Insyd Notification System — System Design Document (POC → 1M+ DAUs)

Version: 1.0
Date: 22 Aug 2025
Author: Vaisshnavi

1. Executive Summary

This document details the design of the Insyd Notification System, starting as a proof-of-concept (POC) and scaling to 1M+ DAUs. The system provides realtime, resilient, and secure in-app notifications (likes, comments, follows, posts). The design ensures correctness (no duplicates, ordered per user), low latency (<500 ms p50 at POC, <2 s p95 at scale), and operational excellence (monitoring, alerting, graceful degradation).

2. Goals & Non-Goals

Goals

- Deliver user notifications with <500 ms p50 latency (POC) and <2 s p95 (1M+ DAUs).
- APIs for fetching, marking read, clearing, and stats.
- Realtime UX via polling (POC) → WebSockets (scale).
- Idempotency, per user ordering, deduplication.
- Clear scaling roadmap: monolith → microservices with Kafka/Redis.

Non-Goals

- Multi-channel delivery (email/push) in POC.
- ML-based ranking (future).
- Multi-region active-active (Phase 4).

3. Scope & Requirements

Functional

- Ingest events: like, comment, follow, post.
- Transform into user-visible notifications.
- Fetch, mark read, clear; manage preferences.
- Admin APIs: retry DLQ, event listing.
- Health endpoints and metrics.

Non-Functional

- Latency: p50 <500 ms (POC), <2 s (scale).
- Availability: ≥99.5% (POC), ≥99.9% (Prod).
- Durability: no event loss, dedupe on handler.
- Scalability: horizontal for APIs, processors, DB shards, WS gateways.
- Security: JWT auth, input validation, rate limiting.
- Observability: logs, metrics, traces, SLO dashboards.

Assumptions

- Auth handled by Insyd Identity Provider.
- Read model in MongoDB; analytics later via warehouse.
- Single region (ap-south-1 initially).

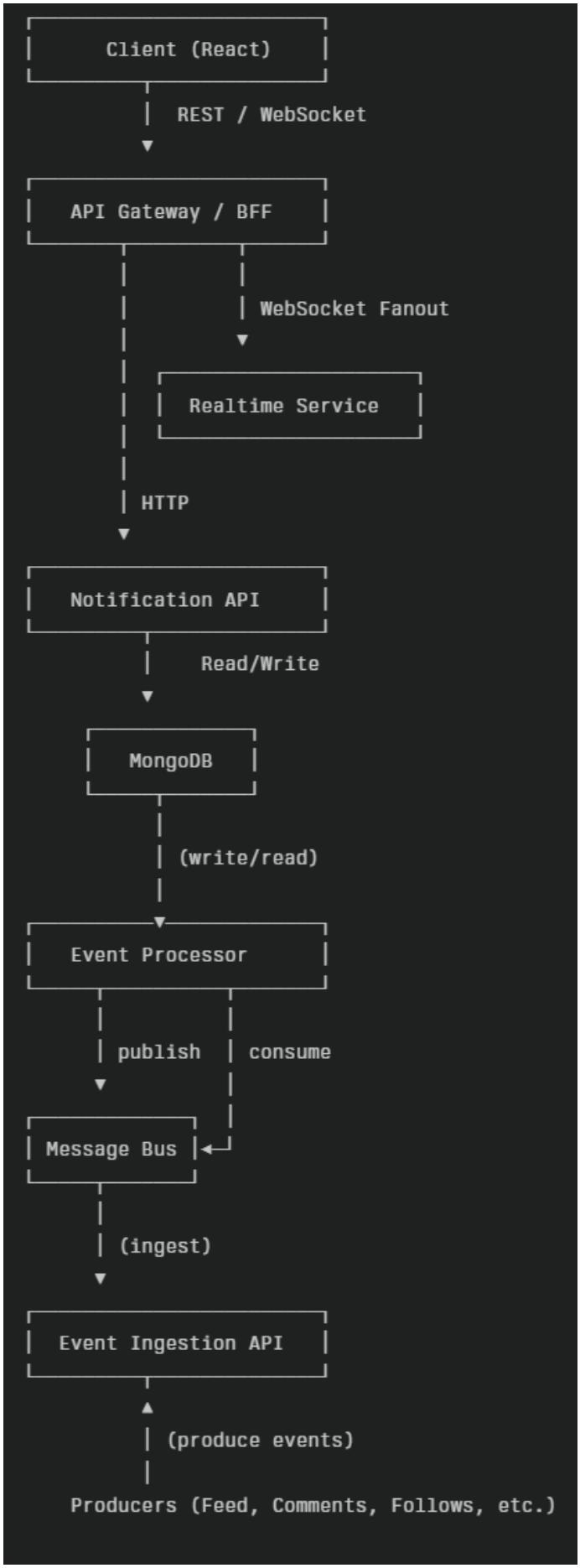
4. High-Level Architecture

POC Topology (100 DAUs)

- Single Node.js API + in-memory queue.
- MongoDB replica set.
- Client polling every 5s.

Scaled Topology (1M+ DAUs)

- Microservices: Event Ingestion, Processor, Notification API, Realtime Gateway.
- Kafka/Redis Streams as durable bus.
- MongoDB sharded cluster.
- Redis for cache/pubsub/rate limits.
- WS fanout via stateless gateways.


Client (React)
REST / WebSocket
API Gateway / BFF
WebSocket Fanout
Realtime Service
HTTP
Notification API
Read/Write
MongoDB
(write/read)
Event Processor
publish
consume
Message Bus
(ingest)
Event Ingestion API
(produce events)
Producers (Feed, Comments, Follows, etc.)

5. Component Design

- **Frontend (React + TS):** Polling & WS clients, optimistic UI, accessibility.
- **Backend (Node.js):**
    - Event Ingestion API → validates & enqueues.
    - Processor → applies preferences, dedupe, writes to DB, emits fanout.
    - Notification API → read-optimized queries, counters.
    - Realtime Gateway → manages WS connections & subscriptions.
- **Data Storage:**
    - MongoDB: users, notifications, events.
    - Redis (prod): cache, counters, WS pub/sub.

6. Data Model (MongoDB)

Users

{

"_id":"user-1",

"preferences": {"like":true},

"counters": {"unread":3},

"createdAt": "Hyd"

}

Notifications

{

"_id":"A1234",

"userId":"user-2",

"type":"like",

"eventId":"evt-1",

"sourceUserId":"user-1",

"status":"unread",

"createdAt":"abc"

}

Indexes: { userId, createdAt }, { eventId } unique, optional TTL.

7. APIs (POC → Scale)

- GET /api/notifications/:userId?cursor&limit — fetch feed.
- PATCH /api/notifications/:id/read — mark read.
- DELETE /api/notifications/users/:id/clear — clear all.
- GET /api/notifications/users/:id/stats — unread stats.
- POST /api/events — ingest event.
- POST /api/events/:id/retry — admin retry.
- Auth: demo tokens (POC) → JWT (Prod).
- Contract: /api/v1/* with OpenAPI.

8. Core Workflows

Event → Notification

1. Producer posts event.
2. Ingestion validates & enqueues.
3. Processor consumes, applies prefs.
4. Writes notification (dedup by eventId).
5. Updates counters, pushes WS update.
6. Client receives via WS/polling.

Read / Clear: update status, decrement counters, optional archive.

Ordering: per user ordering by createdAt.

Idempotency: enforced by eventId unique index.

9. Realtime Delivery

- **POC:** Polling every 5s.
- **Scale:** WebSockets with per-user channels, pub/sub fanout, backpressure control.

10. Capacity Planning

- 1M DAUs → ~20M events/day (~231/s avg, 2.3k/s peak).
- Notification size ~400 B → 8 GB/day, ~720 GB for 90 days.
- Kafka: 3 brokers, RF=3, 12 partitions.

- MongoDB: 3-shard cluster, RF=3.
- Autoscaling processors via k8s HPA.

11. Resilience & Security

- **Resilience:** retries, DLQ, exponential backoff, circuit breakers, graceful shutdown.
- **Security:** JWT auth, schema validation, CORS restrictions, rate limiting, KMS secrets, GDPR data handling.

12. Observability & SLOs

- Metrics: latency, throughput, queue lag, WS connections.
- Logs: JSON structured with correlation IDs.
- Traces: OpenTelemetry across services.
- **SLOs:**
    - Availability: 99.9% Notification API.
    - Latency: 95% notifications visible <2 s.
    - Durability: 0 lost events.

13. Deployment & DevOps

- Envs: dev → staging → prod.
- CI/CD: lint, test, build, deploy, canary.
- Infra: Terraform, k8s, MongoDB Atlas, MSK.
- Rollout: blue/green, feature flags for WS.

14. Testing Strategy

- Unit: transformers, validators.
- Contract: OpenAPI + Pact.
- Integration: API ↔ DB, Processor ↔ Bus.
- E2E: seed → like → notify → read.
- Load: Artillery/K6 (steady, spike, soak).
- Chaos: broker/node failures (Phase 3).

15. Roadmap

1. **POC (Now):** single API, Mongo, polling.
2. **Phase 2:** WS, Redis cache, split processors.
3. **Phase 3:** Kafka, DLQ, aggregation, email/push.
4. **Phase 4:** multi-region, analytics, ML ranking.

16. Risks & Mitigations

- Hot users → hashed shard key + cache.
- WS storms → rate limit, batch updates.
- Duplicates → eventId unique index.
- Schema drift → strict validation + migrations.
- Cost spikes → autoscaling caps, tiered storage.

17. Operational Playbooks

- Elevated 5xx → check DB, queue lag, rollback, fallback WS → polling.
- Kafka lag → scale processors, check consumer groups.
- Replica lag → redirect reads to primaries, throttle writes.

18. Future Enhancements

- Rich templates, deep links, action buttons.
- Multi-channel delivery (email/push).
- ML ranking & bundling.
- Analytics pipelines & compliance exports.

19. Appendices

- **OpenAPI Sketch (YAML):** key endpoints.
- **Mermaid Sequence Diagram:** Event → Bus → Processor → DB → WS → Client.
- **Artillery Load Profile:** steady + spike.
- **Mongo Indexes:** queries & idempotency.

20. Conclusion: The Insyd Notification System is designed to be lightweight at launch yet robust enough to scale to millions of users. By combining modular architecture, event-driven pipelines, and strong observability, it ensures reliable delivery and seamless user engagement across channels.