SYSTEM DESIGN CAPSTONE | 2026

Designing a Planet-Scale Notification & Push System

Multi-channel delivery at 100M DAU, push, SMS, email & in-app, engineered for a ~350k writes/sec burst budget.

Yahya Ahmed
yahyaahmedg@gmail.com
Solution Architect | Solo submission

Scale target: 10M+ DAU & 20k+ QPS (both exceeded)

THE PROBLEM

What we are building & why it is hard

A single, shared platform that every internal service calls to reach a user on the right channel, reliably, in the right order, respecting the user's consent, whether it is one OTP or a 100-million-user campaign.

200M	100M	~800M	~350k
registered users	daily active users	notifications / day	peak writes / sec

In scope

- Transactional (OTP, order, payment) + bulk marketing / campaigns
- Channels: Push (APNs/FCM), SMS, Email, In-app / WebSocket
- User consent, quiet hours, dedup, rate-limit, scheduling
- Delivery tracking, retries, and per-channel analytics

Out of scope

- Authoring UI / campaign builder front-end
- The ML model that picks audience segments
- Billing & reconciliation with SMS/email vendors
- In-app inbox rendering on the client device

REQUIREMENTS

Functional Requirements

1

Send

Producers submit a notification to one user or a segment via a single API, with a priority and an idempotency key.

2

Multi-channel

Route to Push, SMS, Email or In-app; fall back to a secondary channel if the primary fails.

3

Consent & preferences

Honour per-channel / per-category opt-out, quiet hours, locale and timezone before anything is sent.

4

Templating

Render a versioned, localized template with per-user variables; dedup identical sends.

5

Scheduling

Send now, send later, or recurring; campaigns can be paused and resumed.

6

Delivery tracking

Expose sent / delivered / opened / failed status and retry failures with backoff.

Inputs: notification requests (userId(s), templateId, channel, priority, payload) + provider delivery webhooks. **Outputs:** rendered messages delivered to devices + a queryable delivery-status stream + analytics events.

REQUIREMENTS

Non-Functional Requirements

Latency

OTP / transactional p99 < 2s end-to-end. Marketing may take minutes, a deliberate two-tier SLA.

Availability

99.99% for the transactional path (~52 min/yr). Marketing path may degrade gracefully under load.

Durability

No lost transactional notification. Persist to Kafka before ACK; at-least-once + idempotent dedup.

Scalability

Linear horizontal scale to ~350k writes/sec at burst; absorb 100M-user campaign spikes.

Consistency

Eventual for delivery status; strong enough dedup so a user never gets the same OTP twice.

Security & Compliance

PII encrypted at rest & in transit; provider creds in a vault; GDPR opt-out & data retention.

Cost is a first-class NFR: SMS ~100x the cost of push, so the design always prefers the cheapest channel that meets the SLA.

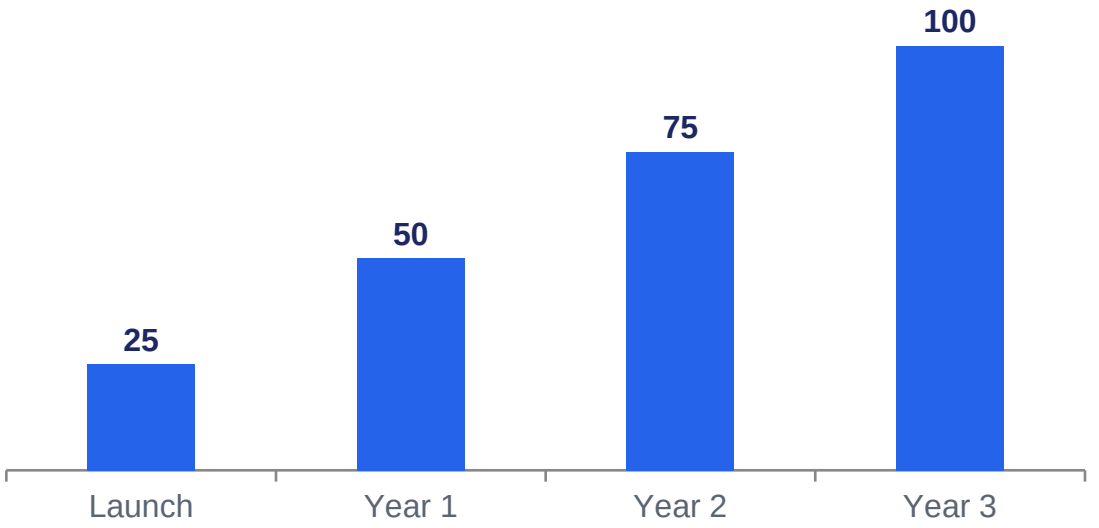
ESTIMATION

Assumptions & User-Growth Model

Assumptions

- Start: 50M registered / 25M DAU. Grow to 200M / 100M over 3 years.
- Each active user gets ~8 notifications/day: ~5 transactional + ~3 marketing.
- Read:write is write-dominated, every notification is a write; reads (prefs, tokens) are cache-served.
- Peak factor 5x over average (evening + campaign bursts).
- A campaign targets up to 100M users and must drain within ~10 minutes.
- Average message payload after templating ~1 KB.

Daily active users (millions)



4x growth in 3 years. We size storage & throughput for the Year-3 figure, then verify the launch config is not over-provisioned.

ESTIMATION

QPS Estimation (step-by-step)

1. **Daily volume**

   `100M DAU x 8 notifs/day = 800M notifications/day`

2. **Average fan-out QPS**

   `800M / 86,400 s = ~9,260 notifs/sec`

3. **Peak (5x)**

   `9,260 x 5 = ~46k notifs/sec sustained peak`

4. **Campaign burst**

   `100M users / 600 s = ~167k notifs/sec -> design for 200k/sec`

5. **Delivery receipts**

   `~1 webhook / delivered msg = up to ~150k/sec inbound`

6. **Preference reads**

   `1 lookup / notif, 95% + from cache = ~200k/sec (cheap)`

~200k
notifications/sec
design target (write path)

~350k
total writes/sec at burst
(fan-out + receipts)

10x
headroom vs the 20k QPS
capstone bar

ESTIMATION

Storage Estimation

Message-status records dominate

1 status record	~1 KB (ids, channel, status, timestamps)
Per day	800M x 1 KB = ~800 GB/day
90-day hot window	~72 TB -> x3 replication = ~216 TB
Older than 90 days	roll off to S3 cold storage (cheap, rarely read)
User preferences	200M x 2 KB = ~400 GB (fits in cache tier)
Device tokens	200M x 2 devices x 200 B = ~80 GB

~216 TB

replicated hot status store (90 days)

Tiered retention keeps the hot store bounded regardless of how long the product runs.

Design implications

- Status store must be write-optimized & horizontally sharded -> Cassandra.
- Payloads stored once as a blob (S3); records hold a reference, not the body.
- Small, hot metadata (prefs, tokens) lives behind Redis.

DATA MODEL

Entities & Relationships

User

The recipient. Owns devices, preferences and a locale/timezone.

Device

A push endpoint (token + platform). A user has many.

Preference

Per channel & category consent, plus quiet hours.

Notification Request

One producer intent, single user or a segment.

Template

Versioned, localized body with variables. Reused by many requests.

Message

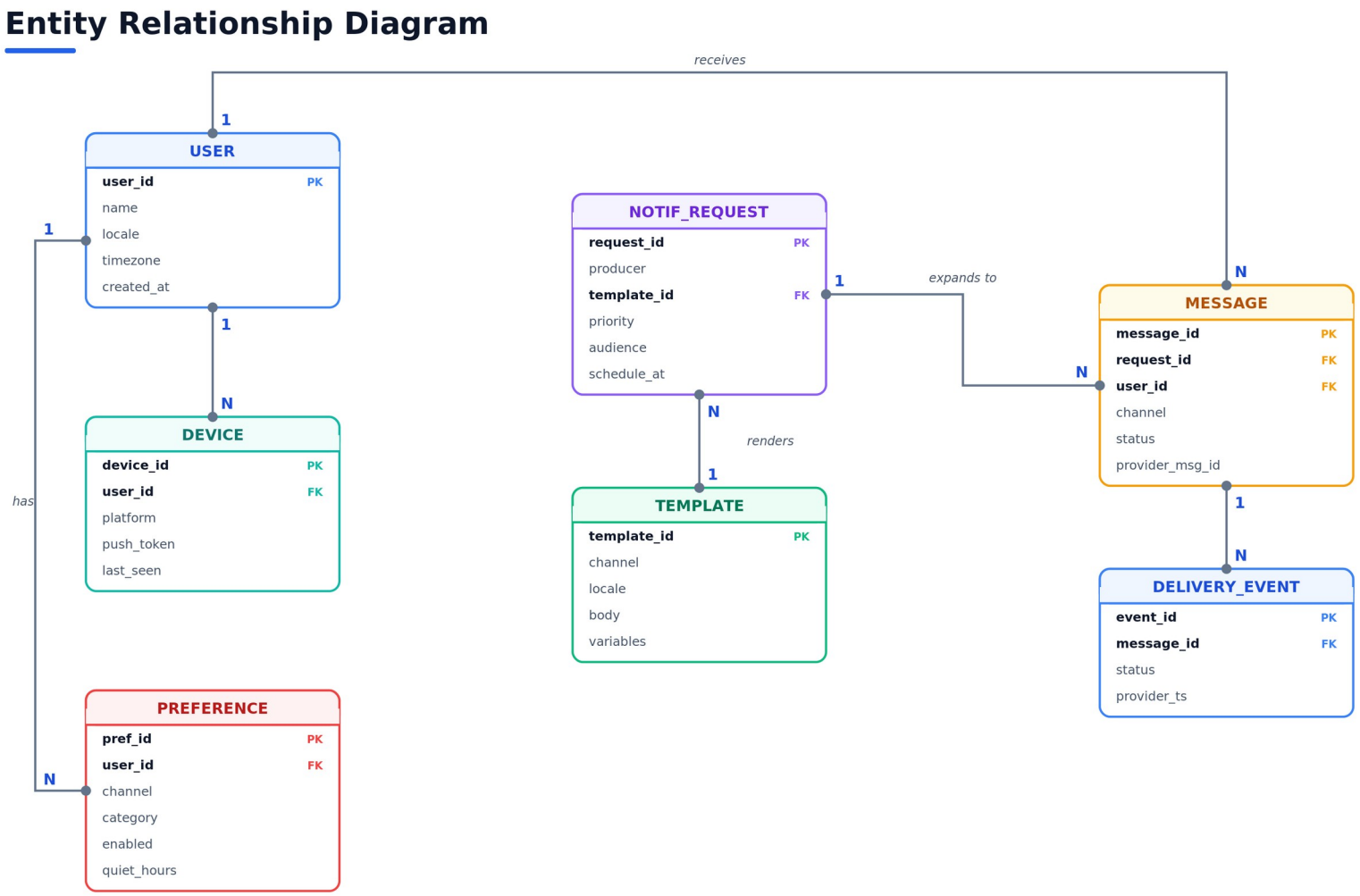
One request expands into many messages (one per user/channel).

Delivery Event

Provider callbacks: sent / delivered / opened / failed.

Key relationships: User 1, N Device, Preference, Message. Request 1, N Message. Template 1, N Request. Message 1, N Delivery Event.

Entity Relationship Diagram



INTERFACES

API Design

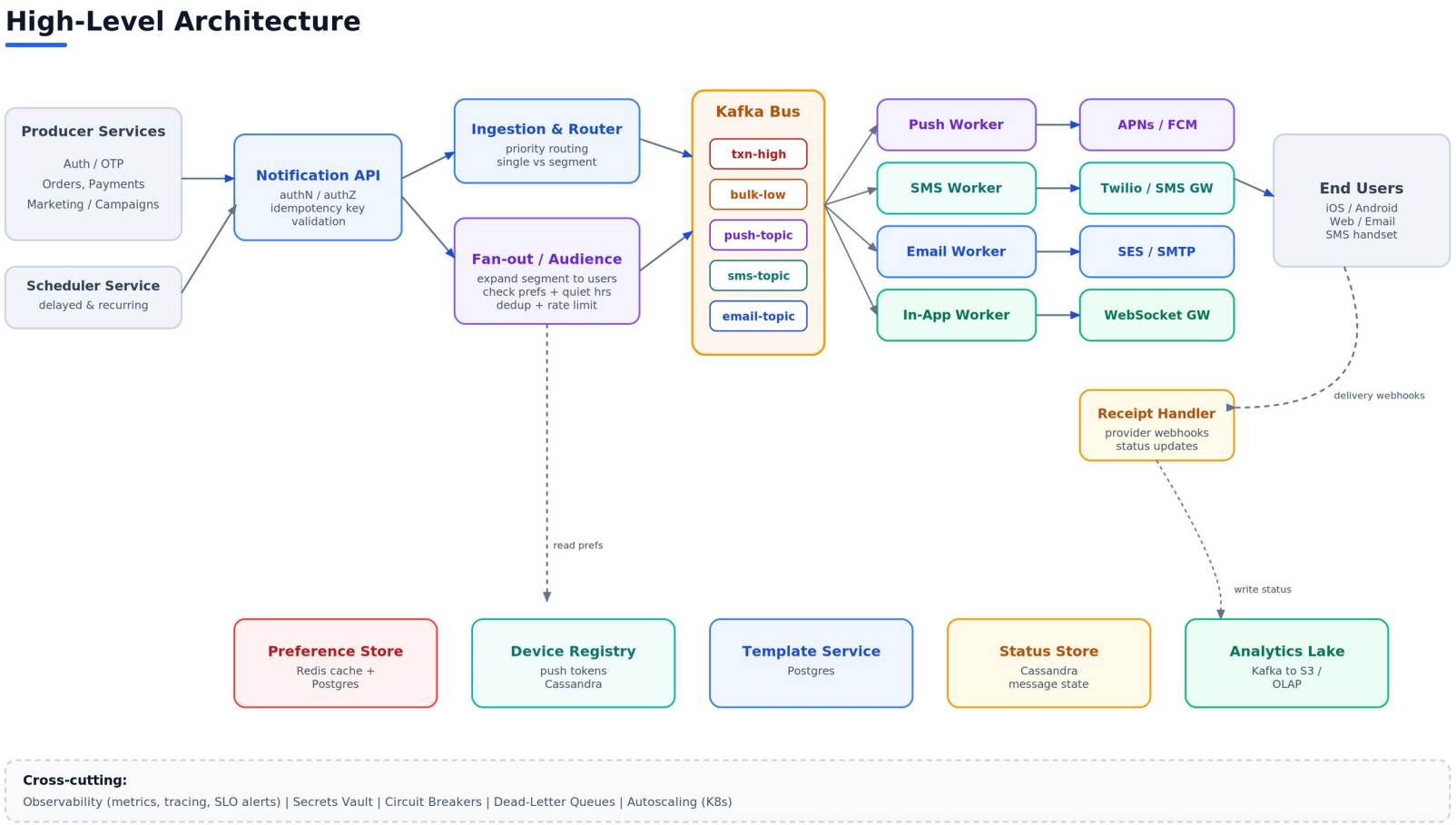
POST	/v1/notifications	Send to a single user. Idempotency-Key header. Returns 202 + trackingId.
POST	/v1/campaigns	Send to a segment. Async fan-out; returns 202 + campaignId.
GET	/v1/notifications/{id}	Delivery status of one message.
PUT	/v1/users/{id}/preferences	Update consent, quiet hours, channels.
POST	/v1/users/{id}/devices	Register / refresh a push token.
POST	/v1/webhooks/providers/{p}	Ingest provider delivery receipts.

Principles

- Versioned (/v1), additive changes only.
- Async by default: 202 + trackingId, never block the caller on fan-out.
- Idempotency-Key makes retries safe (exactly-once effect).
- REST for producers; gRPC internally for low-latency service calls.
- Priority is a first-class field, not a separate endpoint.

ARCHITECTURE

High-Level Architecture

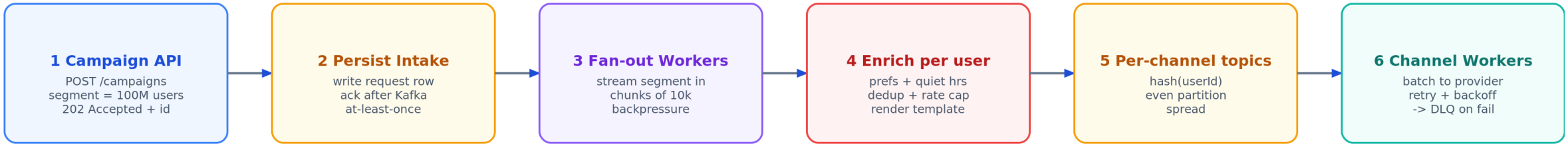


Producers -> API -> Kafka (priority topics) -> fan-out -> per-channel workers -> providers -> devices. Receipts flow back to the status store; everything is autoscaled & observable.

BONUS: WRITE-INTENSIVE DESIGN

The Write Path, Campaign Fan-out

Write Path: Campaign Fan-out (write-intensive core)



Why this survives 200k writes/sec of burst

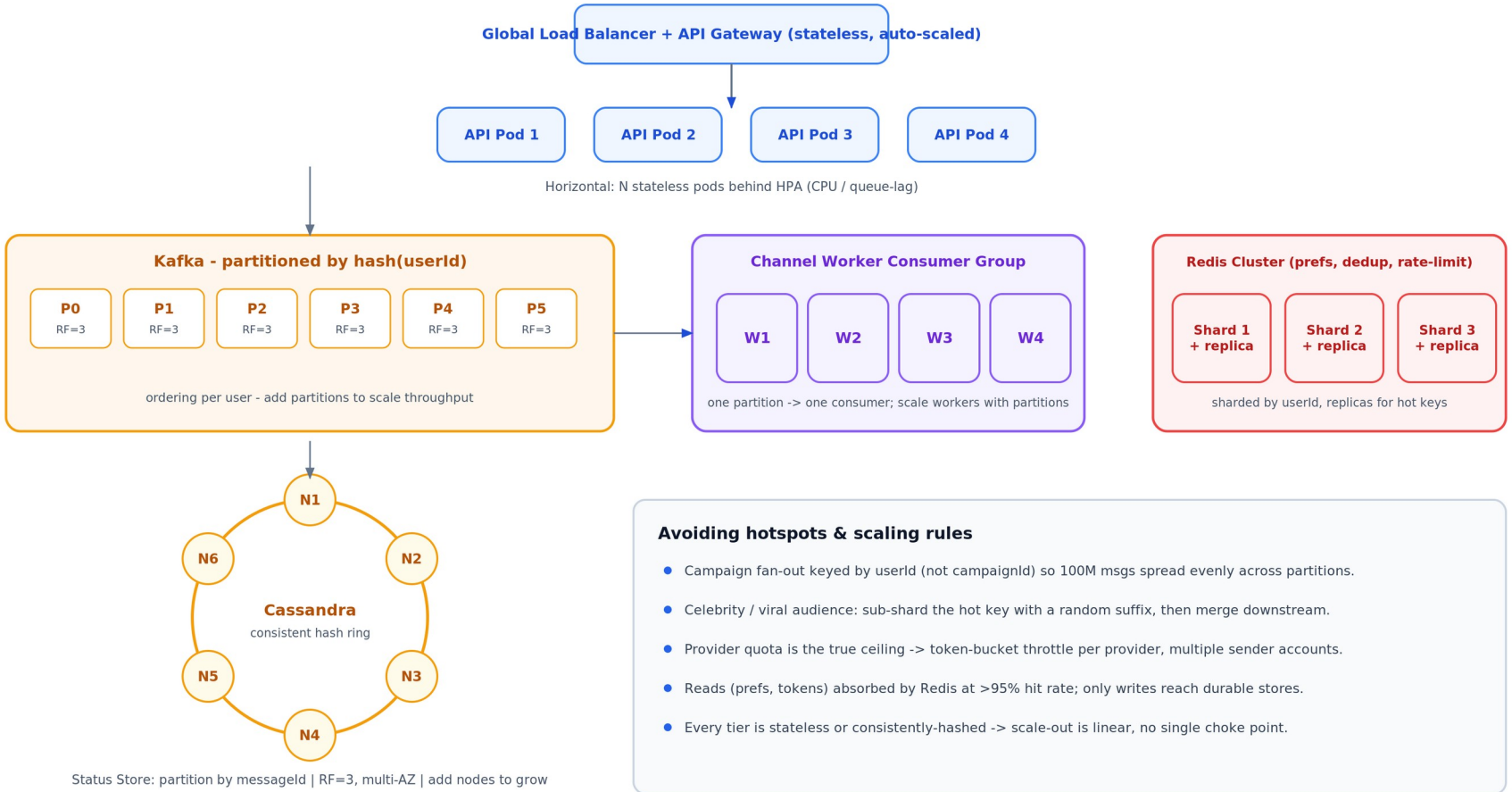
- Kafka is an append-only log: sequential disk writes absorb spikes; consumers drain at their own pace (buffer, not drop).
- Fan-out is async + chunked: one API call never blocks on 100M writes; progress is checkpointed and resumable.
- Status lives in Cassandra (LSM-tree): writes are memtable appends, no in-place updates, ideal for high write volume.

Peak write budget: 200k msg/sec fan-out + ~150k/sec delivery receipts = ~350k writes/sec at burst

SCALABILITY

Scaling & Partitioning Strategy

Scalability & Partitioning



Avoiding hotspots & scaling rules

- Campaign fan-out keyed by userId (not campaignId) so 100M msgs spread evenly across partitions.
- Celebrity / viral audience: sub-shard the hot key with a random suffix, then merge downstream.
- Provider quota is the true ceiling -> token-bucket throttle per provider, multiple sender accounts.
- Reads (prefs, tokens) absorbed by Redis at >95% hit rate; only writes reach durable stores.
- Every tier is stateless or consistently-hashed -> scale-out is linear, no single choke point.

SCALABILITY

Bottlenecks & Mitigations

Fan-out amplification	1 campaign call becomes 100M writes and can swamp everything downstream.	**Fix:** Async, chunked expansion; Kafka as a buffer with backpressure; checkpoint & resume.
Third-party provider limits	APNs/FCM/Twilio quotas are the real ceiling, not our own CPU.	**Fix:** Token-bucket throttle per provider; batch (FCM multicast); multiple sender accounts; failover.
Status-DB write hotspot	350k writes/sec can hotspot a single partition and stall.	**Fix:** Cassandra (LSM) partitioned by messageId; never key by campaignId; spread by userId.
Hot key / celebrity	A viral segment concentrates load on one Kafka partition or Redis shard.	**Fix:** Sub-shard the hot key with a random suffix; replica reads; merge downstream.
Single points of failure	A lost broker, cache or AZ must not drop transactional traffic.	**Fix:** RF=3 multi-AZ, consumer groups, circuit breakers, DLQ, active-active regions.

SCALABILITY

Horizontal vs Vertical Scaling

Rule of thumb: *scale horizontally wherever state can be partitioned or removed; accept vertical scaling only where volume is low and a single writer keeps the design simple.*

API, workers, fan-out	**Horizontal**	Stateless, add pods behind an HPA keyed on queue-lag. Linear, cheap, no ceiling.
Kafka	**Horizontal**	Add partitions & brokers. Partition count is the throughput dial.
Cassandra (status)	**Horizontal**	Consistent-hash ring, add nodes to grow writes & storage together.
Redis (cache)	**Horizontal + replicas**	Shard by userId; a hot key gets read replicas rather than a bigger box.
Postgres (metadata)	**Vertical + read replicas**	Templates/campaign config are low-volume, one strong primary is simplest & correct.

RELIABILITY

Data Durability & Backups

Never lose a write

- Producer ACK only after Kafka persists (acks=all, RF=3).
- At-least-once delivery + idempotency key = effectively-once.
- Poison messages routed to a Dead-Letter Queue for replay.

Survive failures

- Cassandra RF=3 across 3 AZs; hinted handoff + read repair.
- Kafka & Redis replicated; consumer groups rebalance on node loss.
- Circuit breakers isolate a failing provider from the rest.

Recover & restore

- Daily Cassandra snapshots + commitlog to S3 (point-in-time).
- Cross-region async replication for regional DR.
- Kafka retention lets consumers replay after an outage.

Recovery objectives: RPO ~ seconds (Kafka-backed) | RTO < 5 min for a single-AZ failure (automatic).

DECISIONS

Technology Choices, Justified

Concern	Choice	Why it fits	Alternative considered
Message bus	**Apache Kafka**	Durable append-only log, partition-ordered, replayable, proven at 1M+ msg/sec.	*SQS/RabbitMQ: lower throughput, no replay.*
Status store	**Cassandra**	LSM writes, linear scale, tunable consistency, built for write-heavy.	*Postgres won't take 350k writes/sec.*
Cache / counters	**Redis Cluster**	Sub-ms prefs, dedup keys, token-bucket rate-limits.	*Memcached lacks the data structures.*
Metadata	**PostgreSQL**	Relational templates/campaign config; low volume, needs ACID.	*NoSQL adds no value here.*
Blob store	**S3**	Cheap durable payloads + cold status archive.	,
Orchestration	**Kubernetes + HPA**	Autoscale stateless tiers on queue-lag.	*VMs: slower to scale on bursts.*

BONUS: ANYTHING EXTRA

Observability, Cost & Security

Observability

- Per-channel delivery funnels (sent -> delivered -> opened).
- SLO alerts on OTP p99 latency & queue-lag.
- Distributed tracing from API to provider callback.

Cost optimization

- Channel waterfall: prefer push > email > SMS (SMS ~100x).
- Batch email/push to cut per-call overhead.
- Cold-tier status to S3; spot nodes for bulk workers.

Security & privacy

- PII encrypted at rest & TLS in transit.
- Provider creds & keys in a secrets vault, rotated.
- GDPR: honor opt-out globally; retention-based purge.

These are what separate a working system from a production one, and where the last rubric points live.

REQUIREMENTS

Alignment & Trade-offs

FR: send to one user or 100M

Single API + async fan-out through Kafka partitions.

NFR: OTP p99 < 2s

Separate high-priority topic + workers; never queued behind marketing.

NFR: no lost transactional msg

Persist-before-ACK + at-least-once + idempotent dedup.

NFR: ~350k writes/sec

LSM status store + append-only log + userId partitioning.

NFR: consent & compliance

Preference check in fan-out; GDPR purge; vaulted secrets.

Bottom line: every rubric requirement maps to a concrete, defensible design decision, and the numbers drive the design, not the other way round.

Deliberate trade-offs

- Eventual consistency on status (favor availability & write speed over instant global reads).
- At-least-once + dedup instead of exactly-once (cheaper, and dedup covers the gap).
- Not over-engineered: metadata stays on Postgres, no distributed DB where volume is low.

THANK YOU

A notification system that scales with the numbers, not around them.

Yahya Ahmed
yahyaahmedg@gmail.com

System Design for Everyone, Capstone Project, Cohort 4 (2026)