High Level Design

~~Coding~~

1. Problem Statement
2. Functional Requirements
3. Non-functional Requirements
4. Scale Estimation
5. Architecture Design

SDE-2 or higher positions – architectural design decisions

Any tier-1 company – have atleast 1 System Design round (HLD)

Staff Engineer @ Google.] 1-2 Crore per annum in India

10+ yoe

Interview Question for Staff Engineer @Google → HLD round

List of strings → sort them in dictionary order

Cat
dog
apple
high level design
Scale

→ Sort

apple
Cat
dog
high level design
Scale

Catch*

50 Petabytes of data

50,000,000 GB of data

v. large Scale

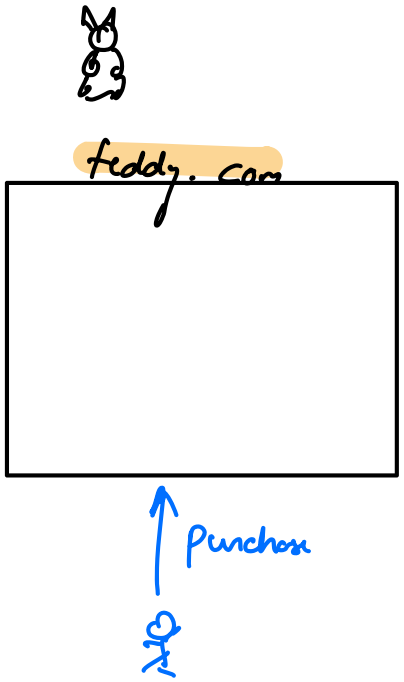
Byte – 8 bits
KB – 1000 B
MB – million B
GB – billion
TB – trillion
PB – quintillion

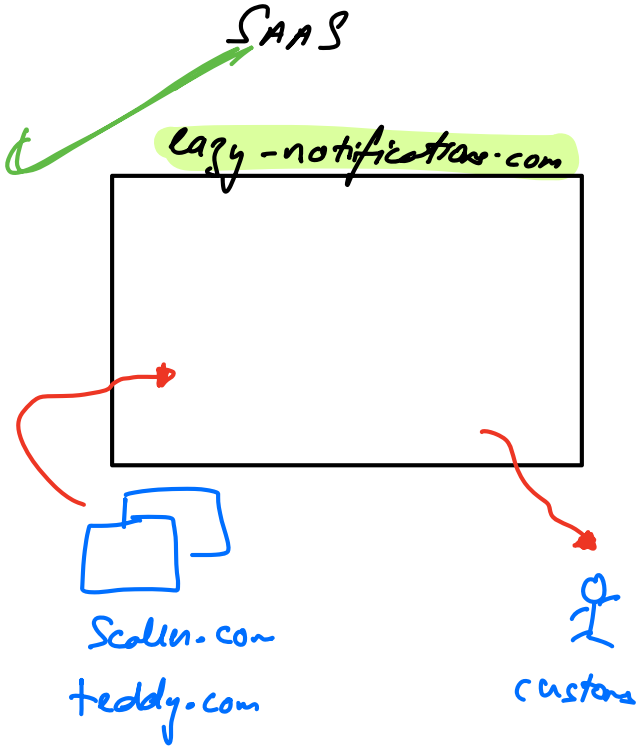
Simple Problems → (v. large Scale / Planet Scale) v. difficult

High Level Design

① Problem Statement

Notifications delivery service





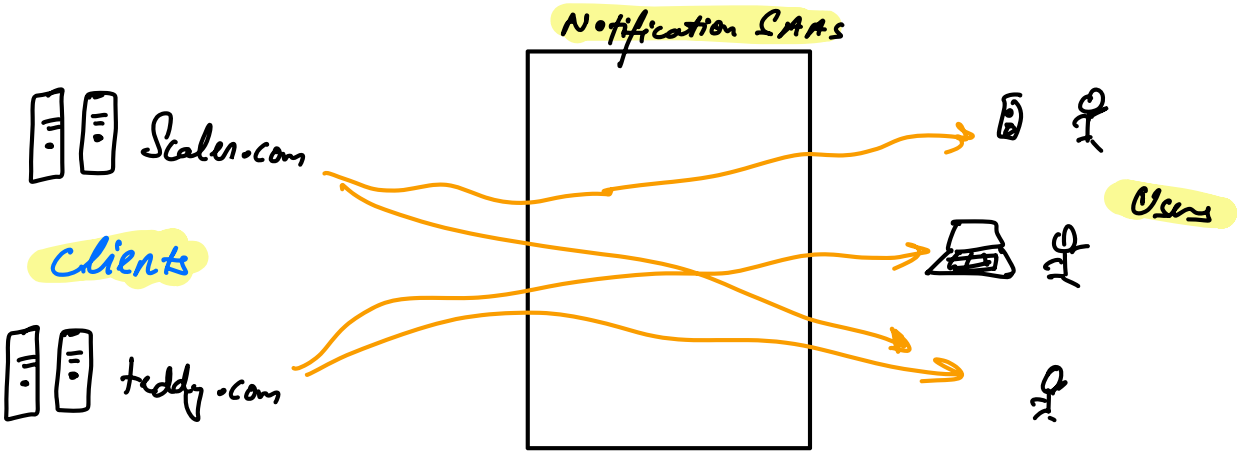
- Needs to send notifications to its customers
- Primary function - sell product

- Primary function is notifications.

② Functional Requirements

MVP features — dont think of fancy features

Minimal Viable Product



Client Requirements

① Clients should be able to send notifications to users

notify (client-id, user-id, message) → success / failure

client-id: Sender
user-id: recipient
message: payload

② Support multiple delivery channels

SMS / Email / in-app

Slack / whatsapp / Pigeon

Extensible / Pluggable

We should be able to add new delivery channels without changing infrastructure

(3) client should be able to broadcast messages to a group of users

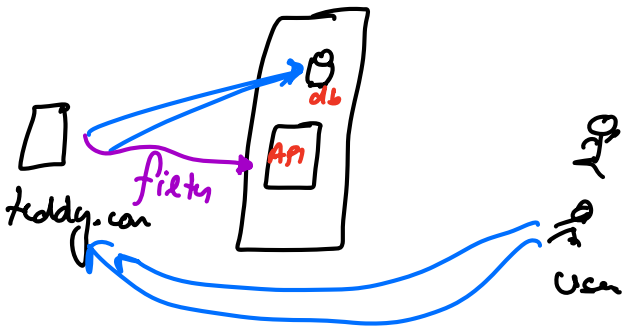
notify (client-id, [list of user-id], message) → success / failure

client-id: sender
[list of user-id]: recipients
message: payload

bad idea!!

Req Payload is too large

- clients already store their customer contact info in our DB
- clients can specify a "broadcast filter" to send bulk messages



broadcast (client-id, filter, message) → success/failure

filter (example): (user.is-subscribed-to ("1224") OR user.last-login > 24 hours)

User Requirements

1. Users should be able to recieve notifications
   - in realtime (if they are online)
   - On re-connecting (once they come back online)

2. Users should be able to customize their notification preferences

Select channels

- [x] Email
- [x] SMS
- [x] Whatsapp
- [x] Pigeons

Notification Type

- [x] Marketting
- [x] Trending/News
- [x] Financial
- [x] OTP

Hidden Requirements

① Rate limiting

- Client Limiting
  - free : 100 notification /day
  - Paid : 50,000 notification /day
  - enterprise : upto 50 M notificat /day
- User Limiting
  - a user should not get more than ___ notification in ___ time

② Observability

back in 2020 Google had ~ 11 Million slaves

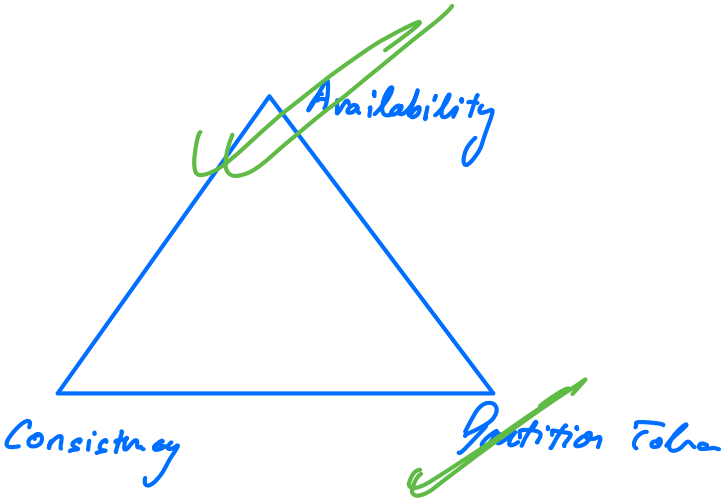
- detecting that something is wrong
  - metrics like # notifications sent /min
    - # of errors / minute
    - avg CPU load
- diagnose
  - search logs
- verify the fix

③ Non-functional Requirements (Design Goals)

CAP / PACELC theorem

Eventual Consistency vs Immediate Consistency

for some time your data can be stale



choose 2
- AP
- CP

Imagine Mala has purchased a Teddy

& teddy is out for delivery

teddy.com → Mala (notify)

in worst case when systems are down

for some time when mala checks her phone she gets no notification

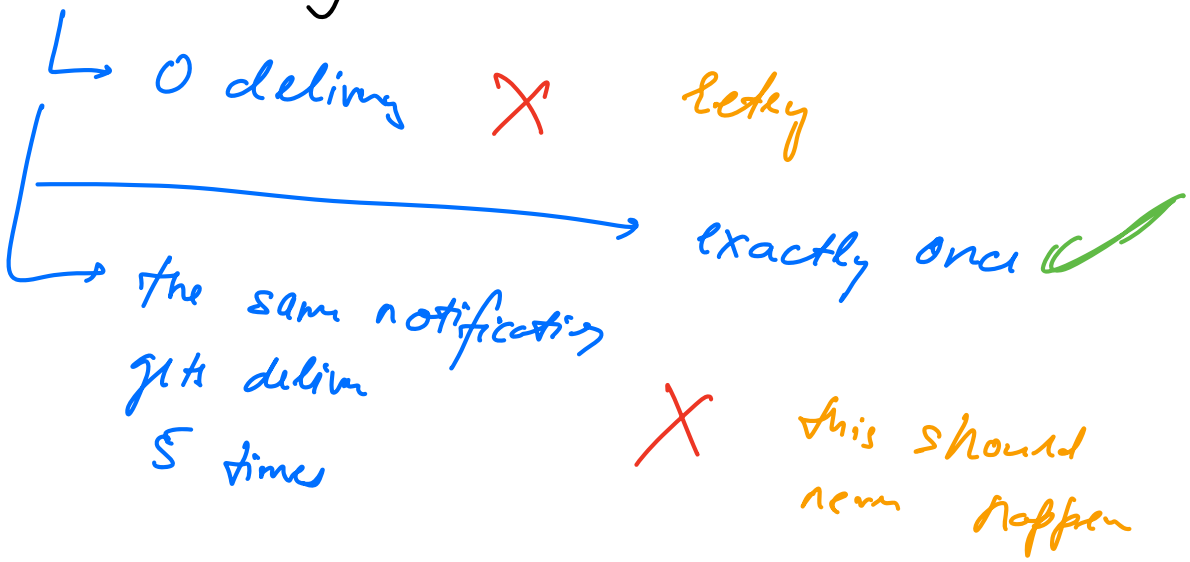
but after some time Mala gets notified

Okay

usually

Mala gets notified immediately

- Compliance — TRAI /FDR/...
- Priority — OTPs should be prioritized over marketing notifications
- Exactly once delivery



- ~~Security~~ — Leave this to security experts

(4) Scale Estimation

↳ it is okay to be a little off

↳ it is okay to overestimate

Total # users ≈ 2 billion

Avg # of notifications / user / day ≈ 10

Avg # Notifications / second = 200,000

2×10^9 users * 10 notification / user / day

$= 2 \times 10^{10}$ notifications / day

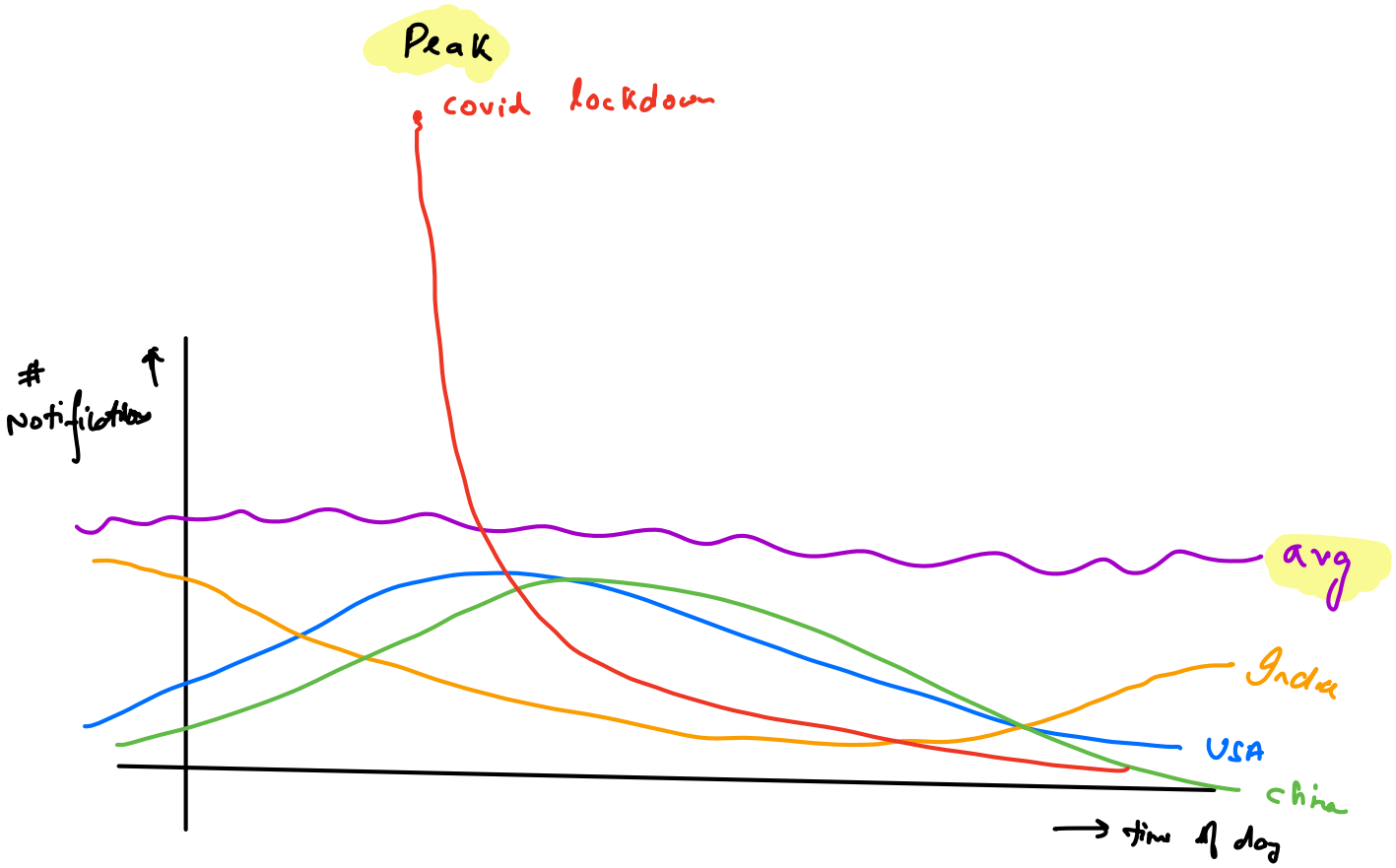
↳ 60 * 60 * 24 seconds

= 86400 seconds

≈ 10^5 seconds

$= 2 \times 10^{10^5}$ $\frac{\text{notifications}}{10^5 \text{ seconds}}$

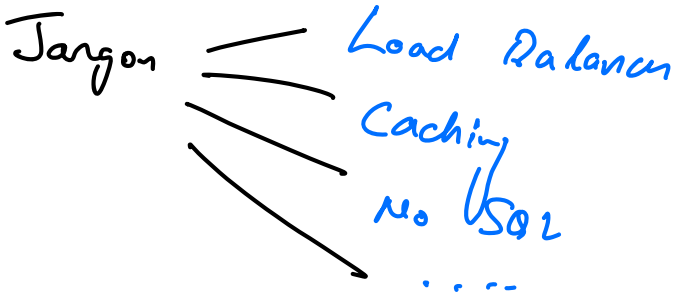
≈ 200,000 notifications / sec

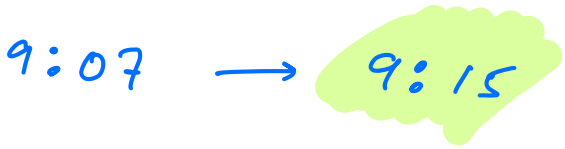


Peak could be 50x higher than avg

Peak # notification / second = 200,000 avg/sec * 50

= 10 million notifications / sec


Jargon
Load Balancer
Caching
No SQL
....


9:07
9:15

Scaler HLD Curriculum

- **How backend systems work**
  - DNS / vertical vs horizontal scaling / load balancing / sharding
- **Load Balancing & Consistent Hashing**
  - Healthcheck & Heartbeat / Roll outs / DNS as load balancer / Untrusted Layer / Consistent hashing - deep dive
- **Caching**
  - Browser / CDN / Backend
  - Local vs Global / Single vs Distributed
  - Redis
  - Cache Invalidation
    - Write through
    - TTL
    - Write back
    - Write Around
  - Cache Eviction
    - FIFO
    - LRU
    - LFU
    - MRU
- **Caching Case Studies**
  - Scaler Code Judge / Scaler Leaderboard / Facebook News Feed

- **CAP theorem & Replication**

  Availability / Consistency / Partition Tolerance

  Consistency: Immediate / Eventual

  CAP theorem / PACELC /

  Master-Slave: Eventual / with Synchronous

  Tunable Consistency: Quorum

- **SQL vs NoSQL**

  Pros & Cons

  Sharding Key

  NoSQL Types: Key-Value / Document / Wide Column / Search / File store

  Graph / Vector / In-memory / timeseries

- **Database Internals**

  Orchestrator / Config Management / Reserved Servers / Shard Creation / Multi-master

- **Database Internals**

  LSM Trees / Bloom Filters / Sparse Index / Compaction / MemTable / WAL

- Case Study — Google Typeahead

  Trie / Redis / Recency / Sampling / Batching / Precomputation

- Case Study — Messaging apps

  Slack groups / Whatsapp / Facebook messenger

- Case Study — Kafka + Zookeeper

  event driven architecture / Pub-Sub / topics & partitions / msg order

- Case Study — Elastic Search

  Inverted Index / TF-IDF / Sharding / Tunable Latency

- Case Study — File Storage

  Upload / download / chunking

- Case Study — Uber

  Quad Tree / traffic patterns / request driven

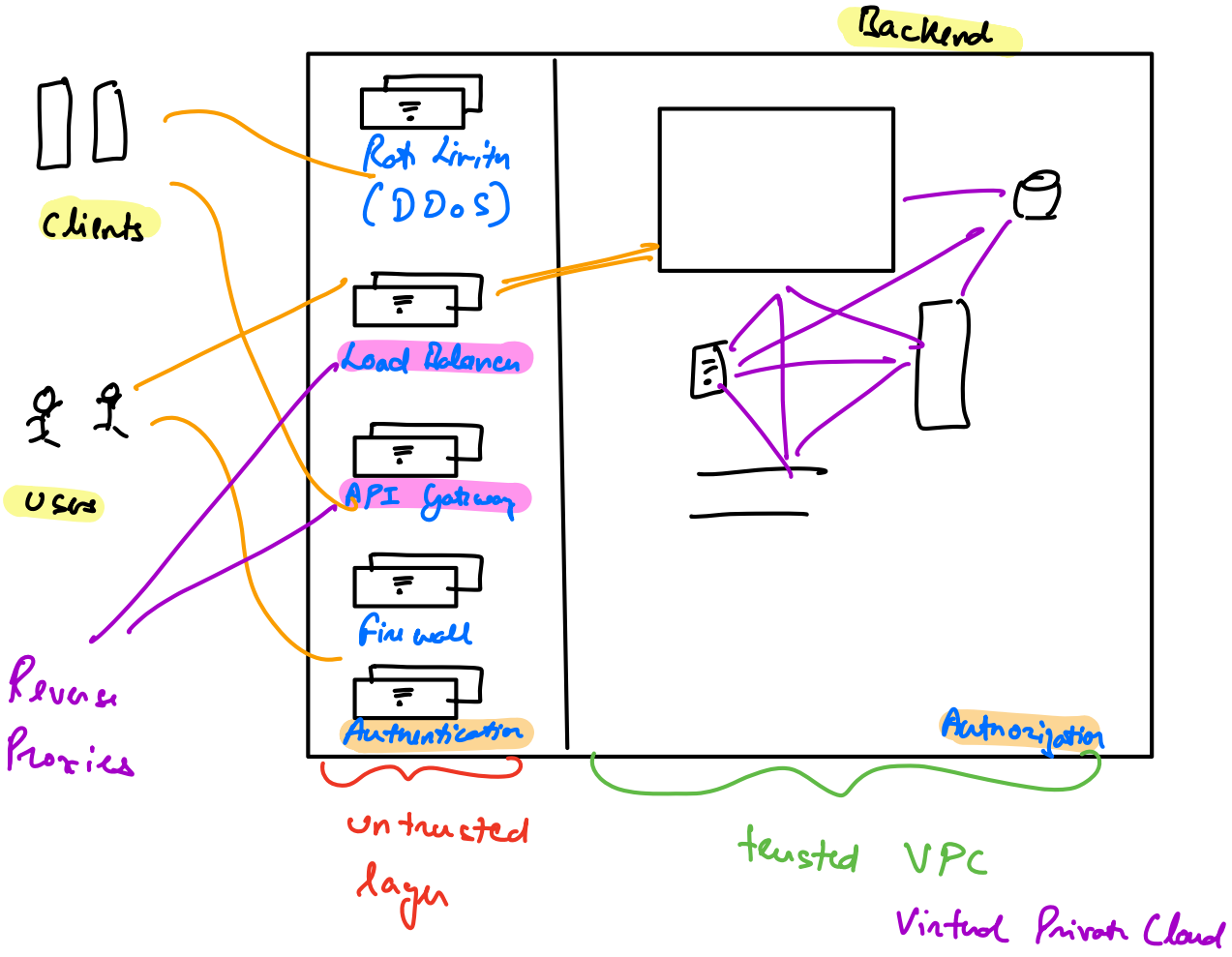
- Case Study — Rate Limiting

  DDoS / Token bucket / Leaky Bucket / Sliding window

- Case Study — Id generation
  - idempotency / Unique / incremental
- Case Study - Video Streaming
  - Netflix / Youtube / Hotstar
  - Concurrency
  - ABS / chunking / CDNs
- Microservices
  - Monolith vs Microservices vs Service Oriented
  - Breaking a Monolith
  - Observability / Distributed Tracing / Logging
  - Thundering herd / Cascading failure / Circuit Breaker
  - Distributed Transactions
    - Two Phase Commit / Saga / CQRS /
    - Orchestration & Choreography
  - Event Sourcing

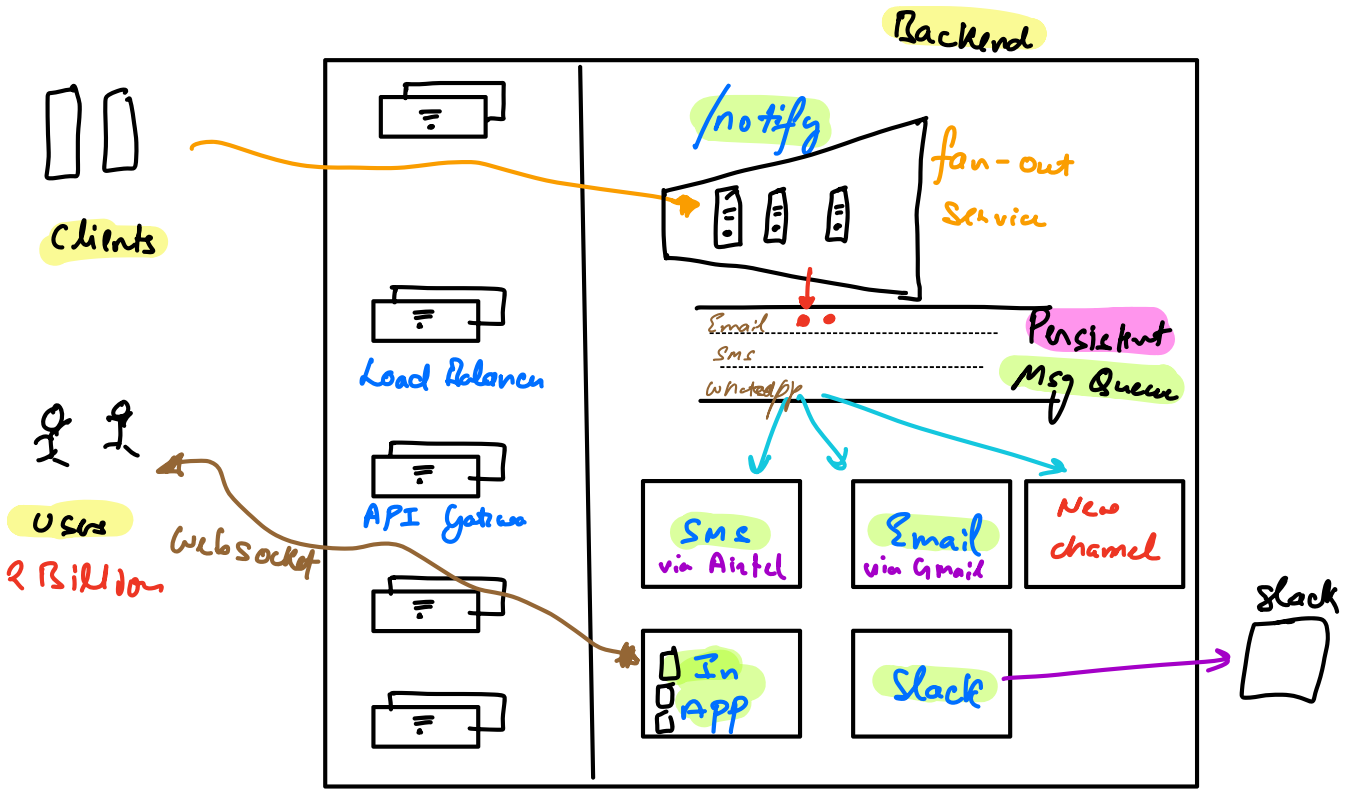
⑤ Architecture Design

Generic Design



① Clients should be able to send notifications

User should be able to recieve notifications



① user is online & connected → send notification ASAP

② user can be offline → they will come back online after some time

Only for in-app

any notifications that occured when they were offline

Microservices → Event Driven Architecture

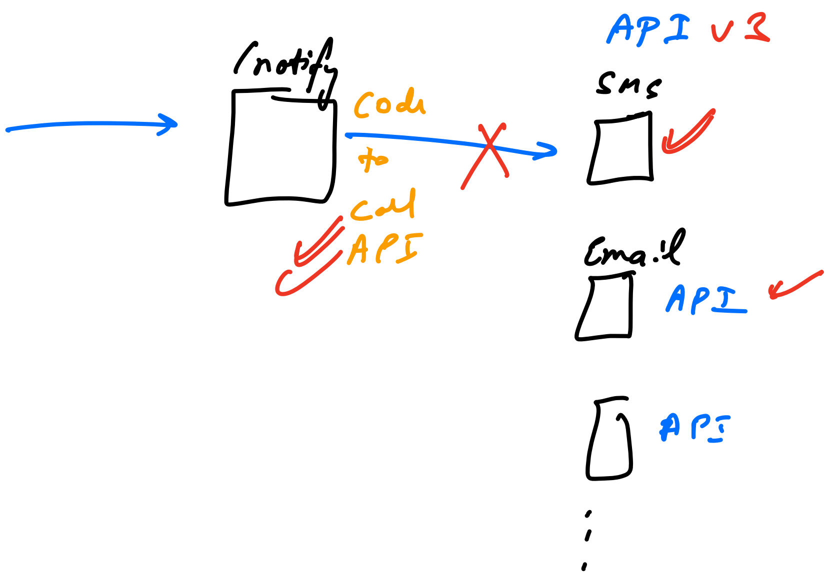
Publisher Subscriber model

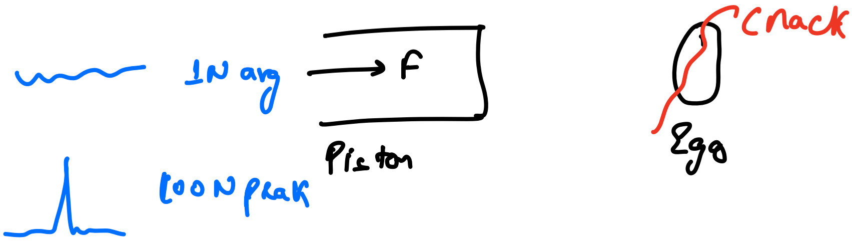
aka Producer Consumer.

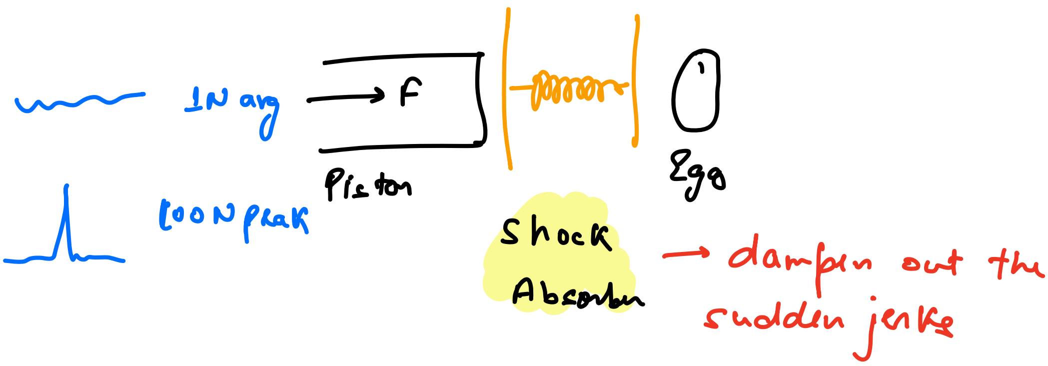
via Message Queue like Kafka

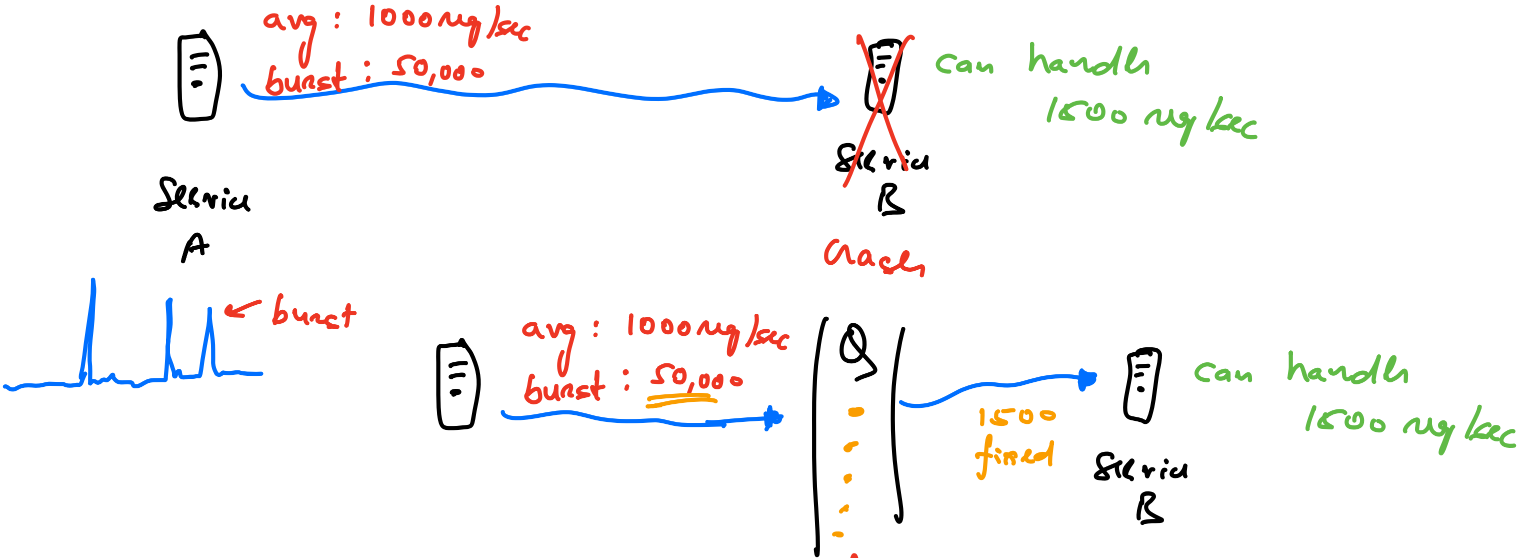
Pub sub = Producer Consumer

≠ Observer ≠ Event Handling


notify
code to call API
API v3
SMS
Email
API
API


IN avg
100N peak
F
Piston
crack
Egg


IN avg
100N peak
F
Piston
Egg
Shock Absorber
→ dampen out the sudden jerks


avg : 1000 msg/sec
burst : 50,000
Service A
can handle 1500 msg/sec
Service B
crash
burst
avg : 1000 msg/sec
burst : 50,000
1500 fixed
can handle 1500 msg/sec
Service B

↓

Q acts as a buffer
- prevents notifications from getting dropped
- allows service from crashing

notify (client-id, user-id, message)

json

msg:

type: OTP / Marketing / Finance / News
used for priority setting

channels: {

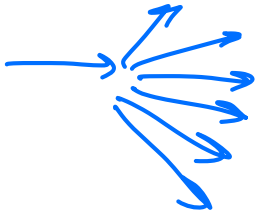
sms { msg: — emoji: — }

email: { subject: ... body: — }

in-app: { msg: — icon: — action-url: 'open(screen 2)' }

}

fan-out: 1 to many

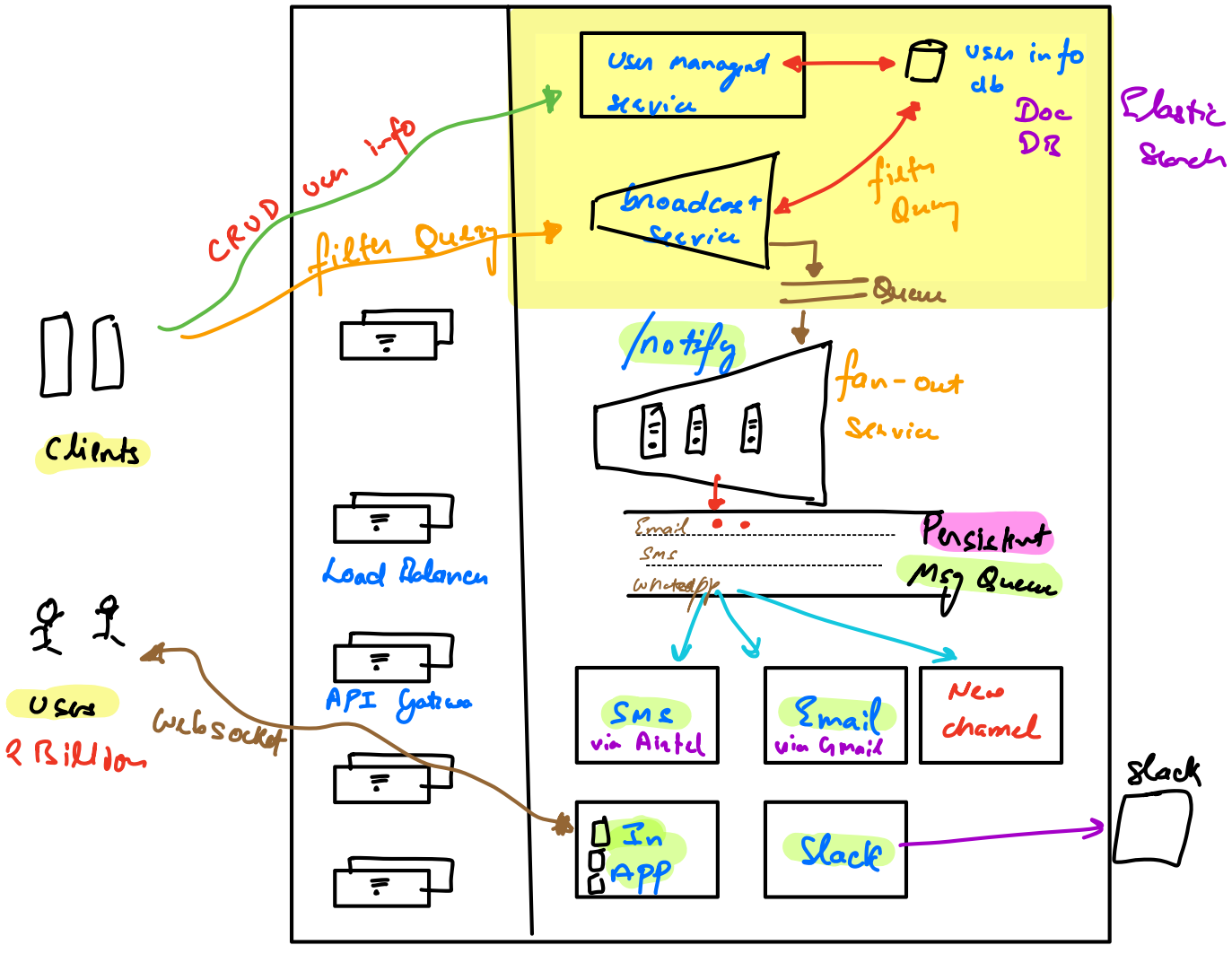


Partition the Msg Queue — handle load
— Ensure msg order

Create various topics — one for each channel (Kafka)

Each topic can be partitioned based on the avg load in that channel

② clients should be able to broadcast notifications



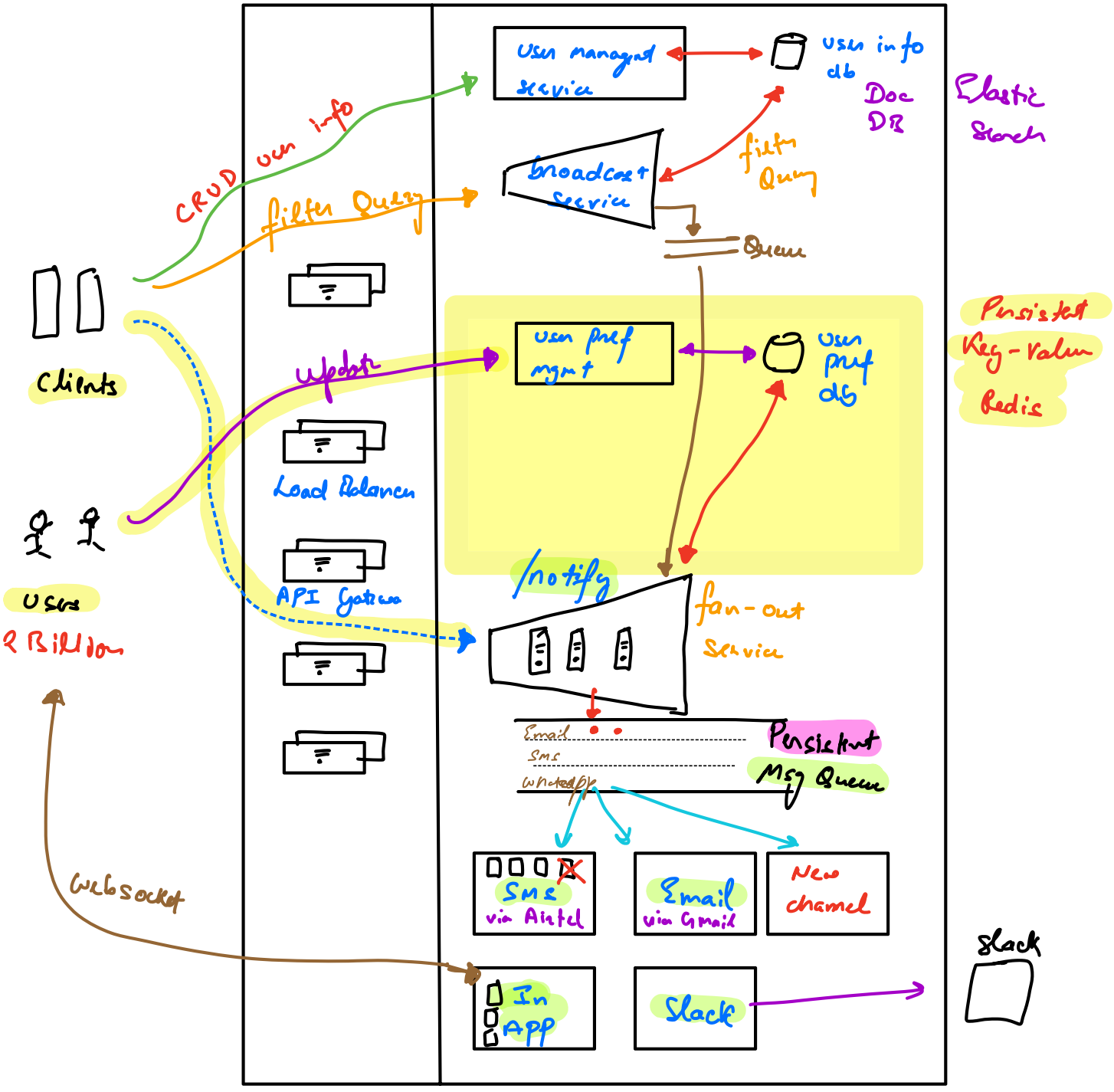
choice of DB for client-user info

	Example	Pros	Cons
SQL	Postgres MySQL SQLite	• ACID • Normalization • joins • complex queries • relations ⋮	• Rigid schema • don't scale horizontally (unless you shard yourself)

NoSQL — built for scale (automatic sharding, replication..)

Document	Mongo DB Couch DB Cockroach DB Elastic Search	• unstructured data • full text search	• no global index • no relations • simple queries
Column	Cassandra Scylla Big Table Hbase	• fast analytics (aggregations) • pagination by time • ultra high writes	• no relations • simple queries • reading via multiple cols is slow
Key-value	Redis Memcache Dynamo DB	• v. fast reads & writes • v. simple	• no schema • no relations • (volatile)

3 Users should be able to manage notification preferences



Choice of db for user pref data?

- Simple data — given user whether a particular channel is enabled or not

User → channel → true/false

- 10 M reads/sec

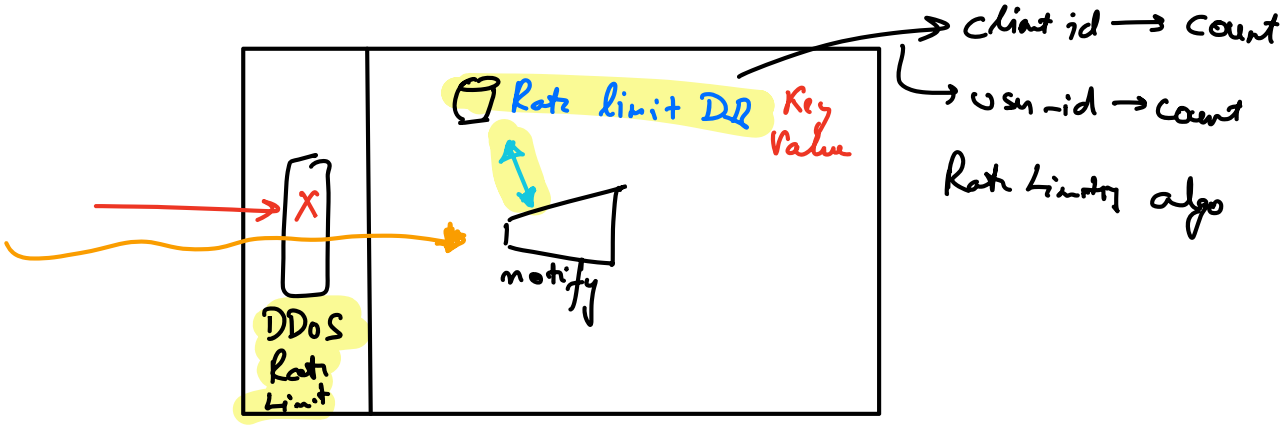
Key-value (Redis + Persistance)

④ Rate Limiting

↳ Prevent DDoS

↳ Client Rate Limits

↳ User Rate Limits



observability