Mastering System Design

Design a News Feed (aka Twitter)

What is a News Feed?

- A timeline of updates/posts from people you follow.
- Core in platforms like Twitter, Instagram, Facebook.

Core Functional Requirements

- Post a tweet/update.
- Follow/unfollow users.
- View home timeline.
- Timeline must show updates from followed users.
- Likes, replies, retweets
- Media uploads (images, videos)
- Interaction Flow (User Perspective):
    a. Alice follows Bob.
    b. Bob tweets “Hello World”.
    c. Alice opens app → sees Bob’s tweet instantly.

Non-Functional Requirements

- Timeline expectations:
  - Feeds load instantly
  - New posts appear in real-time
  - Feed is relevant & fresh
  - Posts are not missing or duplicated
- High Availability
- Low Latency
- Scalability

Constraints & Challenges

- Fan-In / Fan-Out Challenge
  - 1 user → 1 tweet → must reach 1M followers?
  - Should we push to all followers? Or pull on demand?

💡 Fan-out on write vs. Fan-out on read — this will shape our entire design.

Estimating Scale & Identifying Bottlenecks

- Daily Load Estimates:
  - 👫 500M total users | 200M DAU
  - 🧵 ~1B tweets/day | ~300M media uploads
  - 🔄 ~1B likes, 500M replies, 250M retweets
  - 📲 2B+ feed requests/day (10 opens/user)

📉 **Read > Write: ~80% traffic is read-heavy**

- Media Storage & Delivery
  - ~300M media uploads/day = 100–500TB/day
  - Must support:
    - Upload APIs
    - Object Storage (S3, GCS)
    - CDN-backed delivery
    - Tweet ↔ media linking

Identifying System Bottlenecks and Challenges

- Timeline = Fan-out at Scale
  - 1 user tweets → needs to be visible to 1M followers
  - Fan-out Models:
    - Fan-out on write: Pre-compute timelines (fast reads, heavy writes)
    - Fan-out on read: Compose timeline at read time (slower reads, lighter writes)

⚠️ Hot users (celebrities) can trigger write storms

- Read & Write Amplification
  - 🔁 Write-Intensive Actions
    - Posting a tweet: May fan out to 100s or millions of timelines
    - Retweets, replies, likes: Update multiple engagement counters and visibility
    - Media uploads: Chunked writes, storage, and metadata association
- 👀 Read-Heavy Patterns
    - Opening the timeline: Aggregates and sorts tweets from 100s of followed users
    - Viewing a tweet: Triggers fetches for media, replies, likes, retweets
    - Scrolling: Causes pagination, cache lookups, and lazy loads

One user action can trigger multiple backend reads/writes across services.

Core Services Breakdown

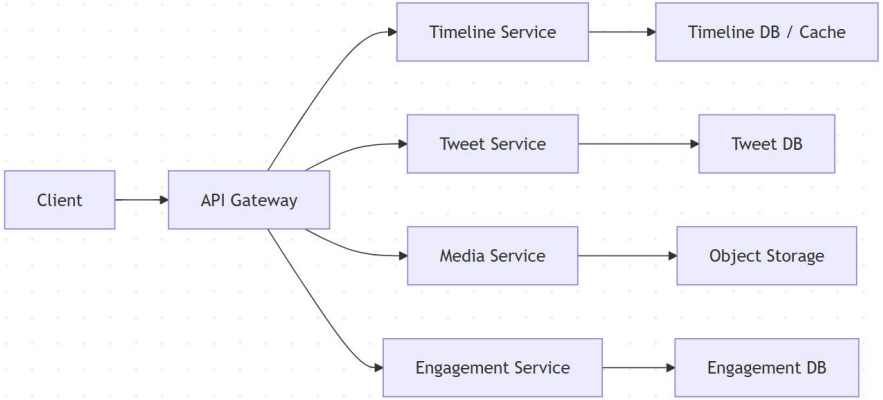
- **User Service** – Profile, follow/unfollow
- **Tweet Service** – Create, retrieve tweets
- **Timeline Service** – Compose timelines for users
- **Engagement Service** – Likes, replies, retweets
- **Media Service** – Upload, store, fetch media
- **Notification Service** – Fan-out alerts, activity
- **Fanout Worker** – Background timeline propagation

High-Level Architecture (Zoomed Out)

- User Flow:

Mobile/Web App → API Gateway → Microservices → DBs, Queues, Storage

- 🧠 Use API Gateway for:
  - Request routing
  - Auth, rate limiting
  - Aggregating service calls



API Design (High level) - Sample API Contracts:

- POST /tweet
  - → create new tweet (text + media references)
- GET /timeline
  - → returns latest tweets for a user
- POST /follow /unfollow
  - → modify following relationships
- POST /like /retweet /reply
  - → register engagement events
- POST /media/upload
  - → upload media, returns CDN URL

Note: All APIs are stateless; session/auth handled via token headers.

Timeline Generation Strategy

- Fan-out on Write
  - → Pre-compute timelines when a user tweets
  - ✔️ Fast reads, ⚠️ heavy writes for hot users
- Fan-out on Read
  - → Compose timeline on demand from followed users
  - ✔️ Lighter writes, ⚠️ slower reads
- 🌀 Hybrid Model = best of both:
  - Fan-out to regular users
  - Read-time fetch for hot users

Sync vs. Async Communication

- 🟢 Synchronous (API Calls):
  - Fetch user timeline
  - Get tweet details
  - Submit engagement
- 🟡 Asynchronous (Event Queues):
  - New tweet → enqueue fan-out jobs
  - Media uploads → process & link
  - Engagement → push notification trigger

📌 Improves throughput, decouples latency-sensitive paths

Strategic Tech & Infra Decisions

- 🧠 Guiding Principles:
  - Optimize for read-heavy load, real-time delivery, and massive fan-out
  - Prioritize latency, scalability, and operational simplicity
- 🧰 Key Tech Choices
  - Storage & DBs
    - Tweets, users, timelines → Scalable NoSQL (Cassandra, DynamoDB)
    - Engagements → Relational or Key-Value Store (PostgreSQL, Redis)
    - Media → Object Storage (Amazon S3, GCS) + CDN (Cloudflare, Akamai)
  - Messaging & Async Jobs
    - Kafka / RabbitMQ for fan-out jobs, engagement events, media pipelines
  - Caching
    - Redis / Memcached for hot timelines, tweets, and user sessions
  - Compute & Infra
    - Kubernetes / ECS for autoscaling microservices
    - API Gateway / Envoy for routing, rate-limiting, auth
  - Monitoring & Resilience
    - Prometheus + Grafana for metrics, alerts
    - Circuit breakers, retries, queues for graceful degradation

📌 Every choice is made to handle massive volume, reduce latency, and isolate blast radius of failures.

The Final Design - News Feed (aka Twitter)