Architecture and High level System Design for a Social Shopping Application

The following document outlines the overall architecture and high level design of a typical Social Shopping application. The document is divided into four sections in the following order:

- ➢ Personas
- ➢ Functional & Non-Functional Requirements
- ➢ Database & Object storage Capacity estimation
- ➢ High-level API and DB design
- ➢ Application flows
- ➢ Architecture block diagram
- ➢ Monitoring and Alerting

Personas

1. Influencer - Typical designers/celebrities/regular shoppers who has the potential to influence regular customers
2. Customers - End user of the application
3. Organization - Who develops the application and launch in the market

Functional & Non-Functional Requirements:

Functional:

1. Influencers should be able to login/signup into the application
2. Influencers should be able to upload/post photos and tag the products used in the photos.
3. Look-books should be maintained for each influencer
4. Customers should be able to login/signup into the application
5. Customers should be able to follow the influencers
6. Customers should be able to view the look books of their followed influencers
7. Customers should be able to place the order for the products from the look books of the influencers
8. Track the leads from the lookbooks of the influencers to share the revenue with influencers
9. Payment should be made to the influencer based on the share percentage

Non-Functional:

1. Latency : All the requests should have low latency

2. Availability : All the components of the system should be highly available
3. Consistency : Eventual consistency of the total System

Design Choice**:**

- Availability will be given more preference over consistency due to the reason that the user can see the post by an influencer eventually but the system should be highly available throughout the time, as the availability directly impacts revenue in the case of e-commerce.

DataBase & Object storage Capacity estimation

- Assuming the approximate bytes that each row occupies in the DataBase as follows:
    - Influencer → 256 Bytes
    - Customer → 256 Bytes
    - Follower → 128 Bytes
    - Product → 512 Bytes
    - Order → 1024 Bytes
    - Post → 2048 Bytes
    - Transaction → 256 Bytes

- Projected Operations per day:
    - 1000 Influencers with 100K followers for each influencer
    - 1M Customers
    - Followers → 1000 * 100K
    - Estimation of 100K Products
    - Projected 10K Orders per day
    - Projected 1000 Posts per day
    - Estimated 1000 Transaction per day

- DB Total Estimated Capacity:
    - For the capacity estimation, we'll calculate per year (365 days):
    - Formula: (1000*0.25KB) + (1,000,000 * 0.25KB) + (1000*100,000*0.125KB) + (100,000 * 0.5KB) + (10,000 *1KB *365) + (1000*2KB*365) + (1000*0.25*365)
    - **~ 16 GB per year**
    - Estimating 1 replica , **~ 32 GB per year**
- Object storage Capacity Estimation:
    - The images of the product, user profile, influencer post can be stored in an object store for efficient retrieval and costs.
    - Assume the average image size is 128 KB
    - Assume each influencer will post 5 images per post.

High level API and DB design

<u>Microservices:</u>

<u>Influencer Service</u>

- InfluencerSignUp()
- InfluencerLogin()
- AddPostToLookBooks()
- GetAllPostsForInfluencer()
- FetchRevenueDetailsForPost()

<u>Customer Service</u>

- CustomerSignUp()
- CustomerLogin()
- FollowInfluencer()

<u>Order Service</u>

- PlaceOrder()
- CheckOrderHistory()
- CheckOrderStatus()
- UpdateOrderStatus()

<u>Product Service</u>

- GetProductDetails()
- GetAllProducts()
- AddProducts()
- RemoveProducts()

<u>LookBookService</u>

- AddPostToLookBooks()
- GetAllPostsForInfluencer()

<u>PaymentService()</u>

- InitiatePayment()
- FetchPaymentStatus()

API Design Considerations:

a. An empty lookbook will be created for each influencer upon the first sign-up

<u>Database Design:</u>

- Design choice would be SQL Database as the data is well structured and APIs can use SQL queries to fetch and update the data
- Images are stored in the Objected storage like S3 and the URLs will be stored in the Tables.
- *To give a better and concise view, I'm giving the table names and keys here - but if required I can explain them in detail.*

<u>Influencer Table</u>
- InfluencerId (Key)
- <OTHER COLUMNS>

<u>Customer Table</u>
- CustomerId (Key)
- <OTHER COLUMNS>

<u>Follower Table</u>
- InfluencerId
- CustomerId

<u>Products Table</u>
- ProductId (Key)
- <OTHER COLUMNS>

<u>LookBook Table</u>
- LookBookId (Key)
- InfluencerId
- <OTHER COLUMNS>

<u>Posts Table</u>
- PostId (Key)
- InfluencerId
- PostContentDetails
- ProductIdList
- <OTHER COLUMNS>

<u>Orders Table</u>
- OrderId (Key)
- CustomerId
- <OTHER COLUMNS>

<u>OrdersList Table</u>
- OrderId
- ProductId
- PostId // Represents the Id post from which the lead was generated.
- Total Amount
- <OTHER COLUMNS>

<u>Transaction Table</u>
- TransactionId
- InfluencerId
- CustomerId
- Amount

<u>Scaling & Sharding:</u>

- For Horizontal Scaling, leverage the Auto Scaling Group for the compute resources based on the load
- For Vertical Scaling, we can promote to bigger instances based on the load and # of transactions.
- Implement Level-1 Sharding (AZ based) for DBs
- Replication - For the 1st release, we can opt for Master Slave configuration (2 replicas), where writes can be made to Master and reads can be done from Slave node

<u>Caching:</u>

- Latest posts from the influencer can be cached as there is a high chance of customers following the influencers and going through their latest posts.
- LRU eviction policy can be used to remove the posts from the cache

Application flows

<u>Influencer:</u>

- Can Signup and Login to the application
- Can post a new photo and tag different products on the photo.
- Query the product table to fetch all the relevant products.
- Query the order table to get the # of leads generated from the post. PostId is stored against each of the line items in the OrderList table. Total amount of order placed from a post can be aggregated and percentage share for all the leads generated can be shown to the influencer.
- Check the transaction status of their transaction from the transaction table.

<u>Customers:</u>

- Can Singup and Login to the application
- Follow other influencers
- Fetch the latest posts of the influencers they follow
- Get product details by clicking on the product added in the post
- Add products into the shopping cart
- Place an order for the products added in the cart
- Check the order status as well as order history

<u>Offline Job:</u>

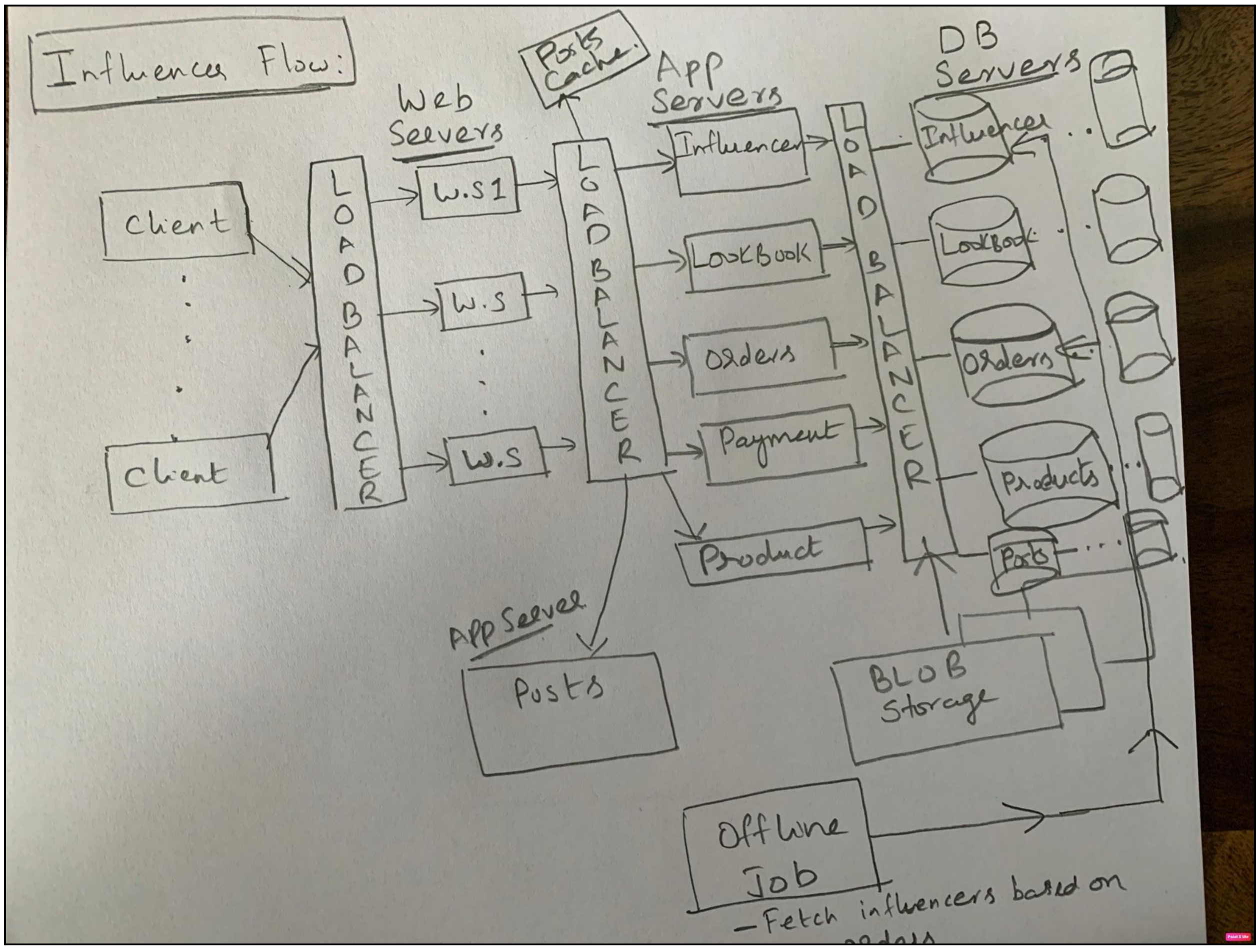
- Set up an **Offline Job** (InfluencerAmountJob) which will run at the end of the day to settle the amount with the influencer by making the payment based on the percentage decided for the lead.
  - Offline Job will be executed at the end of the day and will fetch all the orders created in a day.
  - For each of the orders, the order item list will be fetched and for each of the post Id total amounts will be maintained in the memory.

    - Once the total amount for each of the posts is aggregated, influencer Id will be fetched for each of the posts and the amount will be aggregated for the influencer Id.
    - Amount to be paid will be determined for each of the influencers based on the percentage.

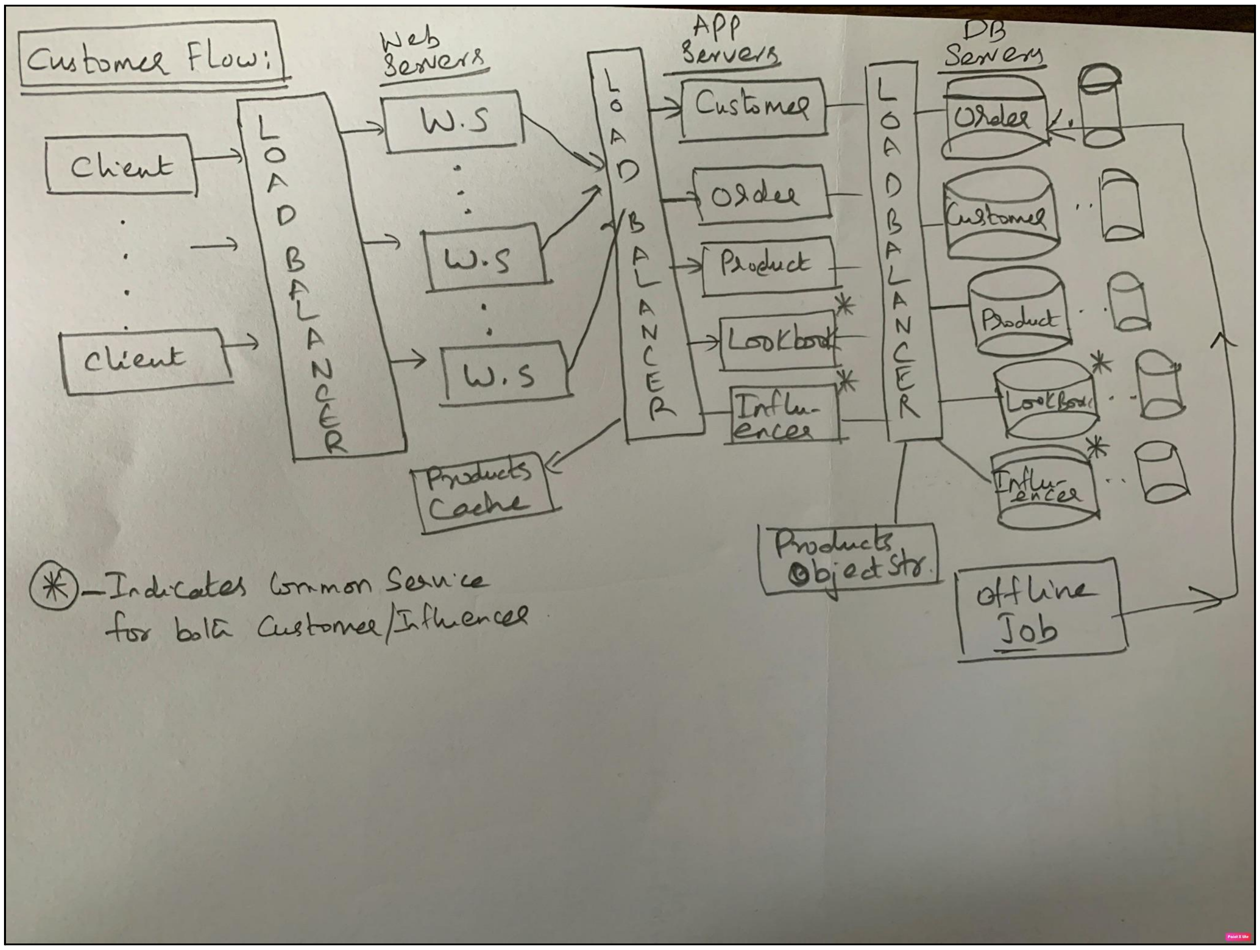
<u>Influencer Performance and Payment:</u>

- Periodical performance can be calculated for the influence - Query the total orders generated for each of the posts and then join the influencer Id in the post table
- Payment Share for Influencer - Based on the percentage share and lead generated with each influencer, the metrics can be created to show the influencer earnings.

Architecture Block Diagram - Influencer Flow



Architecture Block Diagram - Customer Flow



Monitoring and Alerting

- Monitor Latency of each of the API
- Monitor Error rate of each of the API
- Monitor Query Per Second (QPS) and resource usage for each of the API/service to scale up/scale
- Alerts on Errors → Send an alert if a request fails for “n” times in a time frame
- Alerts on latency → Send an alert if “n%” of total requests within a timeframe is taking more than “t” milliseconds to return the results