchronization**: Maintains consistent data across servers.
4. **Redis**: Used for caching or session storage during synchronization.
5. **Kafka**: Used for message streaming and communication across distributed systems.
6. **Message Queue**: Handles asynchronous message processing to ensure reliability.



https://drive.google.com/file/d/1k3eOdMCjpWq2ueZpsO8HO5Hxz1TWOs2c/view?usp=sharing

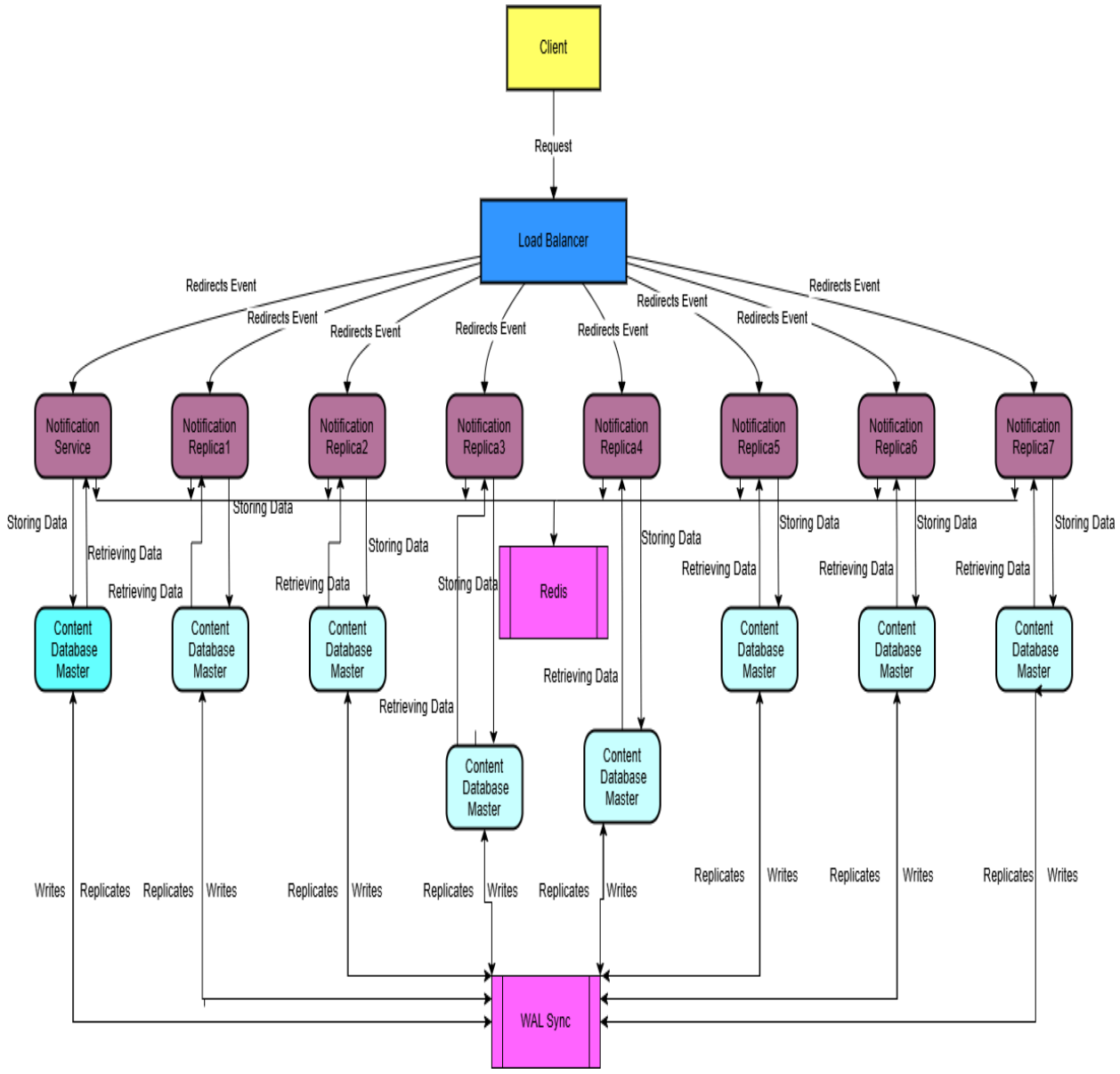
Fig: Distributed Authentication and Database Replication System

This figure represents a distributed architecture for handling authentication requests and database synchronization with high scalability and reliability:

1. **Client**: Sends requests to the system.
2. **Load Balancer**: Distributes incoming requests across multiple authentication servers or replicas.
3. **Authentication Servers**:
    - A central authentication server manages requests.
    - Multiple replicas ensure scalability and fault tolerance, retrieving data from the user database replicas.
4. **User Database**:
    - **Master Database**: Handles write operations and updates.

- **Replicas**: Multiple database replicas are synchronized with the master using replication to handle read-heavy operations and reduce load on the master.

5. **Redis**: Used as a caching layer to improve data retrieval speed for authentication publish/subscribe (Pub/Sub)
6. **WAL Sync**: Ensures Write-Ahead Logging (WAL) to maintain consistency and reliability across the database replicas.



https://drive.google.com/file/d/1DtCG5GhppIcJM4azO-Md5Qkdri9ViS1A/view?usp=sharing

Fig: Distributed Notification and Data Synchronization Architecture

The diagram illustrates a client-server architecture for a notification system, utilizing load balancing, distributed notification replicas, and data synchronization across multiple content database masters. It incorporates:

1. **Client**: Sends requests to the system.
2. **Load Balancer**: Distributes client requests evenly across multiple notification replicas.
3. **Notification Service and Replicas**: Handles event processing and data storage, ensuring redundancy and scalability.
4. **Redis**: Acts as an intermediary for retrieving data in certain scenarios, improving speed.
5. **Content Database Masters**: Store and manage the actual data, with replication mechanisms for fault tolerance.
6. **WAL (Write-Ahead Logging) Sync**: Ensures data consistency and durability across the distributed databases.

Facebook Microservices Feature Collaboration

Communication Technologies Used

1. Synchronous Communication

- **REST APIs**: For direct request-response interactions between services.
- **gRPC**: For high-performance communication, especially for internal service-to-service interactions.
- **GraphQL**: Enables flexible data querying across multiple services, ensuring minimal over-fetching or under-fetching of data.

2. Asynchronous Communication

- **Apache Kafka**: Facilitates event-driven architecture through event streaming and message queuing for decoupled interactions.
- **RabbitMQ**: Provides robust publish/subscribe messaging, particularly for reliable delivery.
- **Redis Pub/Sub**: Powers real-time notifications with low latency.

3. Data Synchronization

- **Change Data Capture (CDC)**: Ensures real-time data propagation from one service's database to others.
- **Event Sourcing**: Captures and stores state changes as events to rebuild service states when needed.
- **Write-Ahead Logging (WAL)**: Tracks database changes to maintain consistency across replicated databases.

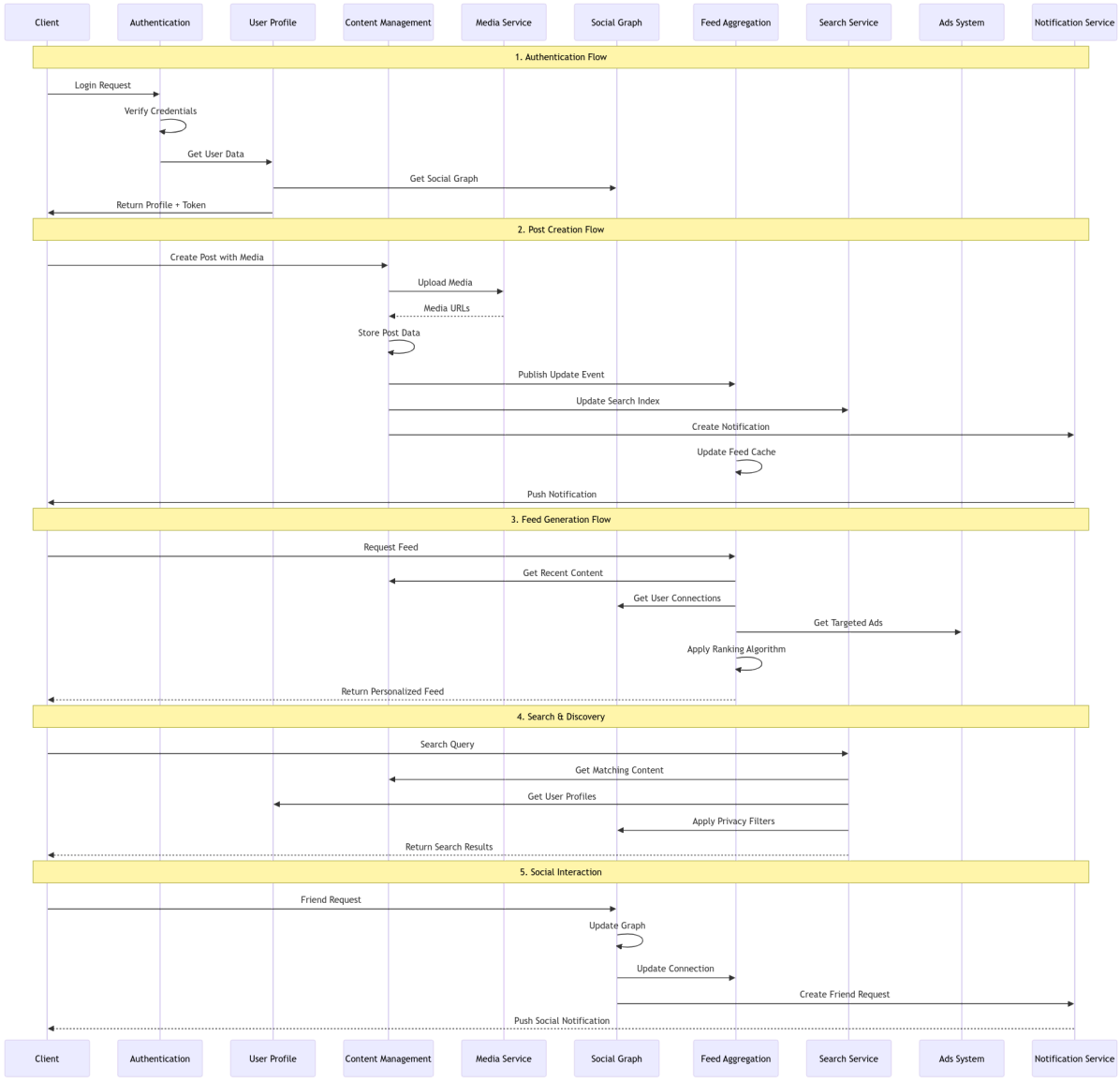


Figure-1 :Complete Facebook Service Interaction Flow

Feature-by-Feature Demonstration

1. User Authentication and Profile Management

Flow:

1. The **Authentication Service** receives a login or registration request.
2. User credentials are validated against the **User Database**.
3. On successful login, the **User Profile Service** loads profile data.
4. The **Privacy Service** applies user-configured privacy settings.
5. A session token (JWT) is generated and cached in Redis for fast validation during subsequent requests.

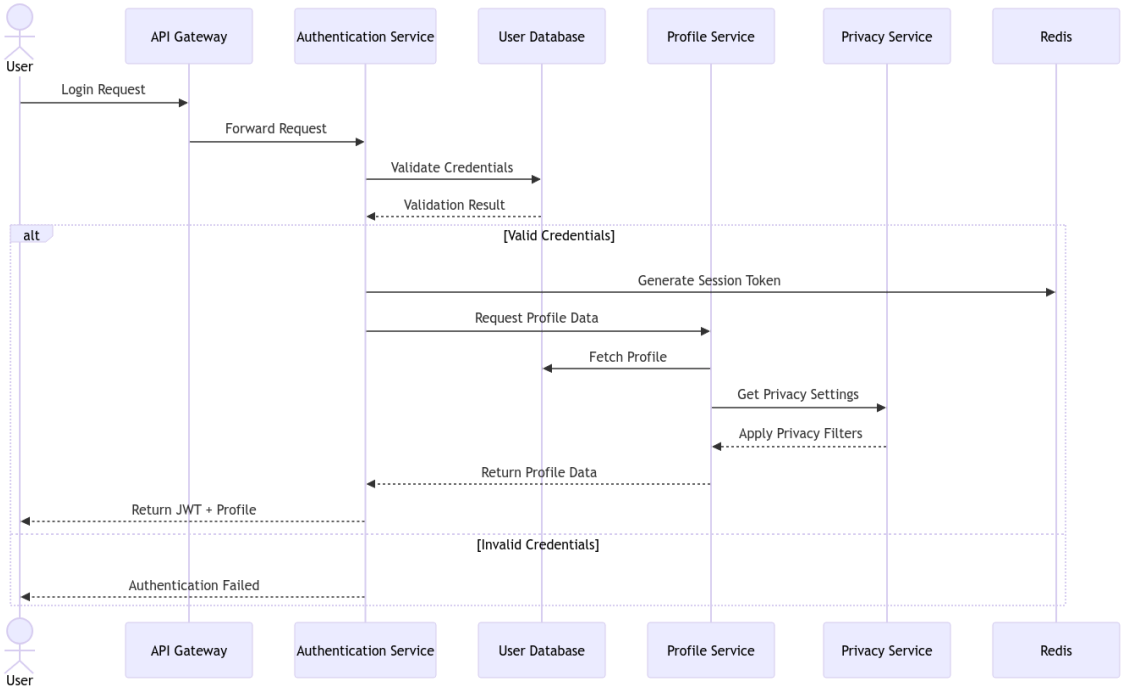


Figure-2 : Authentication and Profile Management Flow

Technologies:

- JWT for secure session management.
- Redis for caching session tokens.
- REST APIs for inter-service communication.
- CDC for syncing changes in the **User Database**.

2. Post Creation and Timeline Update

Flow:

1. **Content Management Service** processes a new post request.
2. Attached media (if any) is uploaded via the **Media Service** and stored in a scalable **Media Database**.
3. The **Post Content Database** is updated with the post details.
4. Events such as `PostCreated` are published to Kafka, triggering:
    - **Feed Aggregation Service**: Updates the user's timeline.
    - **Notification Service**: Notifies followers of the new post.
    - **Search Index DB**: Updates for better discoverability.
5. The **Content Moderation Service** checks the post asynchronously for policy violations.

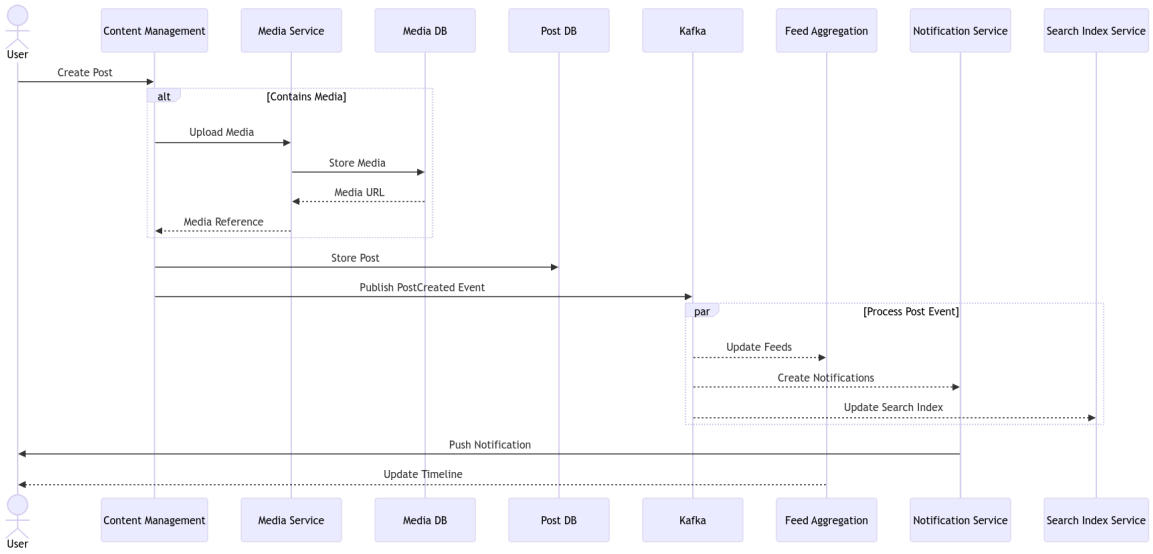


Figure-3 : Post Creation and Timeline Update Flow

Technologies:

- Kafka for event streaming.
- AWS S3 or blob storage for media handling.
- Elasticsearch for search indexing.
- CDC for propagating database updates.

3. Social Graph Operations (Friend Requests/Following)

Flow:

1. A friend request is sent to the **Friendship Service**.
2. The **Friendship Database** records the relationship.
3. Notifications are generated via the **Notification Service**.
4. **Feed Aggregation Service** updates the feed preferences.
5. **Privacy Service** adjusts content visibility settings.

Note: See the next figure of Feed Generation and Social Graph Flow

Technologies:

- Neo4j for managing the social graph.
- Kafka for event propagation.
- Redis for caching frequently accessed friendships.
- REST APIs for service communication.

4. Feed Generation and Content Ranking

Flow:

1. The **Feed Aggregation Service** fetches content from the **Post Content Database** and **Friendship Service**.
2. The **Recommendation Service** personalizes content based on user preferences.
3. The **Ranking Service** orders content by relevance and engagement potential.
4. The **Privacy Service** filters content according to user-defined rules.
5. The finalized feed is cached in the **Feed Cache Database** for efficient delivery.

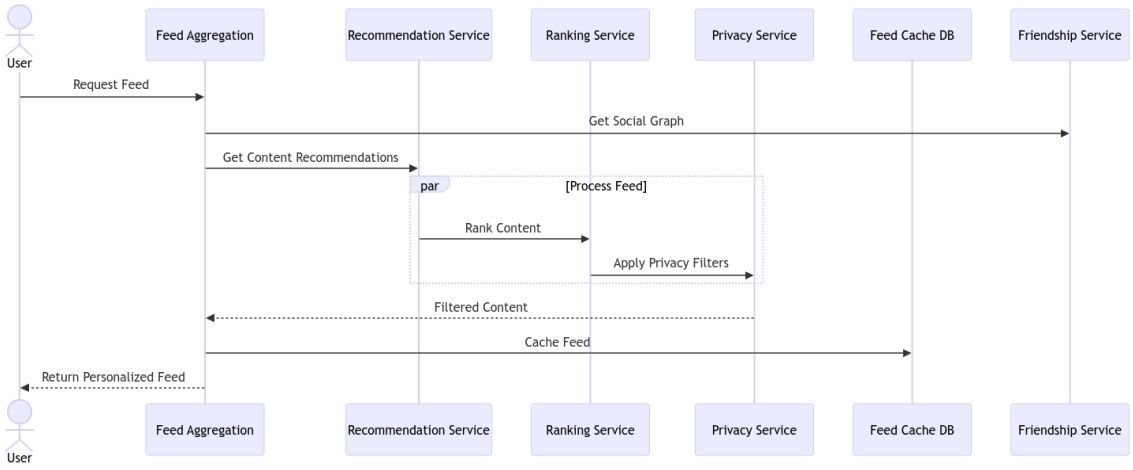


Figure-4 : Feed Generation and Social Graph Flow

Technologies:

- Redis for feed caching.
- gRPC for high-performance data retrieval.
- Machine learning models for personalization and ranking.
- CDC for real-time updates to the feed.

5. Messaging and Real-Time Communication

Flow:

1. The **Messaging Service** receives and processes messages between users.
2. Messages are stored in the **Message Database**.
3. Real-time updates are pushed to the recipient via the **Notification Service**.
4. The **Privacy Service** ensures that message visibility adheres to user settings.

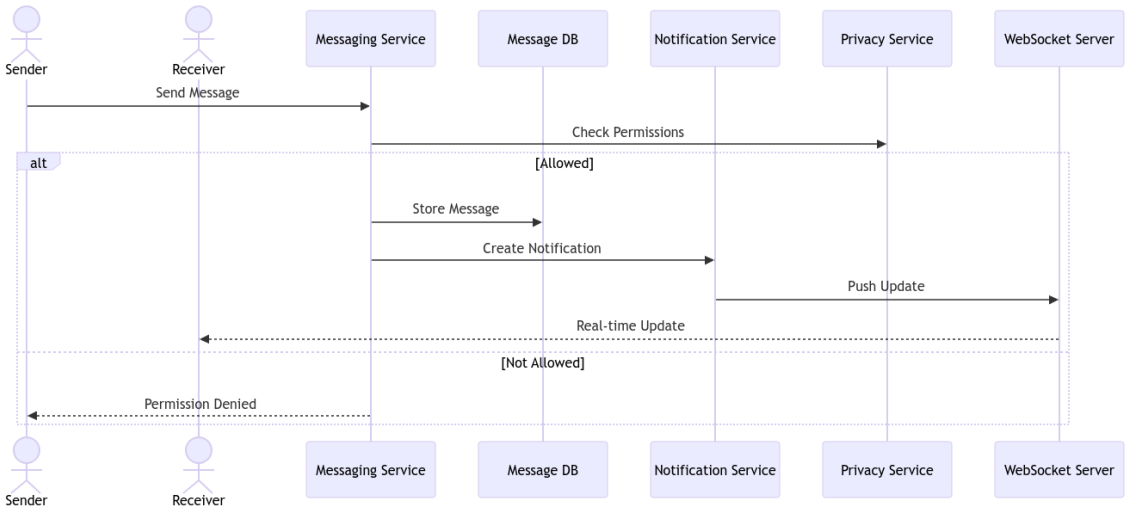


Figure-5 : Messaging and Real-time Communication Flow

Technologies:

- WebSocket for real-time communication.
- Redis Pub/Sub for low-latency notifications.
- RabbitMQ for message queuing and delivery reliability.
- CDC for synchronizing message updates.

6. Ad System Integration

Flow:

1. User activity data is sent to the **Behavior Analysis Service** via Kafka.
2. The **Ads System** processes targeting requests based on behavioral insights.
3. Interactions are logged in the **Event Log**.
4. The **Billing System** manages payment and charges for ad placement.
5. The **Marketplace Service** integrates commercial content into user feeds.
6. Ads are served to users by the **Feed Aggregation Service**.

Note: See the next figure of Ads System and Search Flow

Technologies:

- Kafka for event-driven data pipelines.
- Apache Spark for behavioral analysis.
- PostgreSQL for transactional data.
- Redis for ad targeting and caching.

7. Search Functionality

Flow:

1. The **Search Management Service** processes user queries.
2. It queries the **Search Index Database**, updated periodically via the **Content Management Service**.
3. Results are filtered through the **Privacy Service**.
4. Search results are aggregated from multiple sources, such as user profiles, posts, comments, and marketplace items.

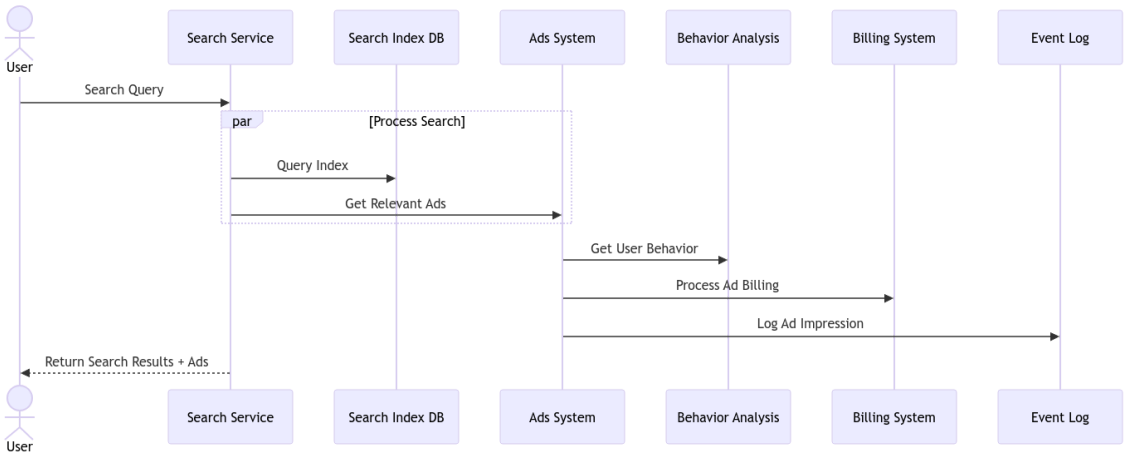


Figure-6 : Ads System and Search Flow

Technologies:

- Elasticsearch for full-text search.
- Redis for caching frequently searched results.
- gRPC for internal communication.
- GraphQL for flexible, aggregated data fetching.

Cross-Cutting Concerns

Data Consistency

- **Eventually Consistent Model**: Used for non-critical updates to ensure scalability.
- **Strong Consistency**: Maintained for sensitive operations like user authentication and financial transactions.
- **CDC**: Ensures real-time synchronization across services.

Security

- **mTLS**: Secures service-to-service communication.
- **API Gateway**: Handles external requests with rate-limiting and validation.
- **JWT**: Authenticates and authorizes users.

Monitoring and Logging

- **Distributed Tracing**: Managed using Jaeger for tracing service requests.
- **Prometheus**: Collects metrics for performance monitoring.
- **ELK Stack**: Aggregates and visualizes logs.

Error Handling

- **Circuit Breakers**: Ensure system resilience by isolating failing services.
- **Retry Mechanisms**: Implemented with exponential backoff.
- **Dead Letter Queues**: Handles undelivered messages gracefully.

Performance Optimization

- Multi-layer caching using Redis.
- Connection pooling for database interactions.
- Load balancing to evenly distribute traffic across service instances.