Experiment No: 3

Student Name: Kanishk Bhandari UID: 23BCS11100
Branch: BE CSE Section/Group: 23BCS_KRG-2_B
Semester: 6th Date of Performance: 28/01/2026
Subject Name: System Design Subject Code: 23CSH-314

1- **Aim** - To design a social media platform similar to Facebook or Instagram

2- **Requirements**: Functional & Non-Functional

A- Functional Requirement

- Client should be able to register and login to the application.
- Client should be able to create post (text / image / videos).
- Client should be able to follow each other (or send friend requests).
- Client should be able to like or comment on the post.
- Client should be able to view the feed of post from users they follow.

B- Non-Functional Requirement

- **Scalability:** The system should support 500 million daily active users.
- **Consistency & Availability:** For a social media application, high availability is prioritized over strict consistency, since temporary delays in data propagation are acceptable, but downtime is not.
- **Latency** (Uploading speed for publishing a post): The target latency is 500 ms for uploading a post.

3- **Core-entities of System**

- Users
- Posts
- Followers
- Like and comments
- Feed

4- **API endpoint creation**

A. User Onboarding APIs

1. **User Registration** POST /api/users/register
2. **User Login** POST /api/users/login
3. **Display User Data** GET /api/users/{user_id}/profile
4. **Update User Data** PUT /api/users/{user_id}/profile

B. User Posts APIs

1. **Create a Post** POST /api/users/{user_id}/posts
2. **Get a Post by ID** GET /api/posts/{post_id}
3. **Update a Post** PUT /api/posts/{post_id}
4. **Delete a Post** DELETE /api/posts/{post_id}

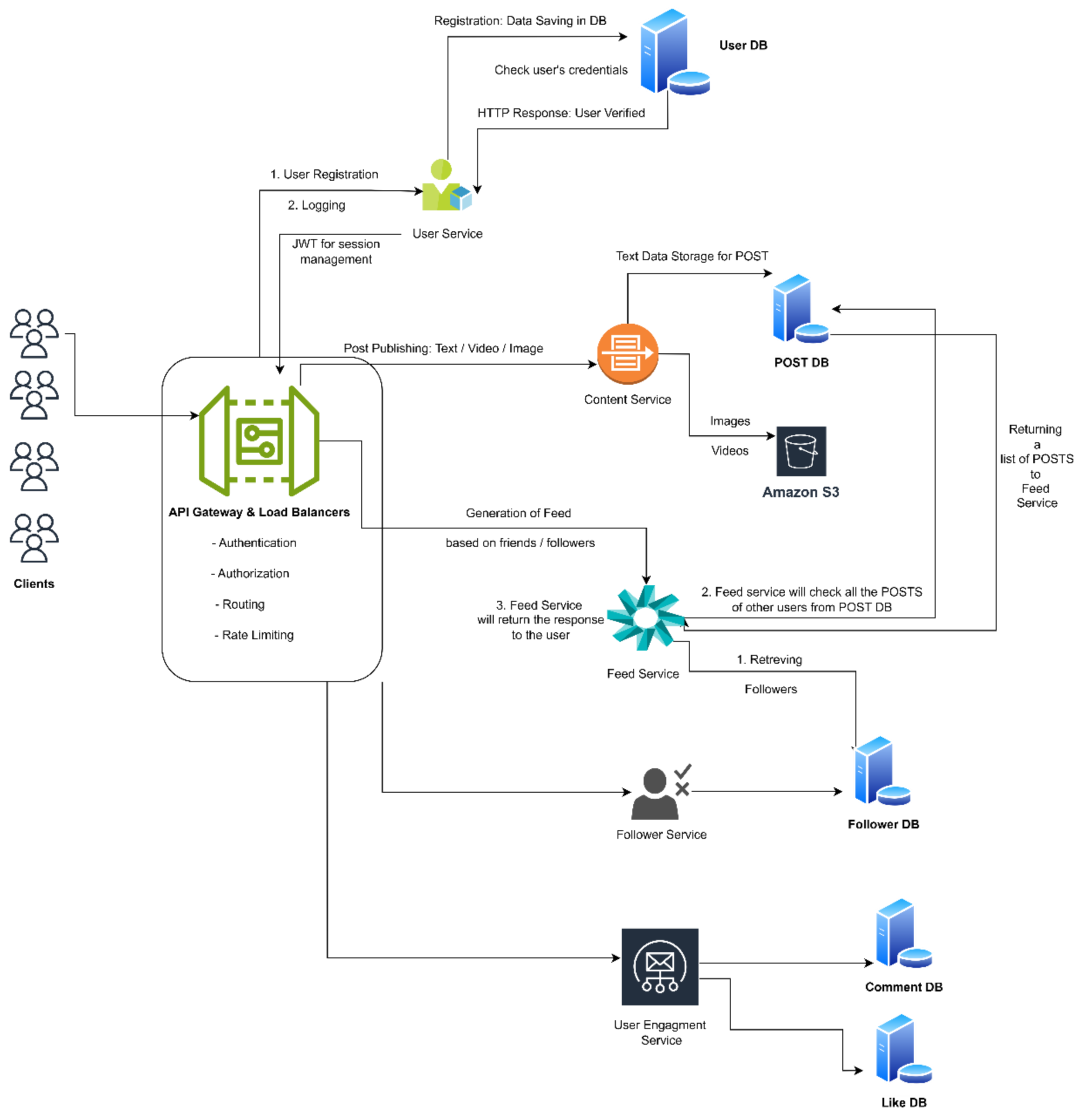
5. **Get Feed (Pagination)** GET /api/posts/feed?limit={limit}&offset={offset}
6. **Get Posts of a Specific User (Pagination)** GET /api/users/{user_id}/posts

C. User Interactions

1. POST/api/posts/ {post_id} / like
2. DELETE / api/posts/ {post_id} / unlike
3. POST/api/posts/ {post_id} / comments
4. GET/api/posts/ {post_id} / comments
5. PUT/api/ comments / {comment_id}
6. DELETE / api/comments/{post_id}/ {comment_id}
7. POST/api/ users / {user_id} / follow
8. DELETE / api / users / {user_id} / unfollow

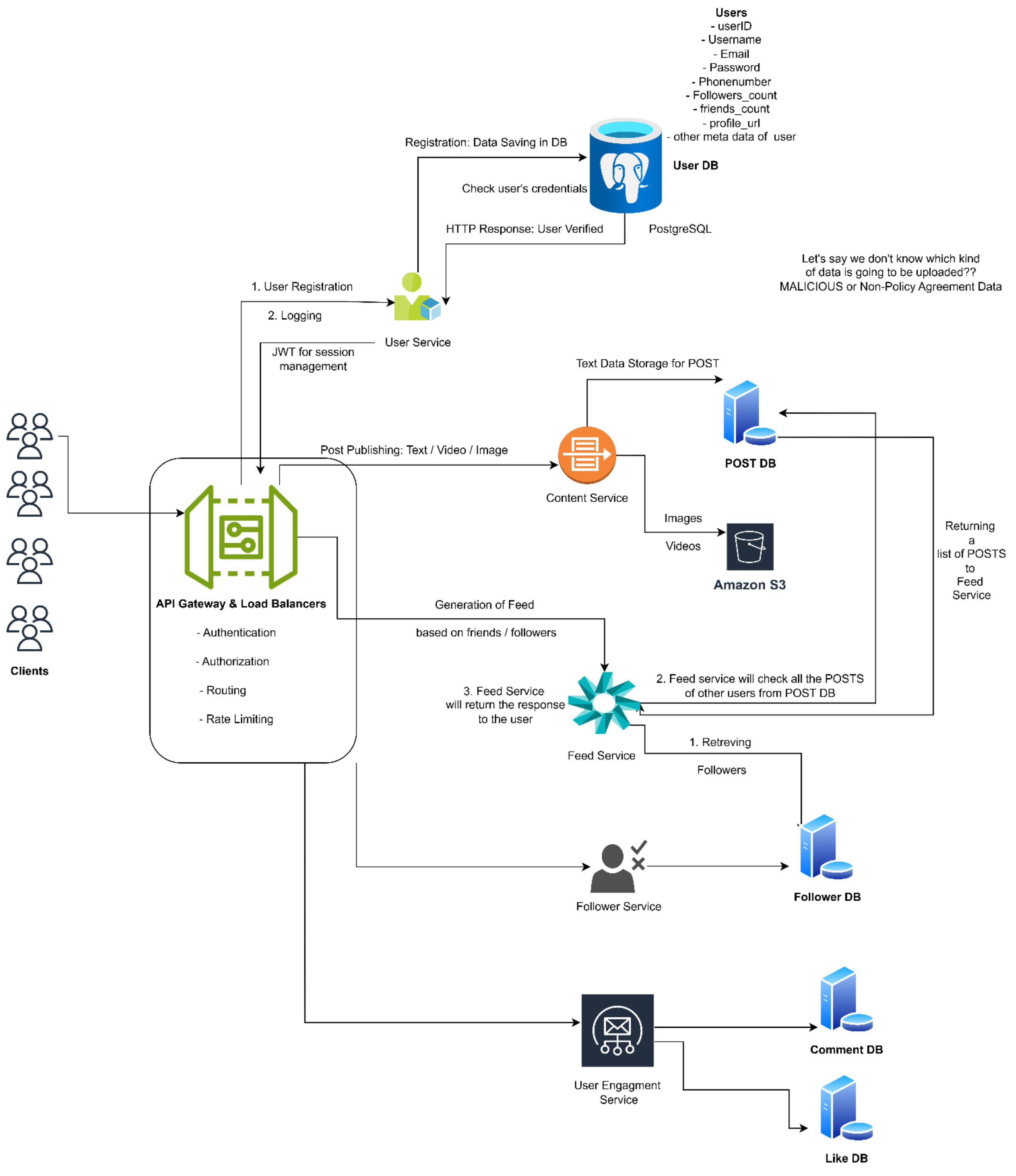
5- High-Level Design

Now According to the functional requirement of the system, we can identify that : We have to follow a distributed / micro-services approach not the monolithic one.

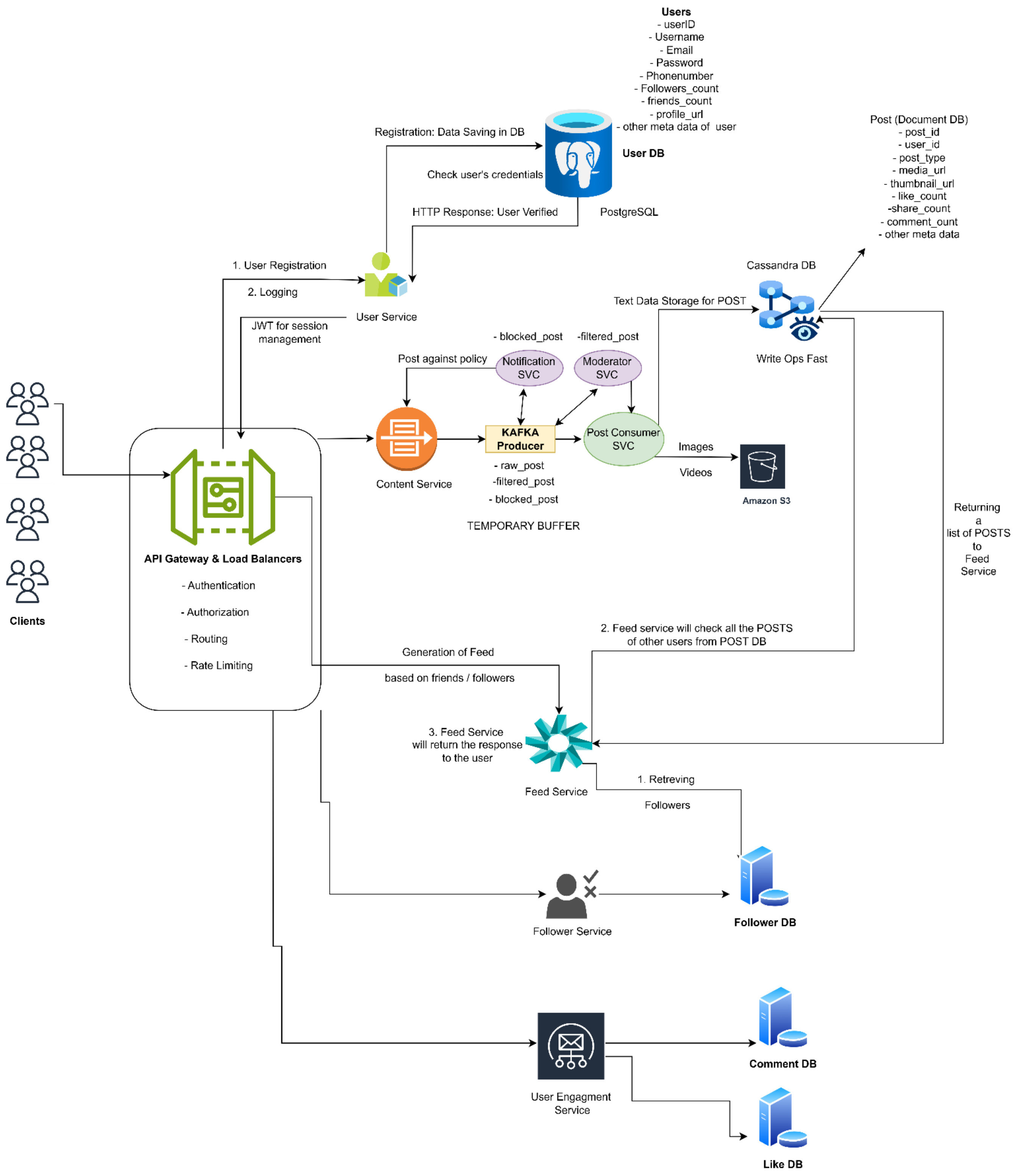

Registration: Data Saving in DB
User DB
Check user's credentials
HTTP Response: User Verified
1. User Registration
2. Logging
User Service
JWT for session management
Text Data Storage for POST
POST DB
Post Publishing: Text / Video / Image
Content Service
Images
Videos
Amazon S3
Returning a list of POSTS to Feed Service
Clients
API Gateway & Load Balancers
- Authentication
- Authorization
- Routing
- Rate Limiting
Generation of Feed
based on friends / followers
2. Feed service will check all the POSTS of other users from POST DB
3. Feed Service will return the response to the user
Feed Service
1. Retreving
Followers
Follower Service
Follower DB
Comment DB
User Engagment Service
Like DB

6- Low- Level Design

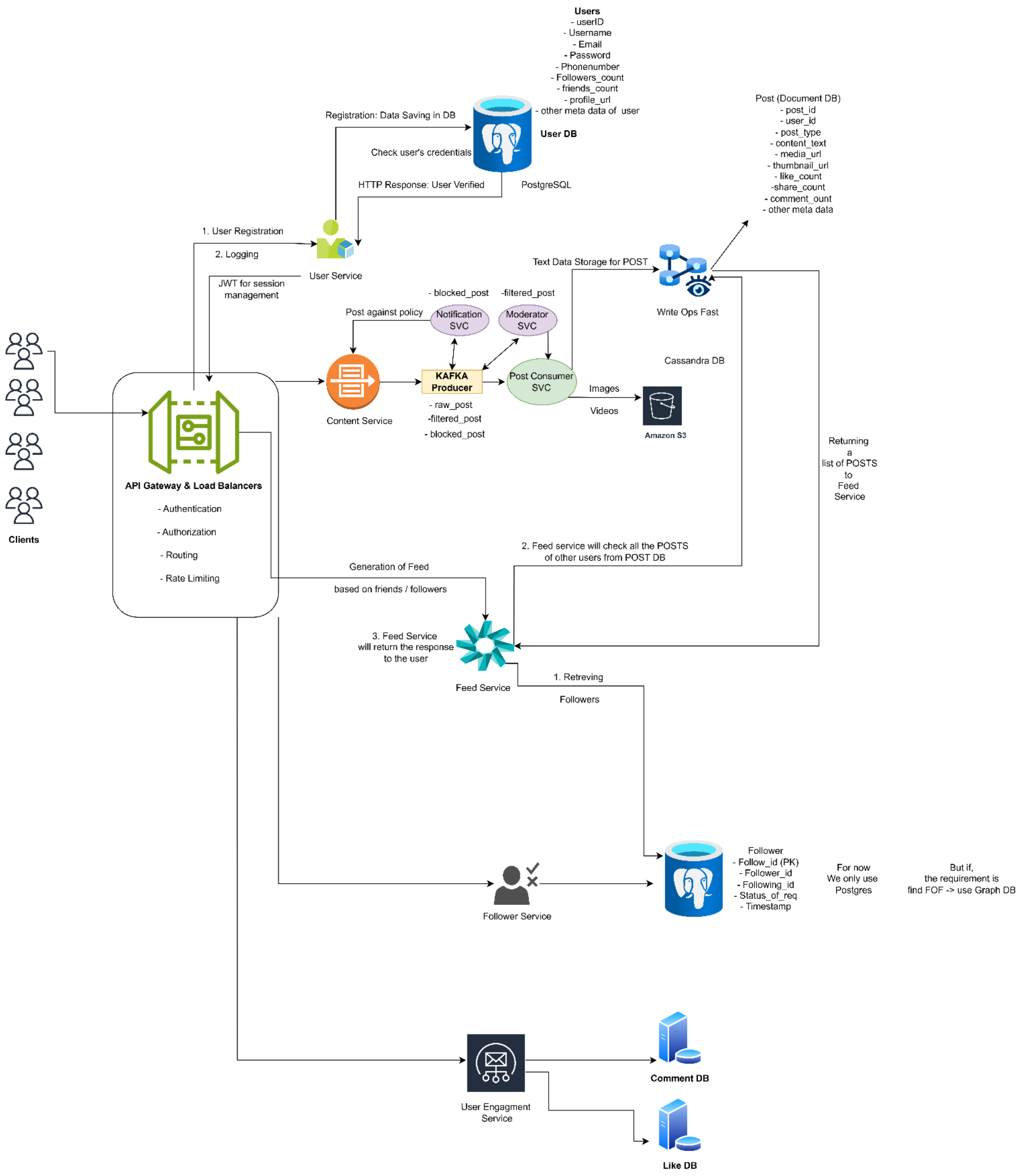
USER SERVICE


Users
- userID
- Username
- Email
- Password
- Phonenumber
- Followers_count
- friends_count
- profile_url
- other meta data of user
Registration: Data Saving in DB
Check user's credentials
User DB
HTTP Response: User Verified
PostgreSQL
Let's say we don't know which kind of data is going to be uploaded?? MALICIOUS or Non-Policy Agreement Data
1. User Registration
2. Logging
User Service
JWT for session management
Text Data Storage for POST
POST DB
Post Publishing: Text / Video / Image
Content Service
Images
Videos
Amazon S3
Returning a list of POSTS to Feed Service
API Gateway & Load Balancers
- Authentication
- Authorization
- Routing
- Rate Limiting
Clients
Generation of Feed
based on friends / followers
2. Feed service will check all the POSTS of other users from POST DB
3. Feed Service will return the response to the user
Feed Service
1. Retreving
Followers
Follower Service
Follower DB
Comment DB
User Engagment Service
Like DB

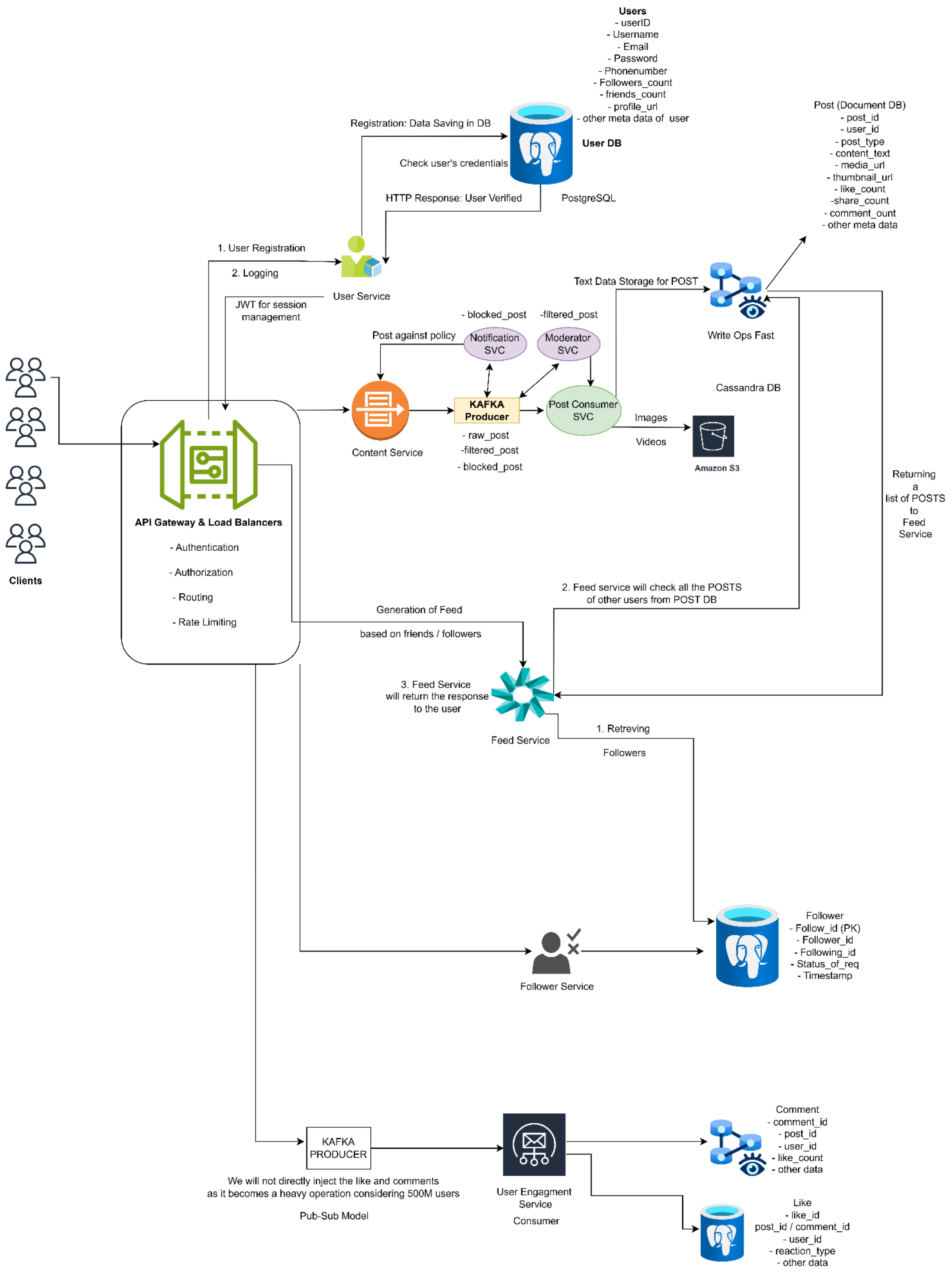
CONTENT SERVICE



FOLLOWER SERVICE



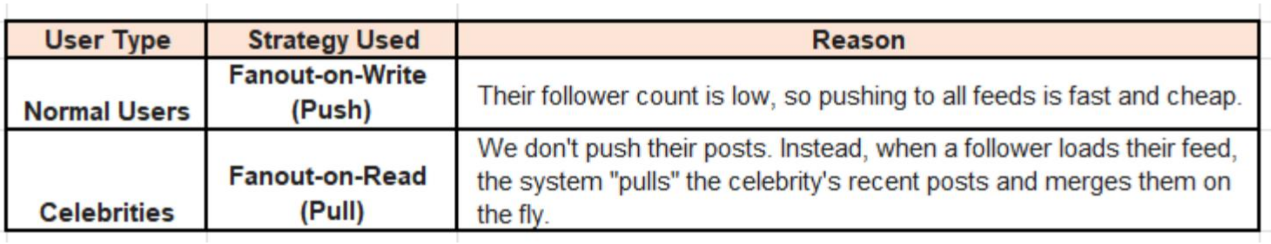
ENGAGEMENT SERVICE



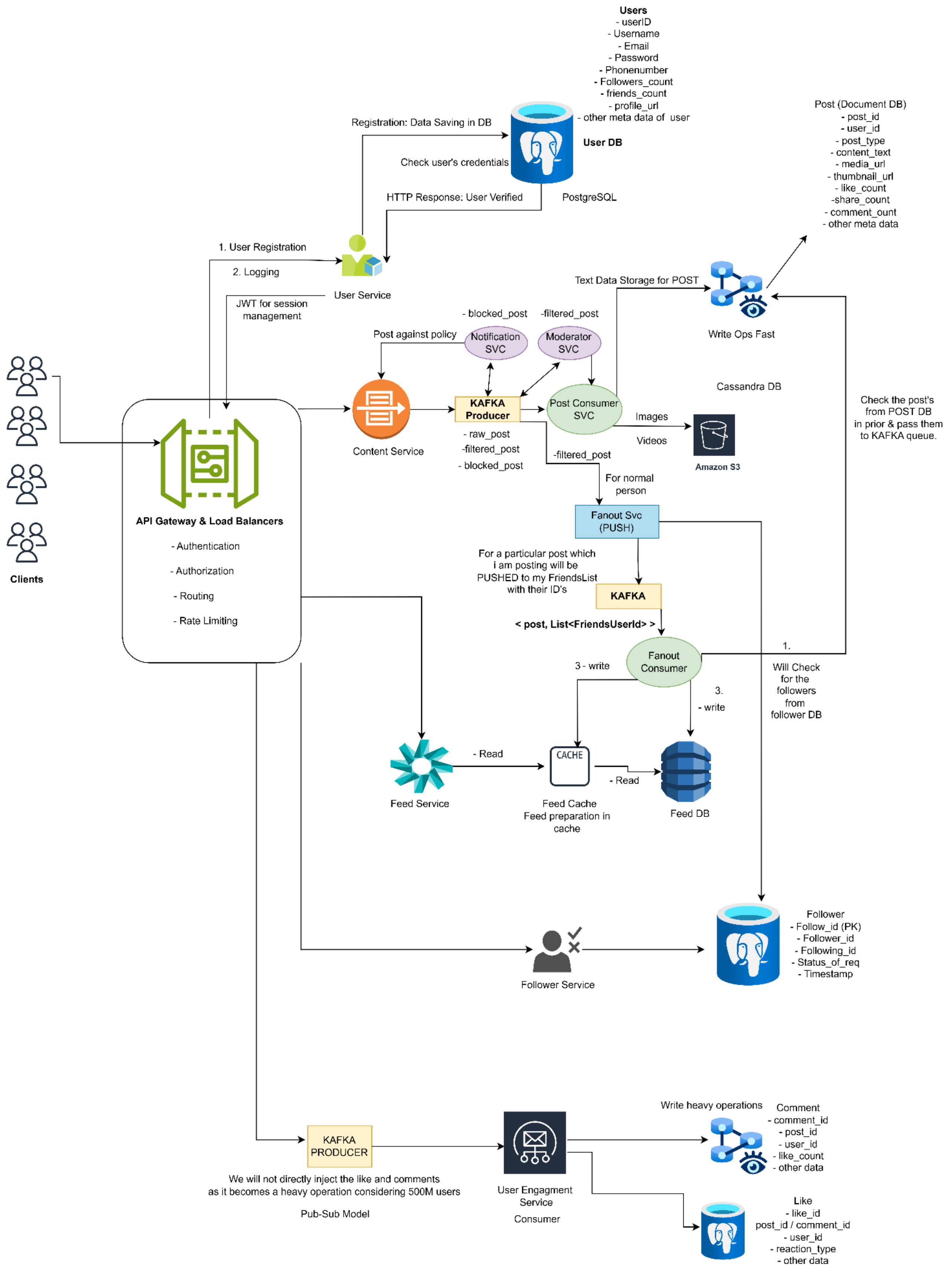
FEED SERVICE
Problems With Feed Service
Feed service has to check data again and again from two databases, i.e., POST DB and FOLLOWER DB
Which cost hight database read / write operations, every time we have to generate the feed.

What if i get some of the pre-computed data for feed service rather than going again and again to POST DB and FOLLOWER DB

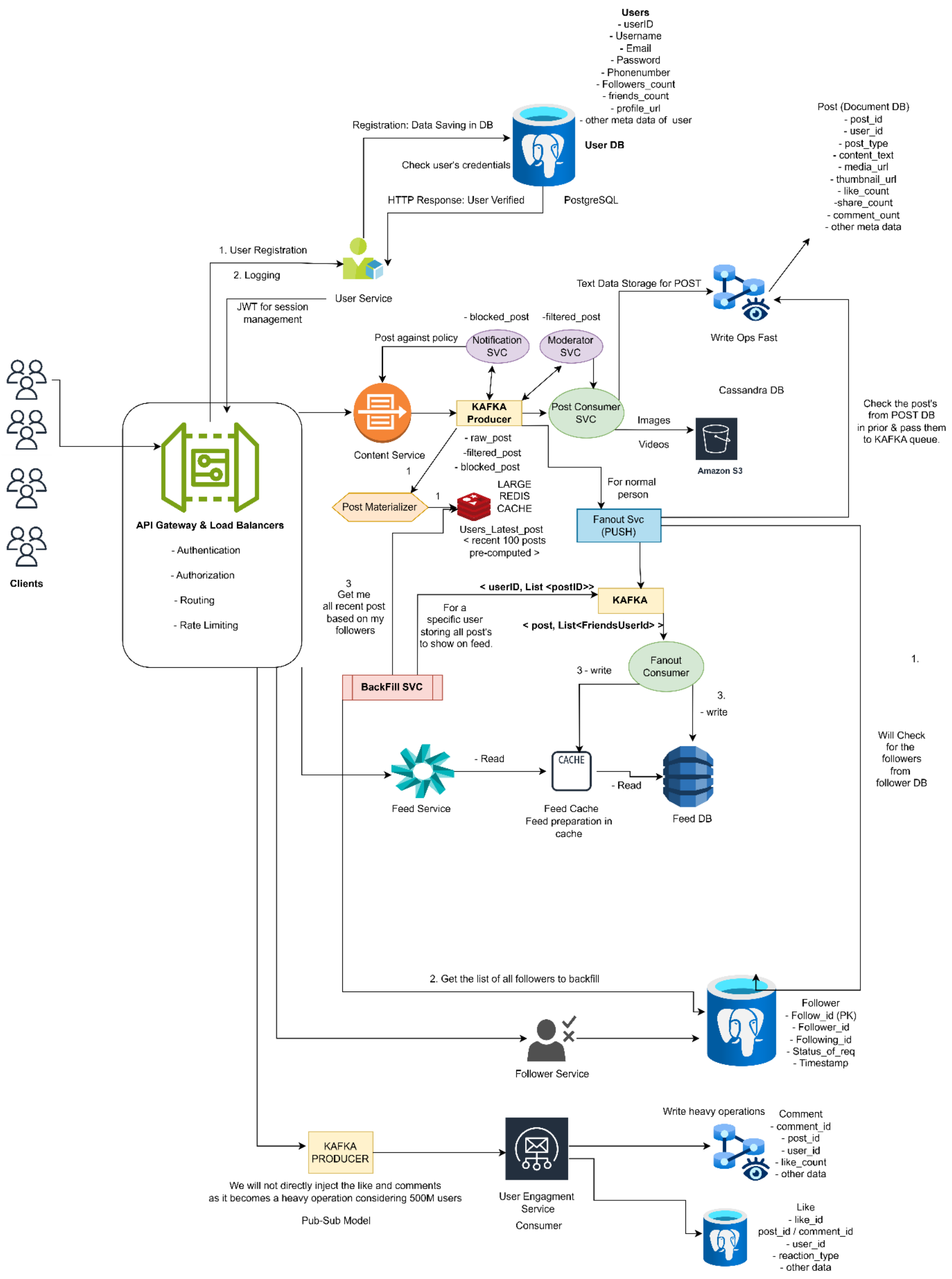
Now this is a common problem while designing the apps like social media & the solution is: **FANOUT MODEL**



User Type	Strategy Used	Reason
Normal Users	Fanout-on-Write (Push)	Their follower count is low, so pushing to all feeds is fast and cheap.
Celebrities	Fanout-on-Read (Pull)	We don't push their posts. Instead, when a follower loads their feed, the system "pulls" the celebrity's recent posts and merges them on the fly.



Problem: Whenever a new feed is generated after 50 posts, we have to follow the same process. So, when a user is about to finish seeing all the posts. We will use a concept called : BACKFILL using a MATERIALIZER

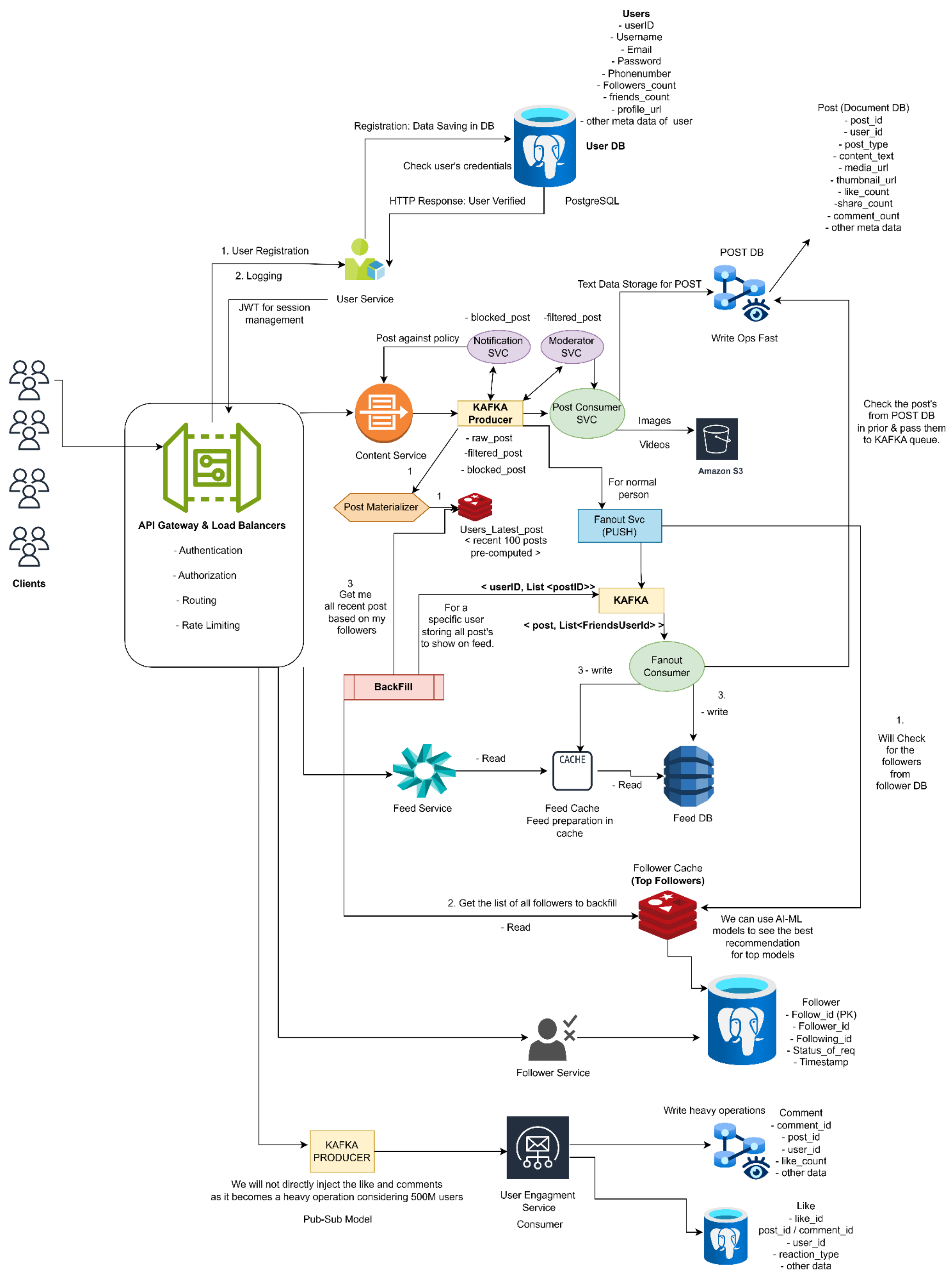


Post Materializer

1. It stores the recently scrolled posts of a user in the Redis cache.
2. The Backfill service retrieves the list of followers from the Follower Database.
3. Based on these followers, it fetches their recent posts from Redis.
   This approach avoids frequent access to the Post Database, enabling real-time feed generation with lower latency.

Backfill
When a user is about to consume the entire feed (for example, 50 posts), an additional API call is triggered to refill the feed using the Post Materializer and Redis cache.

In an app like social-media, we don't need data for everyone, we need data or post of the one's we follow the most.
Instead, of referring to the FOLLOWER DB again & again, we can maintain a cache of top followers we follow.


Users
- userID
- Username
- Email
- Password
- Phonenumber
- Followers_count
- friends_count
- profile_url
- other meta data of user
Registration: Data Saving in DB
User DB
Check user's credentials
HTTP Response: User Verified
PostgreSQL
Post (Document DB)
- post_id
- user_id
- post_type
- content_text
- media_url
- thumbnail_url
- like_count
-share_count
- comment_ount
- other meta data
1. User Registration
2. Logging
JWT for session management
User Service
POST DB
Text Data Storage for POST
Write Ops Fast
- blocked_post
-filtered_post
Post against policy
Notification SVC
Moderator SVC
KAFKA Producer
Post Consumer SVC
Images
Videos
Amazon S3
Check the post's from POST DB in prior & pass them to KAFKA queue.
Clients
API Gateway & Load Balancers
- Authentication
- Authorization
- Routing
- Rate Limiting
Content Service
1
- raw_post
-filtered_post
- blocked_post
For normal person
Post Materializer
1
Users_Latest_post
< recent 100 posts pre-computed >
Fanout Svc (PUSH)
3
Get me all recent post based on my followers
< userID, List <postID>>
KAFKA
For a specific user storing all post's to show on feed.
< post, List<FriendsUserId> >
Fanout Consumer
3 - write
3.
- write
BackFill
1.
Will Check for the followers from follower DB
Feed Service
- Read
CACHE
- Read
Feed Cache
Feed preparation in cache
Feed DB
Follower Cache
(Top Followers)
2. Get the list of all followers to backfill
- Read
We can use AI-ML models to see the best recommendation for top models
Follower
- Follow_id (PK)
- Follower_id
- Following_id
- Status_of_req
- Timestamp
Follower Service
Write heavy operations
Comment
- comment_id
- post_id
- user_id
- like_count
- other data
KAFKA PRODUCER
User Engagment Service
Consumer
We will not directly inject the like and comments as it becomes a heavy operation considering 500M users
Pub-Sub Model
Like
- like_id
post_id / comment_id
- user_id
- reaction_type
- other data