Let's take a **deep dive into designing a scalable notification system**, which is a **frequently asked topic** in system design interviews — especially in the context of social media platforms, e-commerce, or any user-driven applications like Twitter, Instagram, Uber, or Amazon.

**🚨 High-Level Use Case:**

"Design a scalable system to send real-time and batched notifications to users based on events like messages, likes, comments, follows, delivery updates, etc."

**✅ Functional Requirements**

* Support **real-time notifications** (e.g., new message).
* Support **batch/digest notifications** (e.g., daily summary).
* **Multichannel delivery**: Push, email, SMS, in-app.
* **Priority handling**: Critical (message) vs non-critical (like).
* **Read/Seen status tracking**.
* Deduplication: Avoid duplicate notifications.
* Rate limiting (avoid spamming).

**🔒 Non-Functional Requirements**

* **Scalability**: 100M+ users, billions of notifications daily.
* **High availability** and **low latency**.
* **Fault tolerance** and **message retries**.
* **Extensibility**: Add new notification types easily.

**🧱 High-Level Architecture Diagram**

sql

CopyEdit

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

| Producer | --> | Kafka / PubSub | --> | Notification |

| (Tweet, | | Topic: NOTIFY\_EVT | | Worker Queue |

| Order, etc.) +------------------+ +------------------+

| | |

+---------+ | +--------+

| | |

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

| Push Engine | | Emailer | | SMS Engine |

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

| | |

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

| Device Tokens | | Email Queue | | Twilio/Nexmo|

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

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

| Notification DB |

| (seen/unseen, meta) |

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

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

| User Preference Service |

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

**📩 Notification Lifecycle (Flow)**

**1. Trigger Event**

A service (like TweetService, OrderService) emits an event to Kafka:

json

CopyEdit

{

"event\_type": "NEW\_LIKE",

"actor\_id": 123,

"target\_user\_id": 456,

"target\_entity": "tweet:789",

"timestamp": "2025-04-14T09:00Z"

}

**2. Kafka → Notification Workers**

Workers consume from the Kafka topic notify\_events, apply logic:

* Deduplication check (e.g., already notified).
* Priority check: Real-time (send now) vs Batched (store for later).
* User preference check (push/email/SMS).
* Fan-out (e.g., multiple followers).

**3. User Preferences & Deduplication**

* Each user has notification **preferences**:

json

CopyEdit

{

"user\_id": 456,

"preferences": {

"likes": "push",

"mentions": "email+push",

"new\_follower": "digest"

}

}

* Deduplication logic:
  + Use Redis or Bloom filters to prevent double-notifying.
  + TTL-based notification keys.

**4. Notification Rendering**

* Templates per channel: push/email/SMS.
* Can use a **Template Engine** like:
  + Jinja2 (Python)
  + Handlebars.js
  + Mustache

**5. Sending Notification**

* Notification is dispatched via different **channels**:
  + **Push**: Firebase Cloud Messaging (FCM), Apple Push Notification Service (APNs)
  + **Email**: SES, Sendgrid, Mailgun
  + **SMS**: Twilio, Nexmo
* Retry with **Exponential Backoff** on failure.

**🧠 Core Concepts & Design Decisions**

**✅ Fan-Out Architecture**

**Fan-out on Write**

* Store notifications immediately for each target user.
* Good for **fast read** (notifications page).
* Costly if many followers (e.g., celebrity tweet).

**Fan-out on Read**

* Store a single event and compute notifications during read.
* Cost-effective, but **higher latency** during read.

🎯 **Hybrid Approach**:

* Fan-out on write for most users.
* Fan-out on read for celebrity/viral content.

**🗃️ Notification Data Model**

sql

CopyEdit

Table: notifications

- notification\_id UUID

- user\_id BIGINT

- type TEXT

- entity\_id TEXT

- seen BOOLEAN

- created\_at TIMESTAMP

**⚙️ Important Optimizations**

| **Component** | **Optimization Strategy** |
| --- | --- |
| Kafka | Partition by user\_id for parallelism |
| Deduplication | Redis/Bloom filters (key: user\_id:event\_id) with TTL |
| Worker Queue | Use message queue (SQS, RabbitMQ) with retry + dead-letter queues |
| Rendering | Cache rendered templates per notification batch |
| Delivery | Use CDN/local edge servers for push endpoints |
| Batched Digest | Use scheduled jobs (cron, Airflow) for email digests |

**💡 Algorithms/Tools Used**

| **Feature** | **Tool/Algorithm** |
| --- | --- |
| Event Queue | Kafka, Pub/Sub |
| Deduplication | Redis, Bloom Filter |
| Fan-out | Kafka consumers, Lambda |
| Read Status | Redis cache + SQL fallback |
| Notification storage | Cassandra/PostgreSQL |
| Retry Handling | Exponential Backoff |
| Batch Digest | Cron + Batched SQL Reads |
| Multichannel | FCM, SES, Twilio |

**🧪 Interview Questions You Might Be Asked**

1. **How would you ensure real-time delivery for 1 million users?**
2. **How would you store unseen notifications for fast access?**
3. **What if a user has 10M followers and tweets?**
4. **How do you prevent duplicate notifications?**
5. **How do you handle user preferences for different channels?**
6. **How would you scale this across data centers?**
7. **What happens if a notification fails to send?**
8. **Design a dashboard to show all your past notifications with filters.**

**🎓 Beginner-Friendly Story Analogy**

Imagine you're building a mailroom for Hogwarts students. Every time someone gets a letter (event), the owl post (Kafka) delivers it. A sorting hat (worker) decides whether to send the message as a real-time owl (push), a magical scroll (email), or a floating note in the common room (in-app). Some messages are urgent (like Quidditch scores), some can wait (like dinner menu). There are rules for what students want (some hate owls at night), and you don’t want to send 100 letters for 1 party invite. So, you group messages, log who got them, retry failed deliveries, and make sure students aren’t overwhelmed — all while ensuring Hedwig doesn't burn out.

**✅ 1. Core Requirements (Interview Focus)**

**Functional:**

* Trigger notifications from events (likes, comments, mentions, etc.)
* Deliver notifications in real-time (push)
* Support batch/digest mode (e.g., daily summary)
* Allow read/unread tracking
* Mobile and web push support

**Non-Functional:**

* High throughput (millions of events per day)
* Low latency (real-time delivery)
* Fault-tolerant and horizontally scalable
* Rate limiting to prevent abuse
* Personalization & prioritization

**🧱 2. High-Level Architecture**

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

| Event Producers | ---> | Kafka Event Bus | ---> | Notification Worker |

| (Tweet, Like, etc.) | +-------------------+ +----------+----------+

|

v

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

| Notification Store | <-- | Notification Generator |

+--------------------+ | (Builds payload, filters) |

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

|

+-------------------+ |

| Delivery Service | <----------------+

| (Push/Websocket) |

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

|

v

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

| Mobile/Web Push Layer |

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

**📤 3. Push Delivery Layer (Deep Dive)**

**✅ Components**

* **Delivery Service**: Routes the message to correct channel (WebSocket, mobile push, email, etc.)
* **Push Adapter**: Platform-specific (APNs for iOS, FCM for Android, WebPush)
* **WebSocket Service**: Persistent real-time channel

**✅ Real-time Delivery Flow:**

1. New event published to Kafka
2. Notification Worker consumes and processes it
3. Delivery Service checks user connection status
4. If online:
   * Send via WebSocket
5. If offline:
   * Store for push (APNs, FCM) or email

**✅ Optimizations:**

* Retry queues (exponential backoff)
* Adaptive delivery: downgrade push to email for inactive users
* Use Redis for active connection map: user\_id -> socket\_id

**💾 4. Storage & DB Modeling (Interview Focus)**

**✅ Notification Table Schema (SQL/NoSQL)**

Table: notifications

- notification\_id (UUID)

- user\_id

- event\_type (LIKE, COMMENT, MENTION)

- actor\_id (who triggered it)

- content\_snippet

- target\_url

- created\_at TIMESTAMP

- is\_read BOOLEAN

- priority INT

* **Partitioning**: By user\_id for fast fetch
* **Indexes**: On user\_id + is\_read, created\_at

**✅ Redis as a Cache Layer:**

* Recent notifications stored in Redis Sorted Set

ZADD user:<user\_id>:notifications <timestamp> <notification\_id>

* Improves UI performance (scroll, infinite load)

**✅ Fan-out Strategies:**

* **Fan-out on write**: Immediately generate notifications for each follower (common in Twitter-style apps)
* **Fan-out on read**: Generate notifications when user logs in or checks app (scales better for inactive users)

**📬 5. Digest Notification Logic**

**✅ Use Case:**

* User receives many notifications in short time
* Batch them as a digest (e.g., daily email)

**✅ Approach:**

* Store all notifications in DB
* Scheduled batch job aggregates recent unread notifications
* Group by type (e.g., "5 people liked your post")
* Send via email or inbox view

**✅ Digest Job Flow:**

1. Run every X hours via a scheduler (e.g., cron/K8s job)
2. Query unread notifications by user, group by type
3. Format message (HTML for email, JSON for inbox)
4. Send to user via email service or push

**📊 6. Interview Optimization Topics**

|  |  |
| --- | --- |
| **Topic** | **Interview Insight** |
| DB Design | Partitioning by user\_id, indexing on created\_at/is\_read |
| WebSocket Scaling | Sticky sessions, user -> socket map in Redis |
| Rate Limiting | Use token bucket in Redis per user |
| Prioritization | Score-based sorting in Redis sorted set |
| Delivery Channel Choice | Real-time (WebSocket) → Fallback (Push) → Fallback (Email) |
| Caching Strategy | Store recent 100 notifications per user in Redis |
| Fan-out Strategy | Fan-out-on-write for normal users, fan-out-on-read for high scale |