Great choice — a **News Feed System** (like Facebook or Twitter) is a **classic system design interview question** that tests your ability to design **highly scalable, real-time, and personalized systems**.

**🧠 Goal: Design a Scalable News Feed System**

**Example Use Case:**

* Show each user a **personalized feed** with posts from friends, pages, groups, etc.
* Support likes, comments, media, and live updates.
* Handle **millions of users** and **billions of posts**.

**🔧 Functional Requirements**

* Show most recent and relevant posts in feed
* Support infinite scrolling/pagination
* Posts can contain text, images, videos
* Users can like, comment, share
* Live feed updates (optional)
* Privacy settings respected (friends-only, public)

**🚫 Non-Functional Requirements**

* High availability and fault tolerance
* Low latency feed retrieval
* Scalability to millions of users
* Eventual consistency is acceptable

**🧱 Key Components**

**1. User Service**

* Auth, friends, followers, block lists, etc.

**2. Post Service**

* Handles creation, deletion, storage of posts

**3. Feed Generator**

* Builds feeds per user (push or pull model)

**4. Fan-out System**

* Push posts to followers’ feeds or pull on demand

**5. News Feed Store**

* Store generated feed per user

**6. Caching Layer**

* Store hot feeds in Redis/Memcached

**7. Media Service**

* Handles images/videos (uploads, CDN)

**⚖️ Feed Generation Strategies**

**🔁 Pull Model (on-demand):**

Rebuild feed when user opens the app

* Fetch top N posts from friends/pages
* Filter by visibility
* Rank using ML or heuristics

✅ Pros:

* Always fresh
* Less write-time overhead

❌ Cons:

* Slow read time (expensive DB joins)

**📤 Push Model (Fan-out-on-write):**

When a user posts, push to all followers' feeds

* Insert into follower feed DB
* Cache hot feeds in Redis

✅ Pros:

* Fast feed reads
* Easy to paginate

❌ Cons:

* Expensive writes
* Doesn't scale well with celebs (millions of followers)

**⚙️ Hybrid Model**

* Push to users with few followers
* Pull for users with many followers (e.g., >10k)

✅ Most real-world systems use this approach.

**🗃️ Data Models (simplified)**

**User**

json

CopyEdit

{

"user\_id": "u123",

"friends": ["u456", "u789"],

"following": ["u999"]

}

**Post**

json

CopyEdit

{

"post\_id": "p123",

"user\_id": "u456",

"content": "Hello world!",

"timestamp": "2025-04-14T10:00:00Z"

}

**Feed**

json

CopyEdit

{

"user\_id": "u123",

"posts": ["p123", "p127", "p101"],

"last\_updated": "2025-04-14T10:01:00Z"

}

**🧰 Tech Stack (Scalable Example)**

| **Component** | **Tech Example** |
| --- | --- |
| Feed Cache | Redis |
| Feed Store | Cassandra / DynamoDB |
| Post Store | MySQL / S3 (media) |
| Message Queue | Kafka |
| Fan-out Workers | Kafka Consumers, Spark |
| Search | Elasticsearch |
| Feed Ranking | ML Service / Heuristics |
| Media CDN | CloudFront / Akamai |
| API Gateway | NGINX / AWS API Gateway |

**🔄 Real-Time Feed Updates**

* Use **WebSockets / Server-Sent Events** for live updates
* Or poll /feed every few seconds

**📏 Scaling Strategy**

| **Layer** | **Strategy** |
| --- | --- |
| Post DB | Sharding by user\_id |
| Feed Cache | Partitioned Redis cluster |
| Media Storage | Use CDN, compress videos |
| Fan-out Workers | Scale with Kafka consumer groups |
| Feed Generation | Use Spark for batch processing |
| Load Balancing | NGINX, ELB, Round-robin |

**🎯 Interview Talking Points**

* “I’ll use a **hybrid fan-out** strategy to balance read/write load.”
* “Feed is stored in **Redis for fast retrieval**, and **Cassandra** for persistence.”
* “Celebrities with millions of followers will be handled via **pull model**.”
* “Feed ranking can be done using **feature-based heuristics or ML model**.”
* “I’ll keep the post and media storage **decoupled** for faster updates.”

**📊 Optional Enhancements**

* ML-based feed ranking (engagement prediction)
* Graph-based visibility (friend-of-friend feed)
* Notifications system for likes/comments
* Feature toggles (AB tests, trending topics)
* Deduplication & rate limiting (spam protection)

Would you like a **markdown summary**, **PDF diagram**, or a **low-level design with C++ classes** for Feed Generation Service?

**You said:**

Go in more details

**ChatGPT said:**

Updated saved memory

Awesome — let’s go **deep into the design of a News Feed System**, covering **every layer of the architecture**, **data flow**, **storage**, **scalability**, and even **possible low-level C++ service design** for the Feed Generator.

**🔎 News Feed System Design — Deep Dive (e.g., Facebook)**

**✅ 1. System Overview**

“Design a system where users can view a feed of posts from their friends, pages, or followed users, sorted by relevance and freshness.”

* **Read-heavy system** with personalization
* **Real-time updates** desired
* **Privacy-aware** and scalable to **billions of users/posts**

**🧱 2. High-Level Architecture**

sql

CopyEdit

+-------------+

| Clients |

+-------------+

|

+-------------+

| API Gateway |

+-------------+

|

+-----------+-----------+

| |

+----------------+ +------------------+

| Feed Generator |<--->| Post Service |

+----------------+ +------------------+

| |

+-----------+ +-------------+

| Feed Cache| | Media Store |

+-----------+ +-------------+

|

+---------------+

| Feed Database |

+---------------+

|

+-------------+

| Kafka Queue |

+-------------+

|

+------------------+

| Fan-out Workers |

+------------------+

**⚙️ 3. Component-Level Breakdown**

**📝 A. Post Service**

* Handles post creation (text/image/video)
* Saves metadata to a database (e.g. MySQL)
* Publishes event to Kafka:  
  → post\_created(user\_id, post\_id, timestamp)

**📣 B. Fan-out Workers**

* Consumes post\_created events from Kafka
* For each follower of the posting user:
  + Push post\_id to their feed in **Redis/Cassandra**
  + Or mark for **pull-based fetching** if the user is a celebrity
* Stored as feed:{user\_id} → list of post IDs (with TTL)

**📚 C. Feed Generator Service**

* Fetches feed from:
  + Redis (hot cache)
  + Cassandra (persistent store)
  + Post DB (to enrich post data)
* Can apply ranking:
  + Recent > Relevant > Diverse sources
  + ML model or feature heuristics

**🧠 D. Ranking/Personalization Service**

Uses engagement history, user preferences, and post features to score posts. Can be offline batch or real-time microservice.

**Ranking formula (example):**

txt

CopyEdit

Score = RecencyWeight \* freshness

+ EngagementWeight \* user\_interest

+ SocialWeight \* interactions\_with\_friends

**🛢 E. Feed Store (Cassandra/DynamoDB)**

Schema:

plaintext

CopyEdit

Table: user\_feed

Primary Key: (user\_id, timestamp DESC)

Columns: post\_id, source\_id, visibility

Advantages:

* Time-based sorting
* Efficient pagination (page by timestamp)

**⚡ F. Cache (Redis)**

* Redis List: feed:{user\_id} → [post\_id1, post\_id2, ...]
* Store top 100–200 posts for fast retrieval
* Invalidate on new post from friend or when old post expires

**🖼️ G. Media Service**

* Handles image/video uploads (via S3/CDN)
* Async video encoding using:
  + EventBridge → Lambda workers
* CDN to distribute heavy content (CloudFront, Akamai)

**🔐 H. Privacy Filter**

Each post has visibility metadata:

* Public / Friends Only / Custom
* Applied at query time or in fan-out stage

**🔄 4. Data Flow**

**Post Creation:**

sql

CopyEdit

User → API → Post Service → MySQL

→ Kafka: post\_created

↓

Fan-out Workers

→ Update Redis Feed Cache

→ Update Feed DB (Cassandra)

**Feed Retrieval:**

pgsql

CopyEdit

User → Feed Generator →

→ Redis: get recent posts (fast path)

→ Cassandra: fallback

→ Post DB: enrich post data

→ ML Ranking Service: rank posts

→ Return top N posts to frontend

**🛠️ 5. C++ Class Design: Feed Generator Service**

cpp

CopyEdit

class FeedGenerator {

public:

FeedGenerator(Cache& redis, Database& cassandra, PostService& postService);

vector<Post> getFeed(const string& userId, int limit);

private:

Cache& redis;

Database& cassandra;

PostService& postService;

vector<string> fetchPostIdsFromCache(const string& userId, int limit);

vector<Post> enrichPostData(const vector<string>& postIds);

vector<Post> rankFeed(const vector<Post>& posts);

};

**⚖️ 6. Design Tradeoffs**

| **Approach** | **Pros** | **Cons** |
| --- | --- | --- |
| Push (fan-out) | Fast reads | Heavy writes for celeb users |
| Pull | Less storage | Slower reads |
| Hybrid | Balanced | More complex |
| Redis cache | Ultra-low latency | Risk of staleness |
| Cassandra store | Scalable, persistent | Higher read latency |

**📏 7. Scaling Strategies**

| **Layer** | **Strategy** |
| --- | --- |
| Post DB | Shard by user ID |
| Feed DB | Partition by user ID |
| Kafka | Topic partitioning per user |
| Redis | Cluster with hash slot partitioning |
| Ranking | Move to offline batch scoring |
| Feed API | Auto-scale behind LB |

**🧪 8. Additional Features**

* Infinite scroll pagination (cursor-based)
* Feed filtering (e.g. only friends, only videos)
* Blocking/muting users
* Real-time updates (WebSockets)
* Notifications (likes/comments)
* ML-based engagement prediction
* Shadow banning/spam filters

Would you like:

* 📄 Markdown version?
* 📊 Visual architecture diagram PDF?
* 🧱 C++ implementation of a post or fan-out service?

Let me know what you'd like to build next from this!

**You said:**

C++ implementation of a post or fan-out service

**ChatGPT said:**

Let’s build a **C++ implementation of the Fan-out Service**, which listens to a Kafka-like message queue for new posts and pushes them into followers’ feed stores (e.g., Redis + Cassandra).

We’ll focus on:

1. Receiving post events
2. Fetching followers
3. Writing feed entries to Redis and Cassandra

**🧱 Fan-out Service - C++ Implementation (Simplified)**

**👇 High-Level Dependencies**

We’ll mock the following components for simplicity:

* MessageQueue – Kafka-like interface
* FollowerService – retrieves list of followers
* RedisCache – cache layer
* FeedStore – persistent feed database (Cassandra-like)

**🔧 PostEvent Model**

cpp

CopyEdit

struct PostEvent {

std::string postId;

std::string authorId;

std::string content;

std::time\_t timestamp;

};

**🔧 Interfaces**

**MessageQueue**

cpp

CopyEdit

class MessageQueue {

public:

virtual PostEvent consume() = 0;

};

**FollowerService**

cpp

CopyEdit

class FollowerService {

public:

std::vector<std::string> getFollowers(const std::string& userId);

};

**RedisCache**

cpp

CopyEdit

class RedisCache {

public:

void pushToFeed(const std::string& userId, const std::string& postId);

};

**FeedStore**

cpp

CopyEdit

class FeedStore {

public:

void saveToFeed(const std::string& userId, const PostEvent& event);

};

**🚀 FanOutService**

cpp

CopyEdit

class FanOutService {

public:

FanOutService(MessageQueue& mq,

FollowerService& fs,

RedisCache& rc,

FeedStore& db)

: queue(mq), followerService(fs), cache(rc), store(db) {}

void run() {

while (true) {

PostEvent event = queue.consume();

fanOut(event);

}

}

private:

MessageQueue& queue;

FollowerService& followerService;

RedisCache& cache;

FeedStore& store;

void fanOut(const PostEvent& event) {

auto followers = followerService.getFollowers(event.authorId);

for (const auto& followerId : followers) {

// Write to cache

cache.pushToFeed(followerId, event.postId);

// Write to DB

store.saveToFeed(followerId, event);

}

std::cout << "Post " << event.postId << " fanned out to "

<< followers.size() << " followers.\n";

}

};

**🔄 Mock Implementations (for completeness)**

cpp

CopyEdit

class MockMessageQueue : public MessageQueue {

public:

PostEvent consume() override {

return {"post123", "user456", "Hello World!", std::time(nullptr)};

}

};

class MockFollowerService : public FollowerService {

public:

std::vector<std::string> getFollowers(const std::string& userId) override {

return {"user001", "user002", "user003"};

}

};

class MockRedisCache : public RedisCache {

public:

void pushToFeed(const std::string& userId, const std::string& postId) override {

std::cout << "[Redis] Added post " << postId << " to " << userId << "'s feed\n";

}

};

class MockFeedStore : public FeedStore {

public:

void saveToFeed(const std::string& userId, const PostEvent& event) override {

std::cout << "[DB] Saved post " << event.postId << " to " << userId << "'s feed\n";

}

};

**🧪 main() Entry Point**

cpp

CopyEdit

int main() {

MockMessageQueue mq;

MockFollowerService fs;

MockRedisCache rc;

MockFeedStore db;

FanOutService service(mq, fs, rc, db);

service.run();

return 0;

}

**✅ Output Example**

vbnet

CopyEdit

[Redis] Added post post123 to user001's feed

[DB] Saved post post123 to user001's feed

[Redis] Added post post123 to user002's feed

[DB] Saved post post123 to user002's feed

...

Post post123 fanned out to 3 followers.

**📌 Notes for Production Readiness**

* Add batching for follower writes
* Use thread pool for parallelism
* Add retry and error logging
* Replace mocks with actual Kafka/Redis/Cassandra clients
* Use rate-limiting to avoid cache overload