Design a photo sharing app like Instagram

Functional Requirements –

1. Create Posts with media & caption.
2. Follow users
3. View a feed of friends posts in chronological order.

Out of scope

- like, comment on posts.
- search for user, hashtags or locations
- User should be able to go live

Non functional Req –

1. Highly available (Availability >> Consistency)
2. Low latency feed loading (< 500ms)
3. Low latency media rendering (< 200ms)
4. Scalable to 500M DAU.

Out of scope

- secure & Protect user data (authentication, authorization, encryption)
- fault-tolerant & highly reliable (no data loss)
- analytics on user behaviour & engagement

Core Entity –

1. User
2. Post
3. Media
4. Follow

APIs –

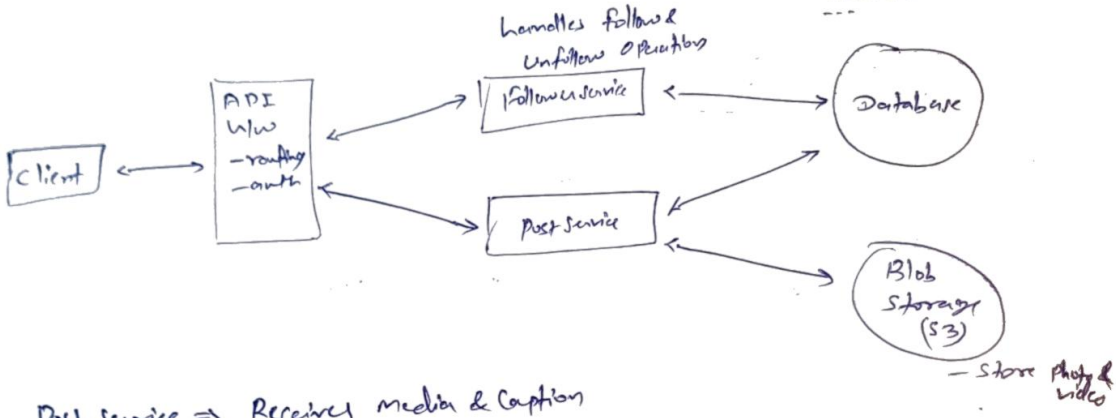
1. POST /posts → postId
   {"media": {photo or video bytes},
   "caption": "my cool photo"
   }
2. POST /follows
   {"followedId": "123"}
3. GET /feed?cursor={cursor}&limit={pageSize}
   → Post[]

HLD

(1) Create post & (2) follow user — (3) view feed

Post
- id
- UserId
- S3link
- Caption
- CreatedAt

Follower
- FollowerId
- FollowedId



Post Service ⇒ Receives Media & Caption
↓
Stores the Post Metadata in DB
+
Store actual bytes in S3

For Feed –
* Query 'follow' table to get a list of UserId's that the current user follows.
* For each of those followed user, query Post table.
* Merge & Sort all Posts
* Return Sorted Posts to the client.

Back to Indexing, given we're opting for DynamoDB, we'll need to add a few indexes to our tables to avoid full table scans.

(a) For 'Follower' table –

Partition Key ⇒ FollowerId
Sort Key ⇒ FollowedId
⇒ Allows us to efficiently query for all users that a given User follows.

(b) For 'Post' table –

Partition Key ⇒ UserId (Since most of our queries will be to get the Posts of a given User.

Sort Key ⇒ Composite of (CreatedAt & PostId) ⇒ (to ensure chronological ordering while maintaining Uniqueness.

Potential Deep Dive

(1) The system should deliver Feed Content with low latency (< 500ms) :-

→ Problem with Fan-out on Read:-

* Model:- Fetch all Post from followed account when user request the Feed.

* Issues:
  - Thousands of DB reads per request.
  - Same Posts fetched multiple time for similar follows.
  - Unpredictable Latency:- Varies by follow Count & activity.

* Scale Problem:
  - 500M DAU * 5 Feed refreshes = 2.5B reads/day
  - Peak: 150K Feed requests/sec
  - Cannot meet 500ms latency target reliably.

* Conclusion:→ Doesn't Scale; Computational work at read time is inefficient

→ Bad solution — Feed Generation with simple Cache —

* Add a Redis cache layer before querying DB.
* Key format: `Feed:{userId}:{cursor}` → list of PostIds
* Pros:→ Faster reads for cache hits.
* Cons:
  - Cache size must be ~~enormous~~ enormous to be effective.
  - Still performs expensive Fan-out reads on Cache misses.
* Conclusion: Helps but doesn't solve root problem; Still scales poorly.

→ Good solution : Precompute Feeds (Fan-out on write) :-

* Idea:- Generate Feed when post is created, not when it's read.
* Flows → (i) New post → Get all followers (via GSI on follows table)
  (ii) For each follower → Add postId to their Redis Feed (ZSET with time stamp)

* Feed Retrival –
  - → Get top N PostIds from Redis.
  - → Hydration options:–
    1) Query each PostId from DB
    2) Cache full Post metadata in Redis
    3) Hybrial: Metadata Cache + batch fallback to DB.

* Conclusion – Fast reads, write-heavy system now. Enables consistent low-latency reads.

⇒ challanges with Fan-out on write –
- → write Amplification (specially for celebrities).
  - → millions of writes for one celebrity Post
  - → Redis storage bloat + system overload risk.

⇒ Great solution : Hybrial Approach (Precompute + Realtime) :–

* Approach → Fan out on write for regular users (< 100K followers)
  - → Fan out on Read for celebrities (≥ 100K followers)

* Feed Retrival → Get Precomputed Posts from Redis.
  - → Merge with recent posts from celebrities (queried in real-time from DB)

* Benifits → Reduces write amplification.
  - → Keeps read latency low for most users

* Challanges → Need to tune celebrity threshold (eg. 100K followers)
  - → Added system complexity
  - → Inconsistency experience for celeb-heavy users.

* Conclusion → Practical trade-off: Prove effective at Instagram scale.

NOTE - Redis Reliability Considerations

* Redis is in-memory & not inherently durable.
* Mitigation Strategies:
  1) High availability → Redis Sentinel for failover.
  2) Durability → AOF (Append-only file) or RDB (Snapshot) persistence.
  3) Sharding → Redis cluster to distribute load.
* Interview tip → Always mention durability + failover plans. (don't treat Redis as magical cache).

(2) Media upload & rendering for large photos/videos :-

We need to → instant rendering of media
[max size → photos up to 8 MB & video up to 4 GB]

* two major challenges → Efficient uploads
  → Low latency downloads/rendering

(A) Uploading Large Media :-

Problem :- HTTP requests are limited (< 2 MB). need chunked upload.

Solution :- Multipart upload to S3.

→ * Use AWS S3 multipart upload API —
  (i) POST /posts create metadata & return a Pre-signed S3 url.
  (ii) client uploads file in 'chunks' to S3 directly.
  (iii) S3 assembles chunks into final file.

→ * Post Metadata update —
  (i) media initially marked as 'pending'.
  (ii) Two options to update metadata after upload :-

(a) client-driven ⇒ Client calls 'PATCH' with 'object_key' & 'complete' status.
(Can't rely much on client)

(b) server-driven ⇒ S3 triggered a 'Lambda' to update Metadata.

(Conclusion → server-driven offers better consistency).

(B) Serving Media (Rendering) :-

(i) Naive approach - Direct S3 serving -

* clients download media directly from S3.
* Problems :-
  - → High latency for users far from S3 region.
  - → No caching → high cost
  - → one-size-fits-all delivery (wasteful on slow/mobile devices)

(ii) Good soln - Using CDN (e.g cloudfront) -

* CDN caches media at global edge locations
* Benifits →
  - lower latency
  - Reduced load on S3
  - Supports file-type-based caching & Compression.
* Limitations →
  - Still serves the same file to all users, regardless of device/network.
  - Cache evictions can cause origin fetches.

(iii) Great soln → CDN + Dynamic Media Optimization

* Generate 'multiple media variants' (resolutions, formats) on upload
  - use tools like cloudinary, Imgix.
  - Adaptive Streaming for videos

* Device-aware delivery :-
  - client sends device/network info.
  - CDN serves optimized version (eg webP on mobile)

* Advanced caching:-
  - Popular media cached aggressively.
  - Proactive cache warming for trending content

* challanges:-
  - more storage (multiple versions)
  - Media Processing pipeline adds complexity.

* Conclusion: Best performance & scalability; used by large platforms.

(3) System should be scalable to support 500M DAU.

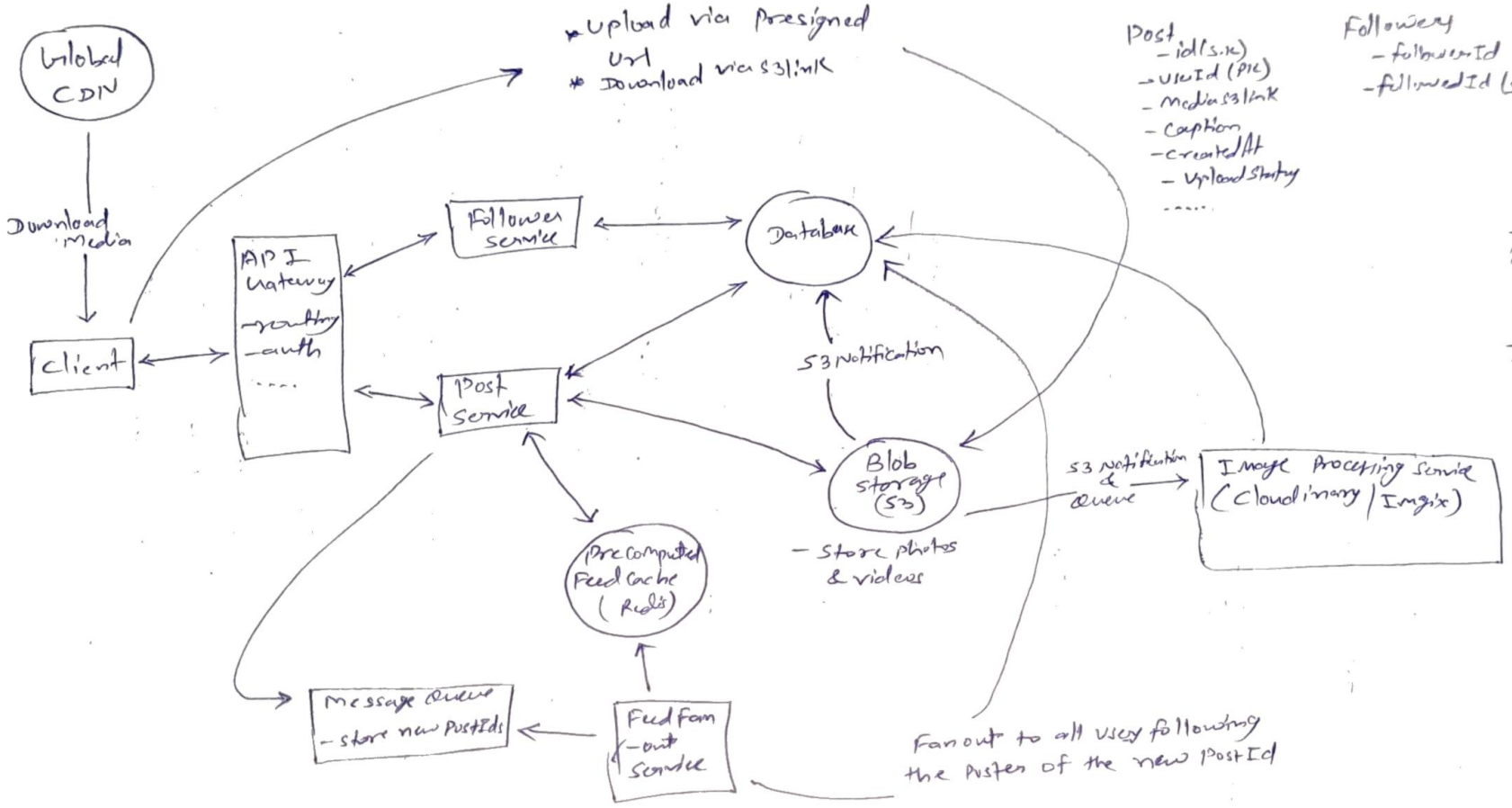
(I) Hybrid feed generation :- Precompute feeds for regular users, merge celebrity posts at read time. Balance read speed & write load.

(II) CDN for media Delivery :- Use a global CDN (eg cloudfront) to cache & serve media close to user. Supports fast, low-latency access worldwide

(III) chunked media uploads - Upload large files (eg 4GB) in parts using S3 multipart upload. Improves readability & allows parallel uploads.

(IV) Scalable Metadata storage - Store Metadata in DynamoDB with Proper Indexing for fast access & horizontal Scaling

(V) Smart storage tiers:-
↳ Move rarely accessed media/Metadata to cheaper storage (eg S3 → Glacier) to reduce cost.



HLD

(5) Auth - Scaling Microservices :- Microservices scale horizontally with traffic, using load balancers & cloud auto-scaling features.

Apache Kafka

(1) Introduction :-

* It is a distributed event streaming platform.
* Can be used as a "message queue" or for "Stream Processing".
* Provides high performance, durability, & scalability.
* Handles real-time, high-volume data reliably.

(2) Kafka in the World Cup use case :-

(a) **Producer** : Sends events (goals/bookings) to kafka

⇓

Think it as : A reporter who captures every match update & submits it to central kafka system

(b) **Consumer** : Reads events & updates website

⇓

Think it as : A reader who reads match updates from kafka & displays them on the live site.

(c) **scaling** : Needed when games/events increase massively.

* **what's the challange?**
  - initially, one game = manageable no. of events
  - But imagine a 1000 team tournament => with hundreds of matches running at once.
  - Now kafka is receiving a firehose of events - too much for a single broker or consumer to handle.
* **solution** : Kafka supports horizontal scaling by :
  - Adding more brokers
  - splitting data across multiple partitions
  - using consumer groups to parallelize processing

(d) Partitioning : Assign events by game → maintain order

* Partition messages by game id.
* All events for the same game go to the same partition → hence, they stay in correct order.

Think it as:- A private log for each match, so that nothing is jumbled.

(e) Consumer Groups ; Ensure each message is processed only once.

* what's the issue ?
  ↳ You want more consumers to handle the load, but :-
  - Don't want all consumers processing the same messages (unless it's pub/sub).
  - You want "each message to be processed once" (not duplicated).

* Kafka's solution :-
  - Use "Consumer groups".
  - Kafka assigns each partition to ~~"only one"~~ "only one consumer in a group".

(f) Topics : Logically separate streams like soccer / basketball.

* why topics ?
  ↳ Now your system grows : You're covering not just soccer, but also basketball, cricket etc
  ↳ You don't want soccer updates showing up on basketball dashboards.

* Kafka solution :-
  ↳ Use "topics" to separate data streams :
  - soccer events, basketball events etc.

↳ Producers send to appropriate topic, & Consumers subscribe only to the topics they care about.

* Think of a topic as:- A dedicated channel for each sport.

⇒ Problem First: Why do we need Consumer groups?

Imagine you're running a live world cup site where millions of events (goals, bookings, substitutions) are coming in at the same time.

The challenge:-
- A single Consumer can't keep up - it's too slow.
- So you decide to add more Consumers
- But if each Consumer reads all messages, every update is duplicated - that's not what we want.

* Kafka's answer: Consumer Groups :-

(a) What is a Consumer Group?

- A Consumer group is a set of Consumers working together to process messages from a Kafka topic.

- Kafka guarantees:
  * Each Partition is read by only one Consumer in the group.
  * So each message is processed only once, by one Consumer.
  * Consumers in the same group share the workload ie load-balanced.

(b) How it works?

Example:- You have a 'Soccer_events' topic with 3 partitions (P0, P1, P2).

- You Create a Consumer Group "Soccer_updates" with 3 Consumers (C1, C2, C3).

Kafka will do this:-

P0 → C1
P1 → C2
P2 → C3

} Now each consumer gets a unique partition = No duplication, full coverage

if one consumer fails (say C2), Kafka will rebalance:-

P0 → C1
P1 → C3 ← reassigned
P2 → C3

(c) what if we want multiple consumers to get the same message?

That's when you use multiple consumer groups.

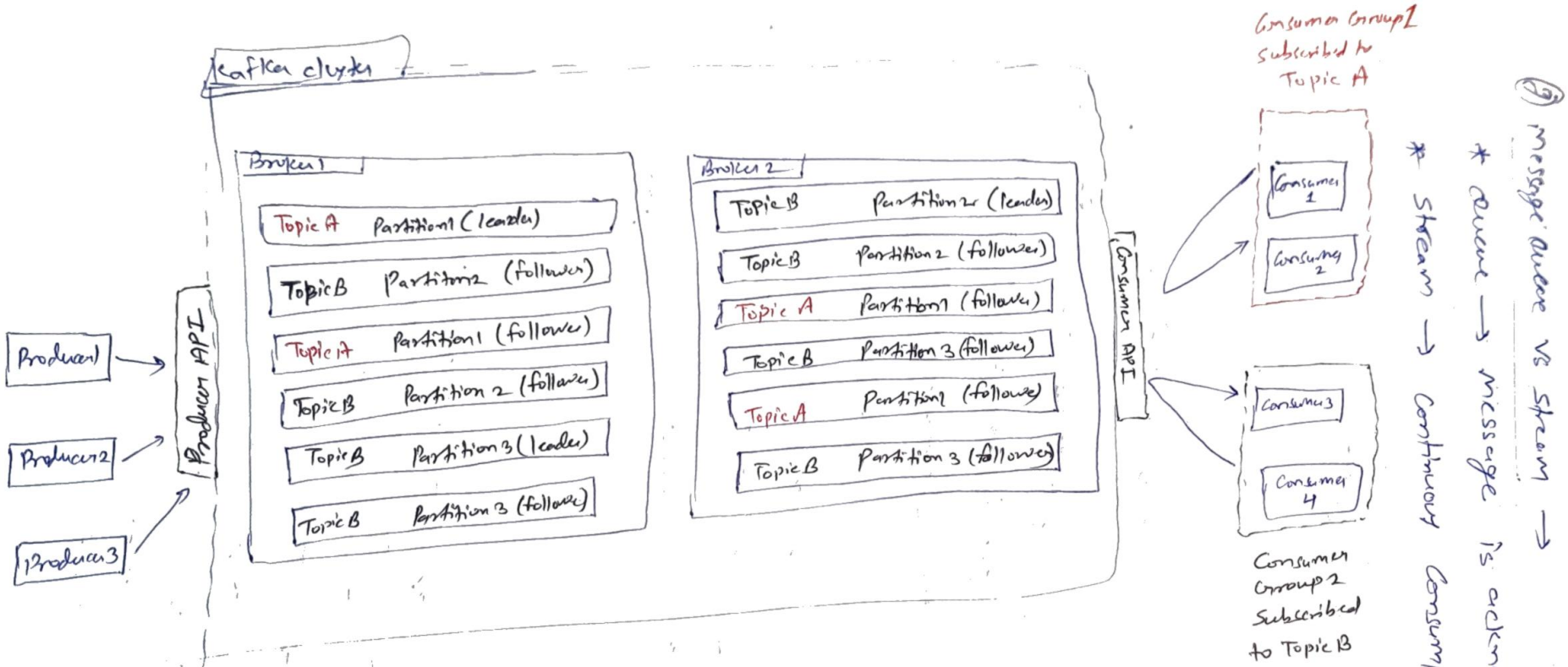
- Each group gets a copy of the topic's message.
- This gives you pub-sub behaviour.

Example:-

* analytics_group → consume messages for dashboards
* alerts_group → consume same message to trigger alerts.

(3) Kafka Core Taminology & Architecture :-

(a) cluster → Group of brokers (servers).
(b) Broker → Stores partitions & servers clients
(c) Partition → Append-only, ordered logs of message
(d) Topic → Logical grouping of partitions.
(e) Producer → Sends data to topics.
(f) Consumer → Reads from topics.



(6) Message queue vs stream →

* queue → message is acknowledged post processing.

* stream → continuous consumption with acknowledge.

(4) Kafka message —

Each message (also called a record) has:-

* Value : the actual data (eg "user clicked on add")
* Key : decide which partition it goes to.
* timestamp : when the event happened.
* headers : metadata (like HTTP headers)

→ without a key, Kafka picks a ~~random~~ partition randomly.

→ with a key, it hashes it and sends similar keys to the same partition (Preserving order).

(5) How Kafka decides where Data goes :-

(a) Kafka first "decides the partition" using the message key and a hashing algorithm.

(b) Then, it uses "cluster metadata" to know "which broker" holds that partition & sends the message there.

(6) Consumers and offsets :-

* Each message in a ~~message~~ Partition has an offset, like a unique message number.
* Consumers Pull (not Pushed!) message & "Commit offsets" after Processing — this is how Kafka knows where to resume if something crashes.

(7) Kafka uses Pull model (why?)

↳ Kafka Consumers | Poll | brokers for data, why?

* So they don't get overwhelmed (good for slow consumers).
* They can control their pace
* Allows efficient batching & better fault handling.

(8) When to use Kafka?

(a) Message Queue :-

* For asynchronous jobs: Like sending a video link to be processed later.
* If order matters.
* If Producer & Consumer need to scale independently.

(b) Stream :-

* For real-time processing: Like showing live ad clicks or FB live comments.
* When multiple consumers need to ~~read~~ read the same stream.

(9) Hot Partitions Problem (& Fixes) :-

Sometimes, one partition gets too much traffic (e.g. Popular ad like Nike). That's bad - it can crash that broker.

Fixes:-

(a) Don't use a key. → Random ℙ distribution (but lose ordering)

(b) Salt the key: add random suffix to it (e.g. "ad123-r1", "ad123-r2") → spreads load

(c) Compound key: Combine adId + region, or adId + userId group.

(d) Use backPressure :- make the Producer wait if the Partition is lagging.

(10) Durability in Kafka :-

(a) Replication Mechanism :-

↳ Each Kafka Partition is replicated across multiple brokers.

↳ one broker is the leader, other are followers.

↳ messages are written to leader & then replicated to followers.

⇒ This ensures that even if a broker fails, the data remains available.

(b) Producer acknowledgement :-

↳ `acks = all` : message is acknowledge only after all replicas receive it.

↳ Ensures "maximum durability".

(c) Replication Factor :-

↳ Defines how many replicas a partition has.

↳ Common value: 3 (1 leader + 2 followers)

↳ If one broker fails, a follower can take over.

(11) Consumer Fault Tolerance :-

(a) Offset management :-

↳ Consumers commit offsets after processing messages.

↳ on restart, consumer resumes from the "last committed offset", avoiding data loss or duplication.

(b) Rebalancing :-

↳ If a consumer in a group fails, Kafka reassigns partitions to remaining consumers.

(c) Offset Commit Timing:-

↳ Commit after critical Processing (e.g. storing HTML)

↳ Prevents data loss in the event of consumer failure.

↳ Minimize Consumer's responsibility to reduce retry overhead.

(12) Handling Failure & Retries:-

(a) Producer Retries:-

* Causes of failure: network issues, broker down etc.
* Kafka supports automatic producer retries:-

```
Const Producer = Kafka.Producer({
    retry: { retries: 5, initialRetryTime: 100},
    idempotent: true, //avoid duplication
});
```

(b) Consumer Retries:-

* Kafka does not support consumer retries natively.
* Common solution:-
  - Move failed messages to a "retry topic".
  - Process them with a seperate Consumer.
  - After max retries, move them to a "Dead letter Queue (DLQ)" for manual inspection

(13) Kafka vs SQS (for retry use case):-

In systems like Web Crawler, AWS SQS may be Preferred:-

* Built-in retry & DLQ support.
* Reduces complexity by eliminating need to manage retries manually.

(14) Performance Boosters :-

* Batching : Send many messages together.
* Compression : Use GZIP, Snappy etc. Smaller Payload = faster
* Partition wisely : Good Partition key = even load = faster Processing

(15) Retention Policies :-

Kafka does not delete messages immediately after consumption :-

- Messages are retained based on policy, not consumption status.
- This allows "replaying" messages if needed (e.g. for recovery or new consumers).

(a) Types of Retention :-

(i) Time-based retention (retention.ms) :-

→ Messages are kept for a fixed time duration (eg 7 days).
→ After this period, Kafka deletes the messages, regardless of consumption status.

(ii) Size-based Retention (retention.bytes) :-

→ Messages are retained until the total size of data in a topic exceeds a set threshold.
→ Oldest messages are deleted to stay within the size limit.

(b) Why Customize Retention policies?

(i) Storage Limitations:-

↳ Free up disk space by controlling how long/how much data is stored.

(ii) Business requirements:-

↳ Example: Audit logs may need to be retained for months or years for compliance.

(16) Scaling Kafka :-

Kafka is fast, but don't treat it like a database!

A single broker can handle:-

* ~1 TB data
* ~1 million messages/sec. (ballpark)

To scale:-

* Add more brokers.
* Add more partitions.
* ~~Do~~ choose partition key smartly to avoid hot spots

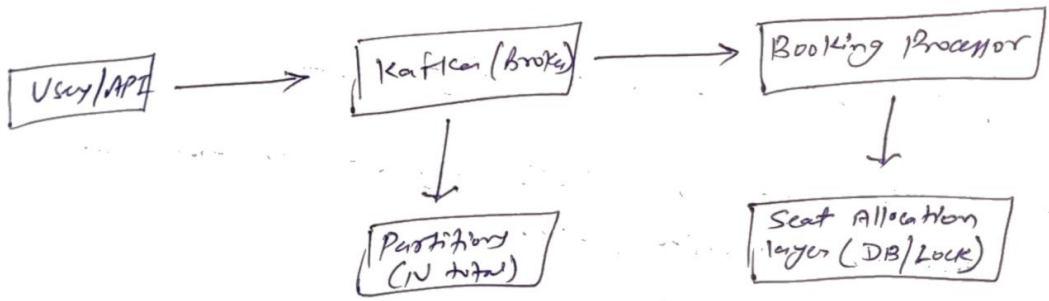
Example- Don't put entire video in Kafka. Put it in S3 and send the video URL via Kafka.

Scalable Ticket Booking System with First-Come, First-Served Semantics

Goal :-

Design a highly scalable ticket booking system that preserves first-come, first-served (FCFS) semantics while handling "millions of concurrent requests", ~~leveraging~~ leveraging Kafka for message ~~partitioning~~ ingestion.

Key Components :-



How it works?

Step (1) : Request Ingestion at API Layer :-

* API assigns each incoming request a 'unique', 'time-ordered token' (eg. ULID or timestamp).
* This ensures that even across partitions, requests can be ordered based on time.

Step (2) : Write to Kafka Topic with Multiple Partitions :-

* Partitioning can be done by event ID (eg. ConcertID) or user region.

* Multiple partitions enable horizontal scalability.
* Kafka ensures ordering within each partition.

Step (3): Booking Processor layer :-

* Multiple Consumers ~~Consume~~ Consume from different partitions in parallel.
* Each request carries its timestamp/token for global ordering.
* Consumers place incoming requests into a Priority queue based on token. (Can be Redis sorted set or in-memory structure).

Step (4) Seat Allocation (Critical section) :-

* Booking Processor picks requests in timestamp order from the Priority queue.
* For each request :-
  - Try atomic booking using DB constraint or distributed lock.

```
UPDATE seats
SET status = 'booked'
WHERE seat_id = ? AND status = 'available';
```

  - only one will succeed, others get a failure response.

Step (5) Response Handling :-

* Successful booking return Confirmation.
* Others receive appropriate error or retry suggestion

Guarantees & trade-offs :-

Aspect	Technique
(a) Ordering :-	Timestamp / ULID tokens + Priority Queue
(b) Scalability :-	Kafka multi-Partition + Parallel Consumers
(c) Atomicity :-	DB Constraint or distributed lock (e.g. "Redis Redlock")
(d) Performance :-	In-memory queue for FCFS, no global kafka ordering required
(e) Fault-tolerance :-	Kafka retry + idempotent Processors.

Summary

* Don't rely on single Kafka Partition - it limits throughput.
* Use token-based ordering, Partitioned ingestion, and atomic seat allocation.
* Kafka Provides durability & scalability; ordering logic is moved to Processors.