Summary of Last Class

- Pros & cons of monolith
- Break a Monolith < SOA / Microservices
- Pros & cons of Microservices

started with solutions

Cons

1. Debugging across services is hard — Observability
2. Consistency " " " " — Distributed Transactions
   - 2PC
   - Saga — Orch. / Chor.
3. Communication is hard — Event driven architecture — CQRS
4. Robustness — Circuit breaker

Observability

Solution / code is never perfect — always be some bugs / corner cases)
errors (n/w / disk)
requirements change

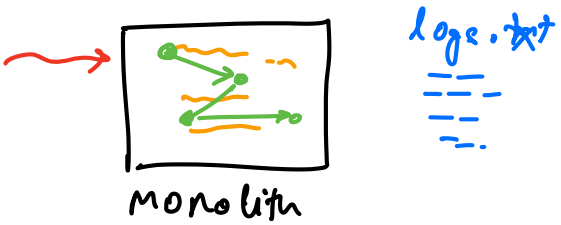
Errors are okay as long as they don't pass silently.

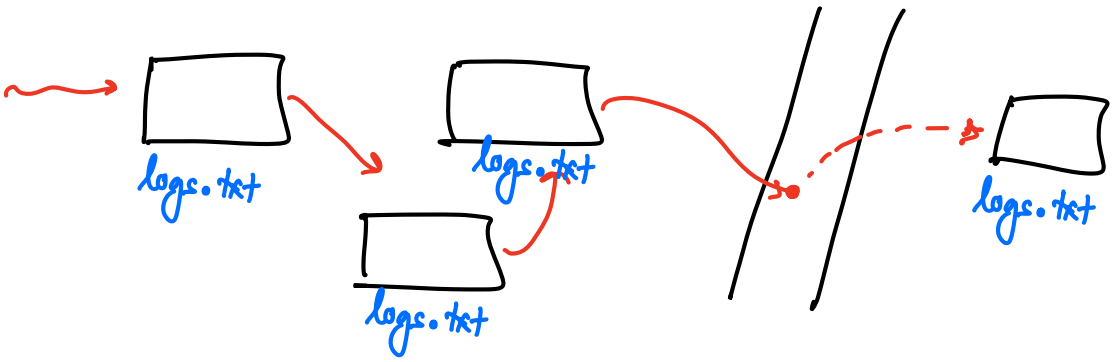
1. Detection — How will I get to know that something is wrong?
2. Diagnosis — Identify where the problem occured
   machine / service
   & identify what caused it

③ Verification Once fixed, how do I verify that the fix worked?

Diagnosis

Tracing a request





① when a req arrives at backend (LB) you can generate a unique correlation-id / request-id / trace-id / msg-id ..

Every time you take an action based on this request or forward this req

You MUST pass this correlation id

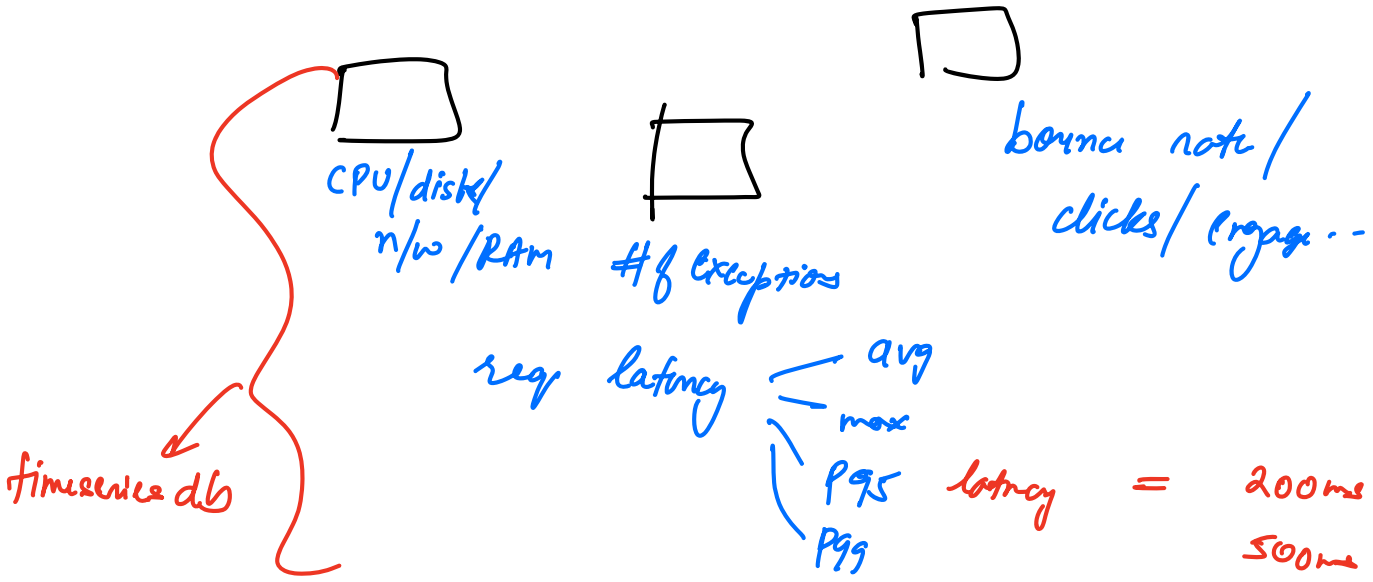
② Aggregating Logs & Searching

Elastic Search / Logstash / Kibana → UI / Analytics platform

↓

central repo for storing all logs

Detection & Verification — track certain metrics



should this data go into logs.txt? No!

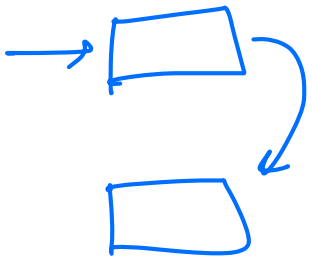
You only want aggregate info
- count
- avg/max
- avg errors since last week

Sentry / Prometheus

Alerts — if # of exception /hour > 5,000
- detection
- → send SMS/Email → ---

verify — have # of exception improved?
- metric

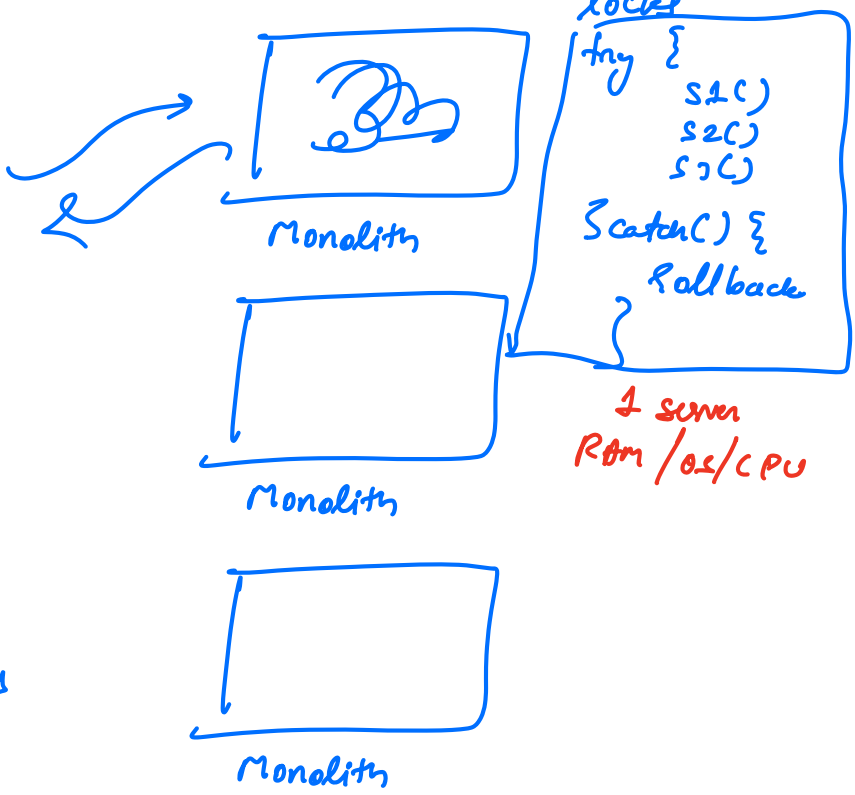
Consistency



Microservices – any transaction might span multiple servers – v. hard

CAP

PACELC



Challenge: Consider two services A & B

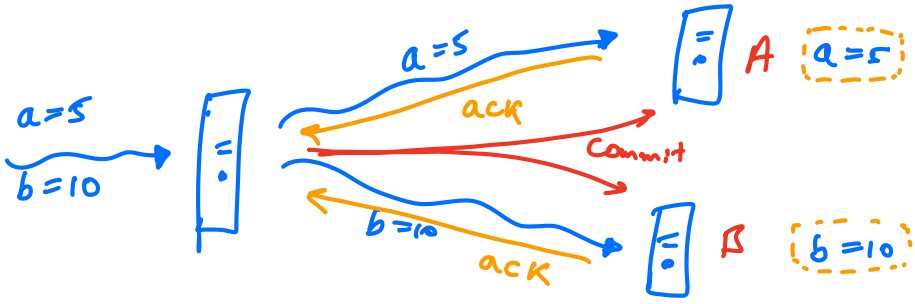
you wish to make a transaction that has multiple steps

this transaction requires steps to be executed in both A & B

has to be atomic – either full failed / fully successful

How to do this nicely?

① Two Phase Commit



① Prepare

Take note of this info

do NOT commit it

Please acknowledge

once ack you're NOT allowed to go back

② Commit

you had promised (ACK) some data earlier

Please commit it

When these servers (A & B) save data & give ack they MUST acquire locks. — No other reads/writes are allowed.
on Row/Key
v. Poor performance

∵ the state diagram is v. complex & has high depth
v. poor latency.

∴ 2PC is considered an anti-pattern in microservices

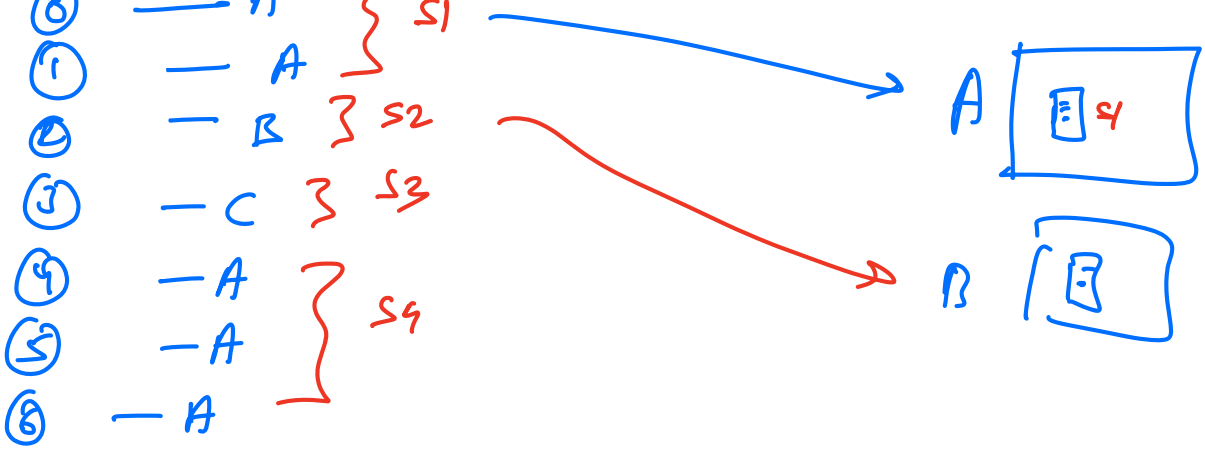
pattern : common & robust solution that is widely accepted as correct

anti-pattern : common but flawed solution

② Saga Patterns

Break down a large & complex transaction (involves lots of steps that span across multiple services) into smaller 'Sagas'

→ each saga should be 'local' to a specific microservice
You should NOT need to talk to multiple microservices in any saga



Example Person A ——1000Rs——→ Person B

axis Bank SBI

① Ensure A has sufficient balance } S1 Axis

② deduct 1000 Rs from A

③ add 1000Rs to B } S2 SBI

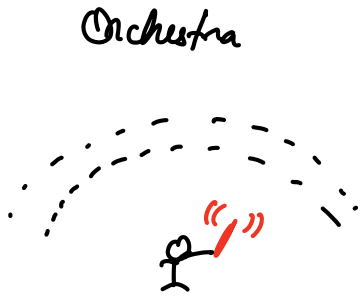
- Execute S1
  - fail — show error
  - succeed
    - execute S2
      - succeed — success (done)
      - fail — rollback ALL earlier steps

How to ensure that sagas are executed reliably?

- Orchestration — Music
- Choreography — dance

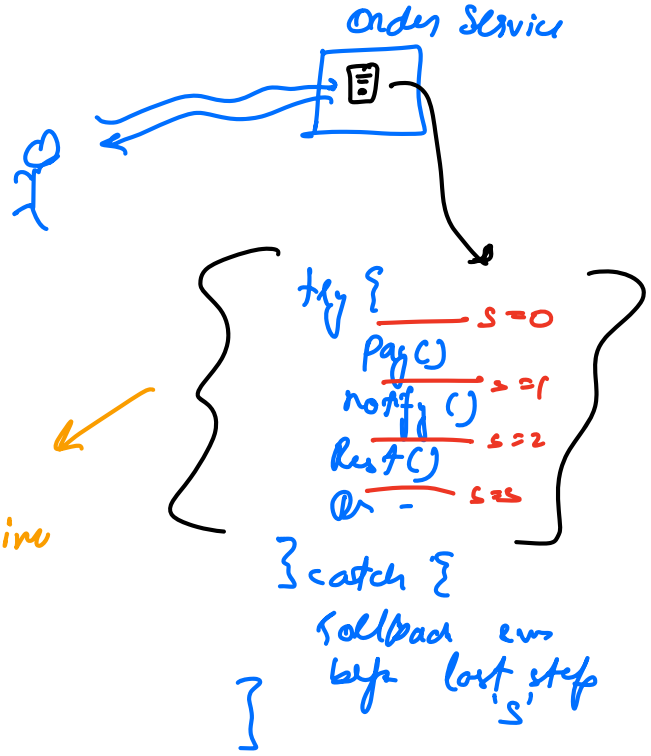
Orchestration — central authority that oversees the execution of Sagas

a dedicated microservice to handle the transaction.



Swiggy — order food

1. Order placement — SQL
2. Payment
3. Notification
4. Restaurant
5. Delivery assignment
6. Notification



Pros Simple to understand

Cons
- Not simple to code — State Machine
- SPOF — even if we have multiple servers for orchestrator if an orchestrator server dies it becomes v. v. difficult to reproduce the faulty state & correct it

detect? State is stored in DB

Choreography

- every service knows what it does
- each service is ONLY concerned about itself!!

↓

Inherently Event Driven

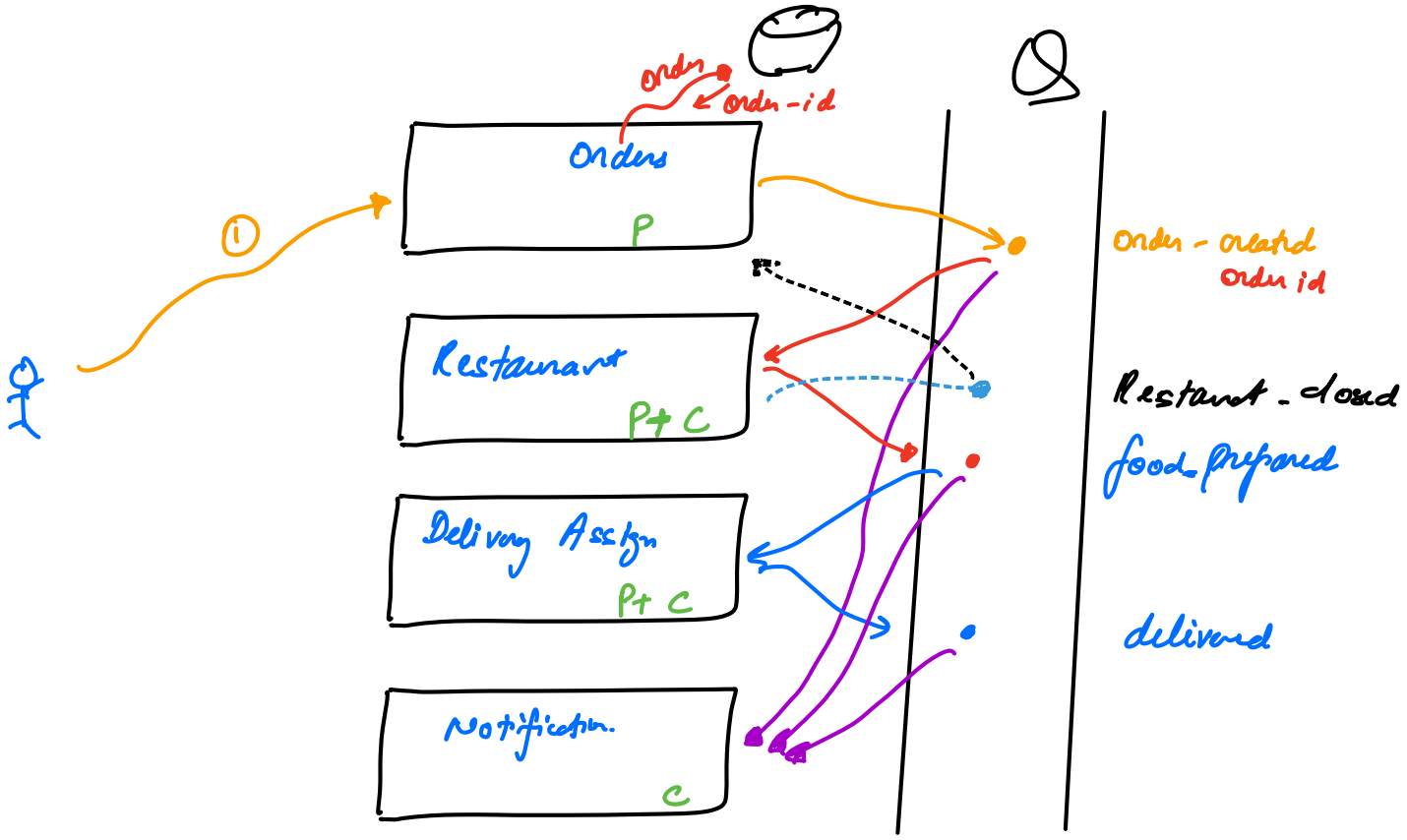
Inherently Distributed — no single orchestrator.

Once a service has done its job it will

Communicate to others that a task has been done.

ANNOUNCE & forget

- I don't care who listens to it
- My job was to announce

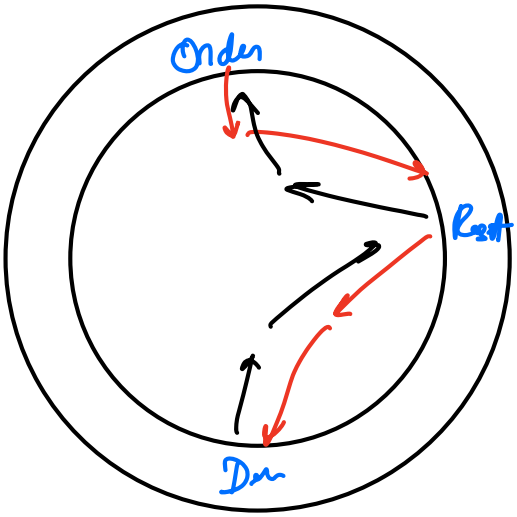


In choreography — each service is (usually) both a producer & a consumer.

- consume tasks that it wants to do
- perform task
- announce task completion.

How to handle failure in Choreography

- each service must also announce failure
- each service must consume failure of the immediately next service



Dead Letter Queue

- if a service fails to do a task it might want to retry before announcing failure.
- if all retries fail you do NOT want to lose this req completely, you also do NOT want to keep retrying it

- drop this msg into a separate topic in Msg Q
  ↓
  Dead Letter Q

Choreography vs Orchestration

choreography is generally better than orchestration.
- ↳ things are more reliable
- ↳ msgs cannot be lost ∵ Kafka
- ↳ if a service fails — no issue ∵ msg can be consumed again.

Cons of choreography

① high latencies — ∵ inherently event driven

② observability is harder — need to have dedicated infra & tool for observability.

10:55 → 11:05

CQRS (Command / Query Request Segregation)

treat 'commands' (write/update/del) separate from 'queries' (Read)

Facebook — 2 microservice — Messages / Notifications

Users — 2B+

Pages (celebrity) → few pages only.

every time a msg is sent (to user/page) a notification is triggered.

find 1 — How many notifications does a page get on avg? Notification
2 — " " messages " " " " msg.

3 — what pages get more messages but less notifications.

to ans this (3) you need to join the data b/w Msg & notification service.

join small portion of overall data

① ask msg service (msg) for this data

Msg service will read its own DB

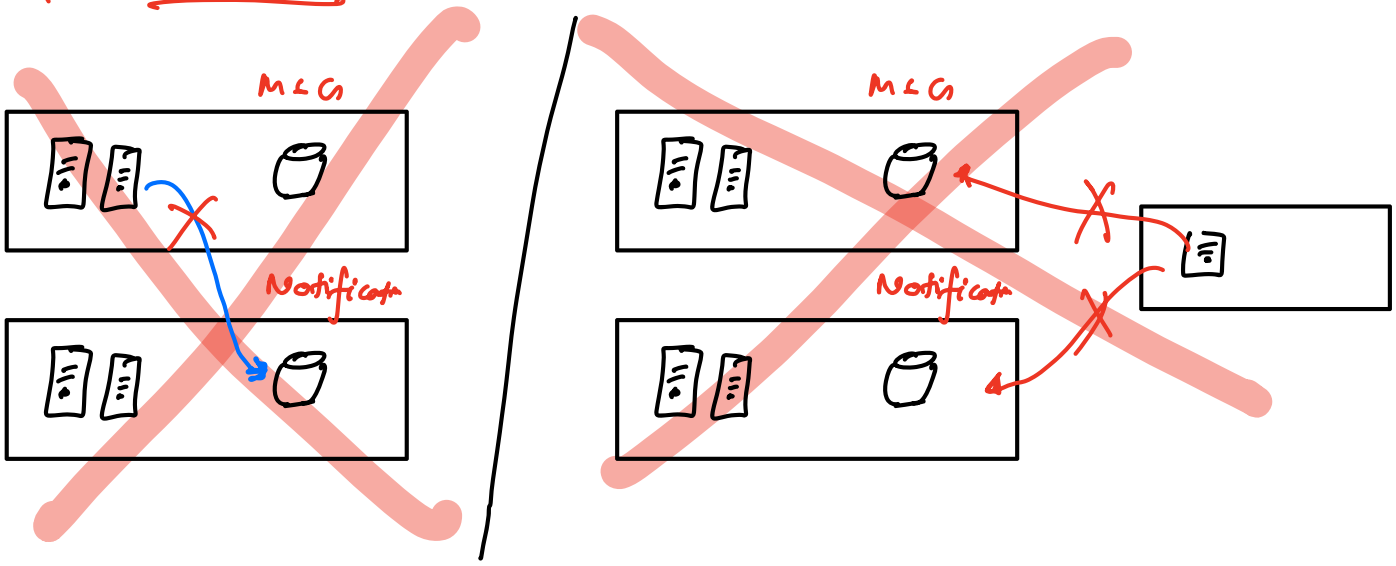
read the DB of notification service ✗ violation of separation ?

join

return.

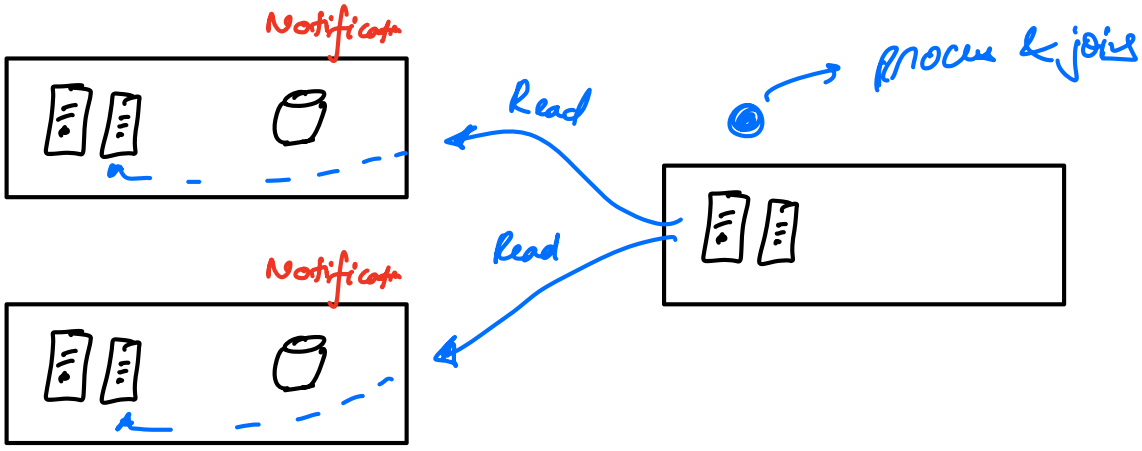
Concerns.

~~2~~ 2nd service that queries db of both msg & Notifn



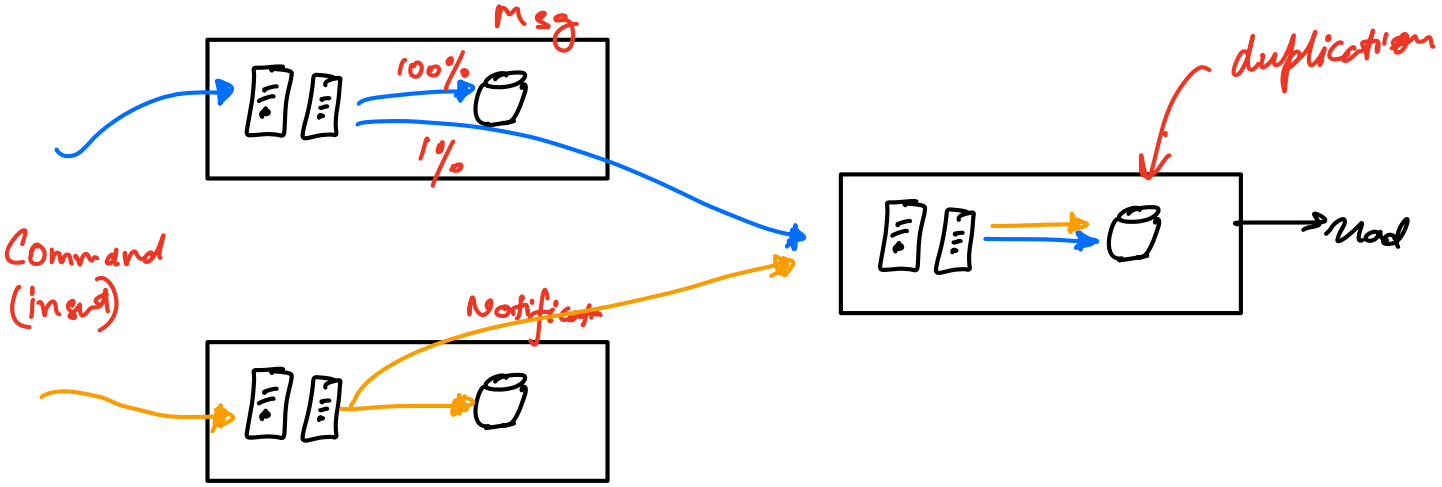
3

Anti Pattern.



4 CQRS — segregate reads & writes

any reads for the joint data go to the other service



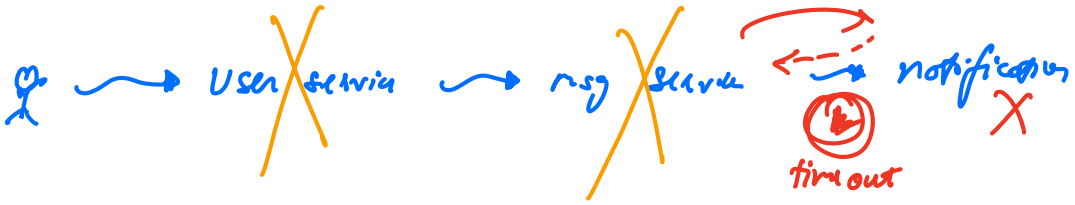
CQRS — instead of fetching data from multiple services A, B, C...

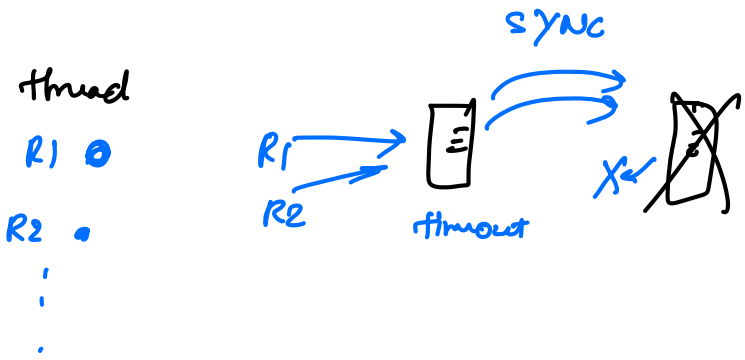
You push data from multiple services into another service X so that any read queries for the joined data of ABC can go to X

Circuit Breaker Pattern

multiple microservices — have dependencies

even more relevant when we are making sync calls





Cascading failure (sync calls)

when notification service is 'partially fixed'

∴ if the piled up messages it will be slow to process new requests

Thundering Herd

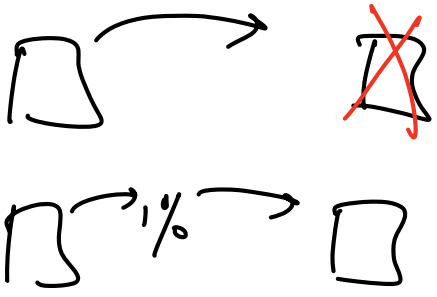
Error
- Regular — okay — Service is healthy
  error was because of other reasons
  auth fail / rate limit / 404
- Non-transient
  - bad — service is unhealthy — serious issue

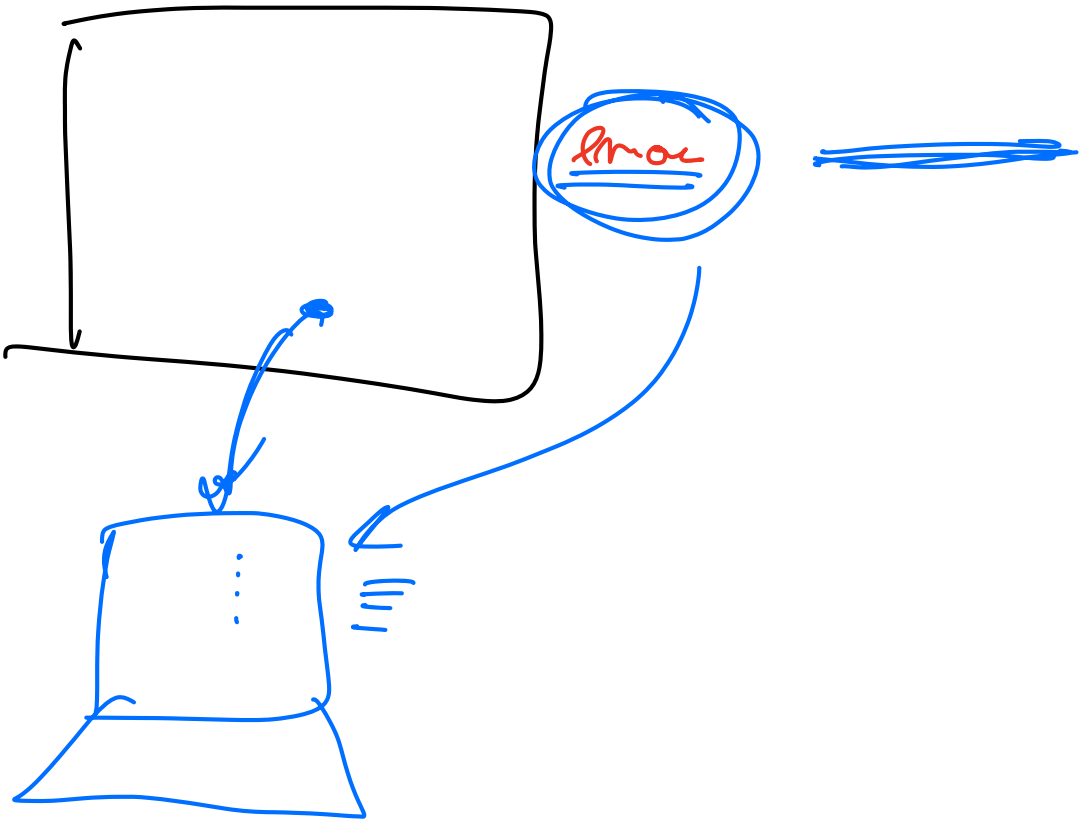
Service
- down — open (issues) — unhealthy
- partial — half open
- recovered — closed (issues) — healthy

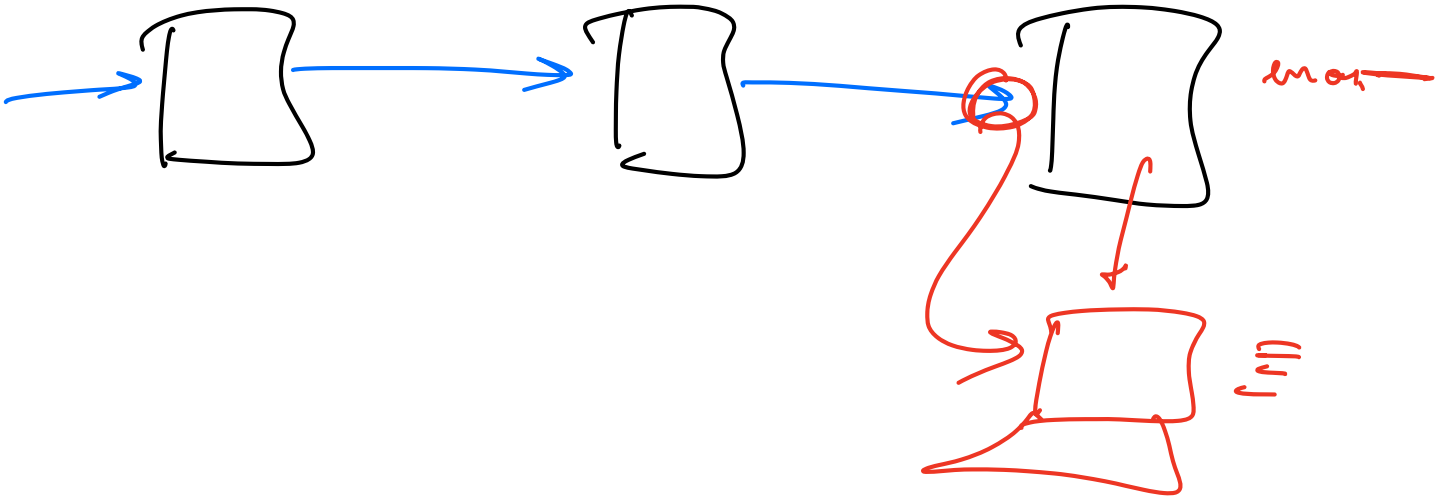
States are maintained in Redis in 1 place for all services

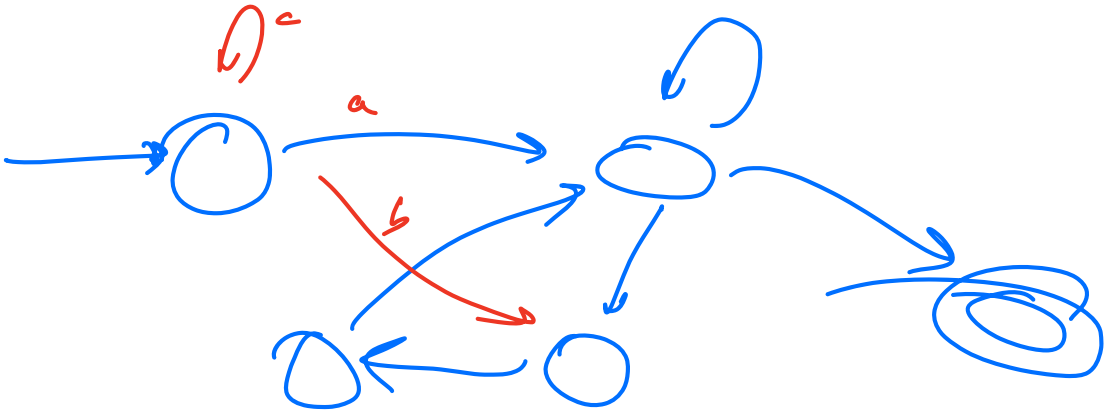
- Constantly monitoring metrics (# of req successfully processed / min)
- By default each microservice starts off in closed state — healthy
- if suddenly the metrics fall — open — unhealthy
- any service dependent on unhealthy service will stop sending additional requests
- eventually this failed service will be fixed. — Half open
- monitor
  - fail → unhealthy
  - rise → healthy

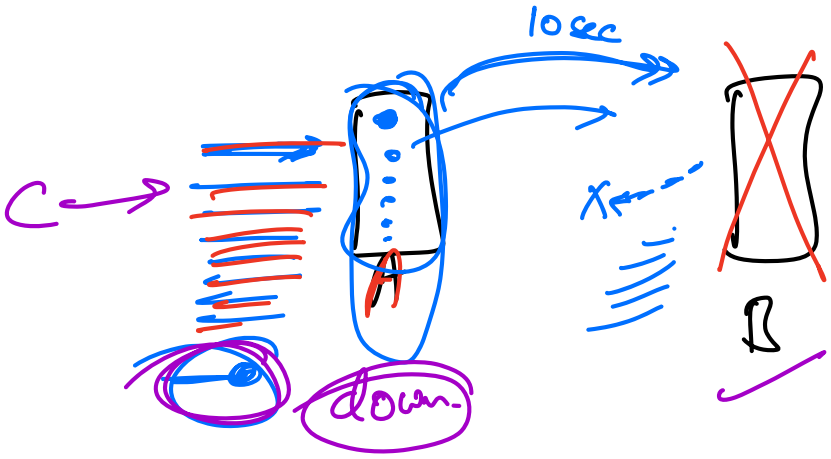
calling service will only send a % of its req




lmax


lmax


a
a
b


10 sec
down