CS6650 Final Project

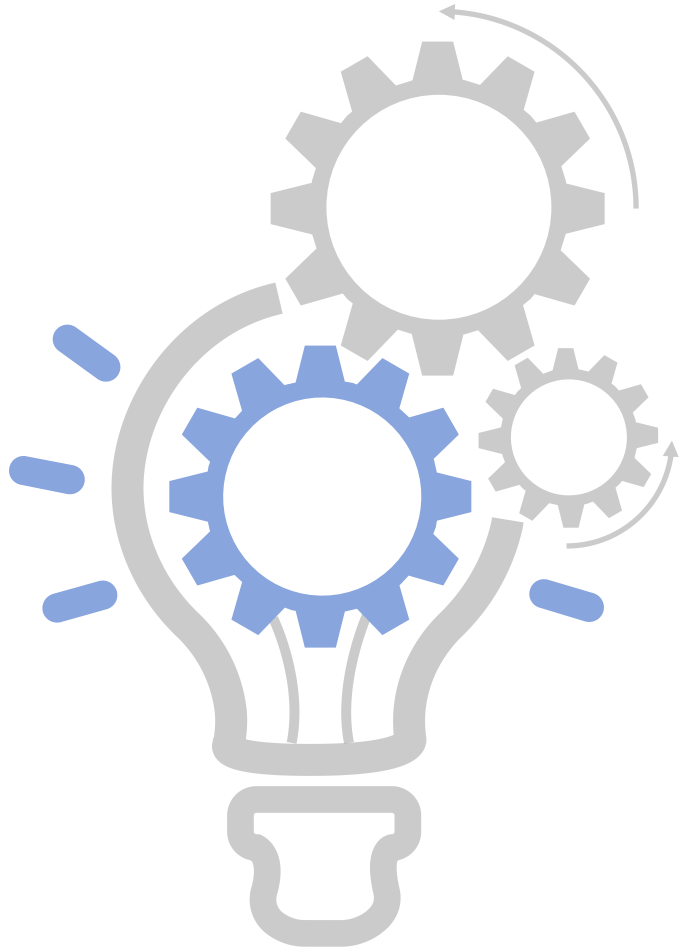
Team QuickHorse (AKA 快速牛马)

Shibo Zheng

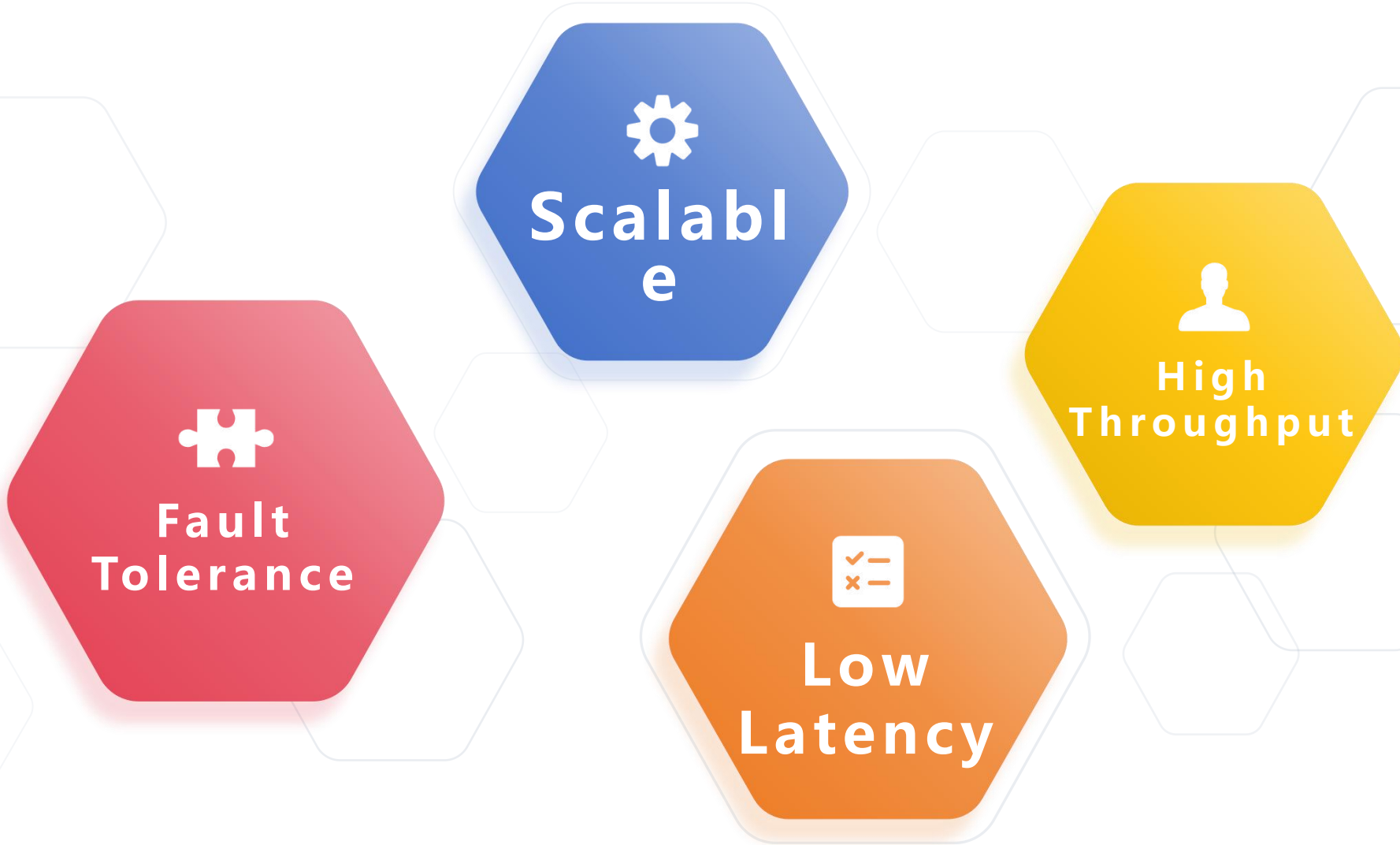
Ruiyi Li

Zhongzhe Liu

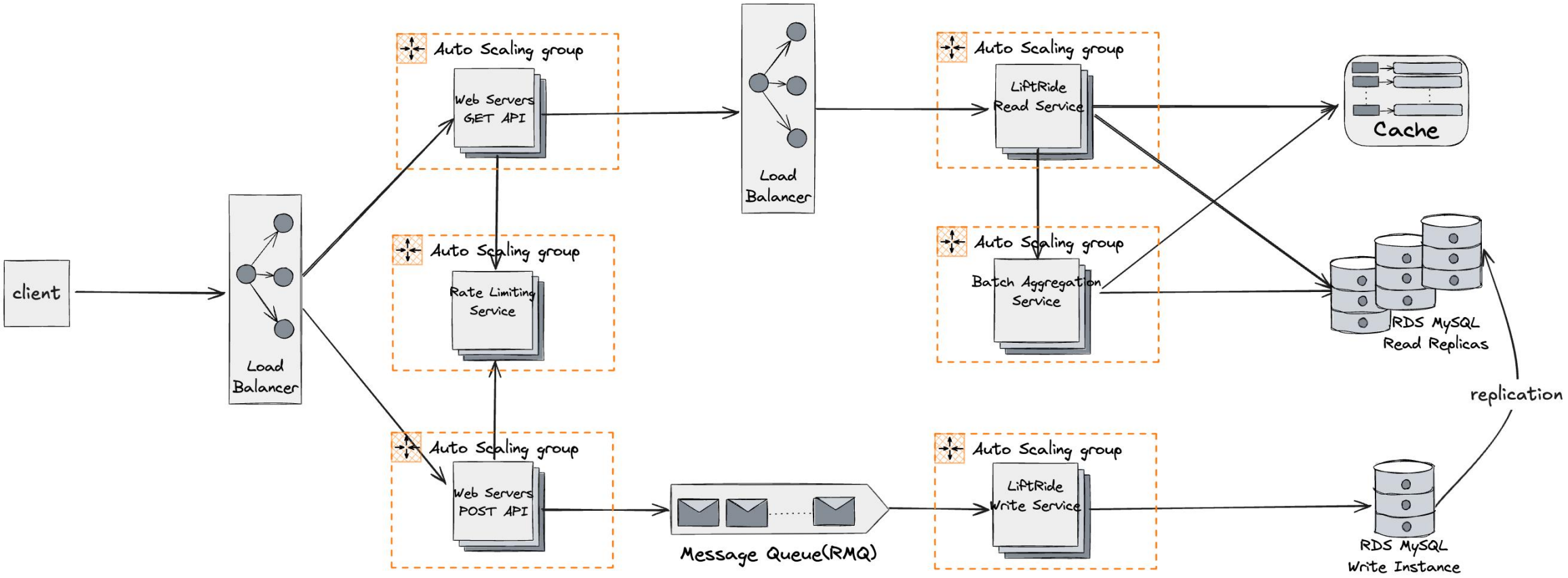
Han Yang



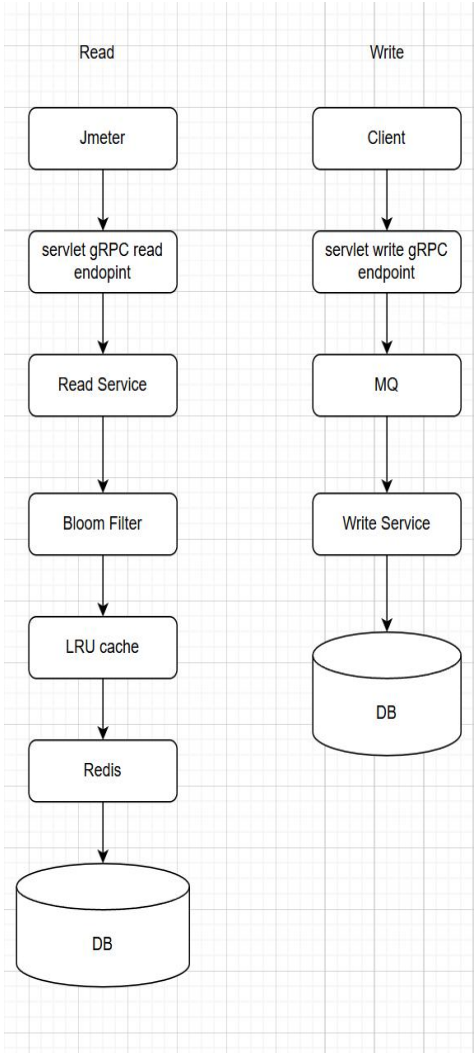
Goal

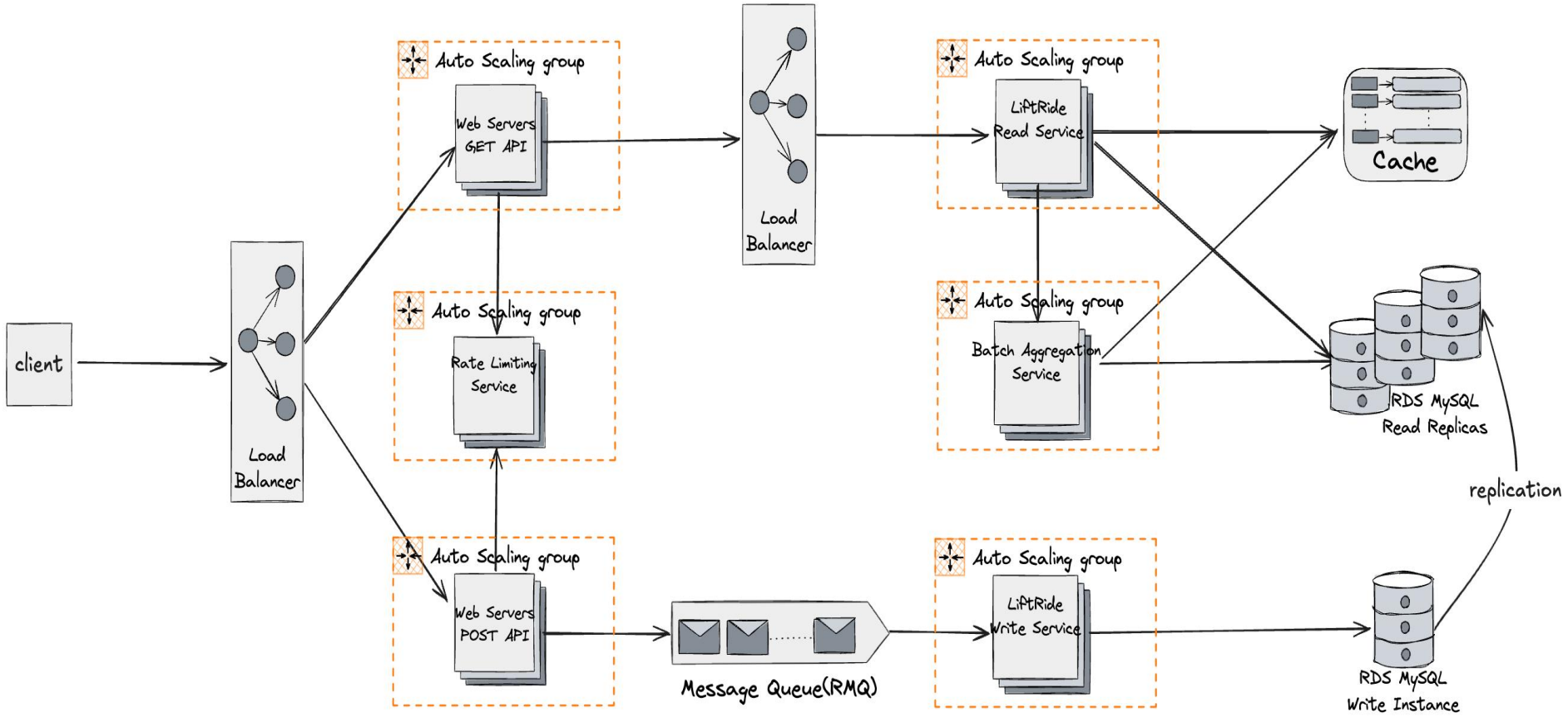


System Architecture

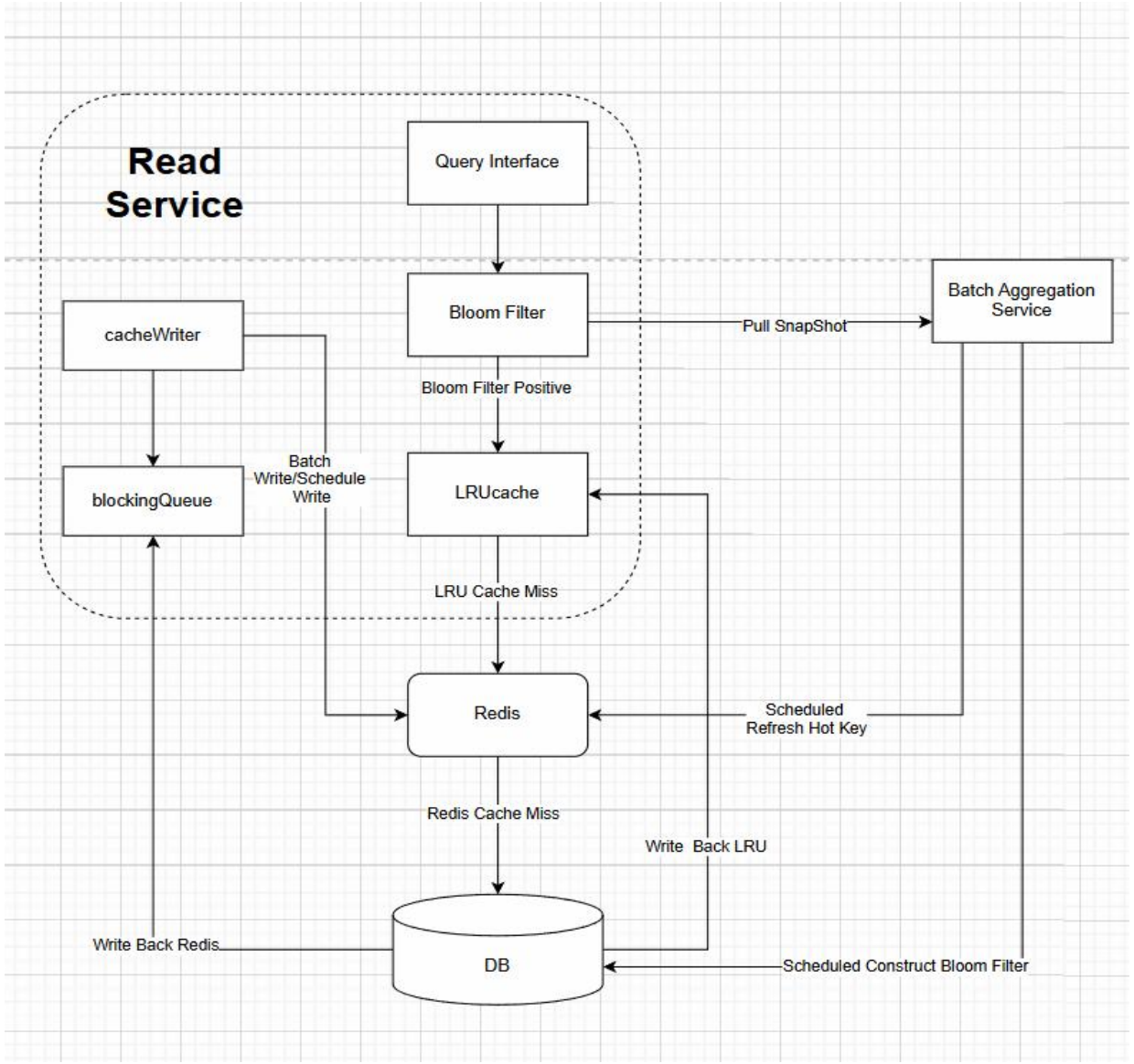


Work Flow


Read
Write
Jmeter
Client
servlet gRPC read endopint
servlet write gRPC endpoint
Read Service
MQ
Bloom Filter
Write Service
LRU cache
DB
Redis
DB


client
Load Balancer
Auto Scaling group
Web Servers GET API
Auto Scaling group
Rate Limiting Service
Auto Scaling group
Web Servers POST API
Load Balancer
Auto Scaling group
LiftRide Read Service
Auto Scaling group
Batch Aggregation Service
Auto Scaling group
LiftRide Write Service
Cache
RDS MySQL Read Replicas
replication
RDS MySQL Write Instance
Message Queue(RMQ)

Read Path

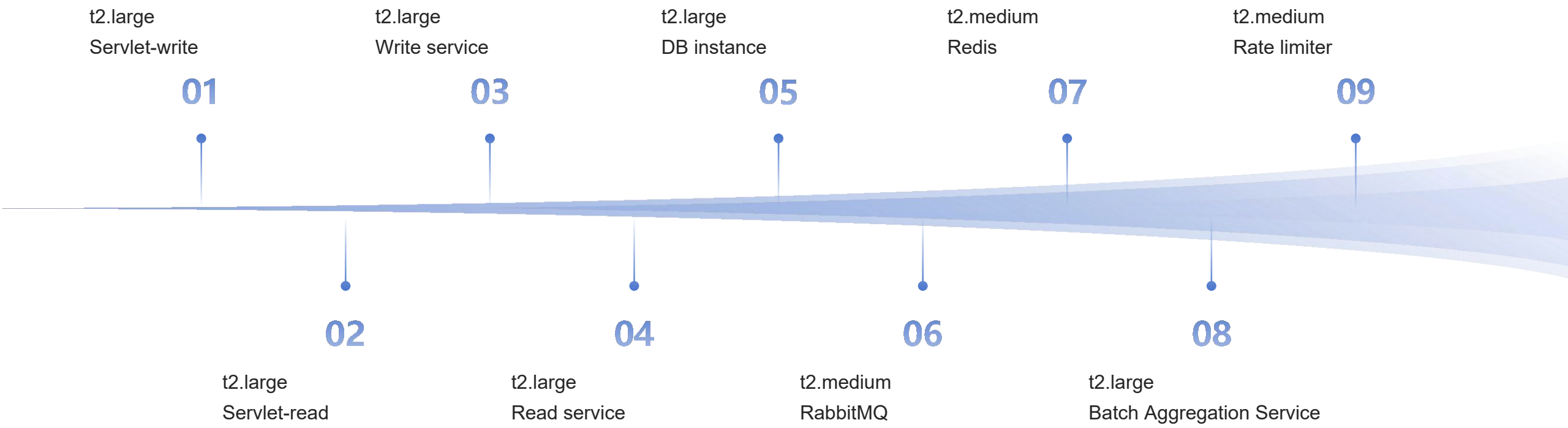


- Batch Aggregation Service:
  - Periodically aggregate data from DB
  - Based on strategy:
    - FULL → update Redis + Bloom
    - BLOOM_ONLY → update Bloom only
    - REFRESH_EXISTING_CACHE → update Redis only
    - Builds Guava Bloom Filters, exposes snapshot via gRPC

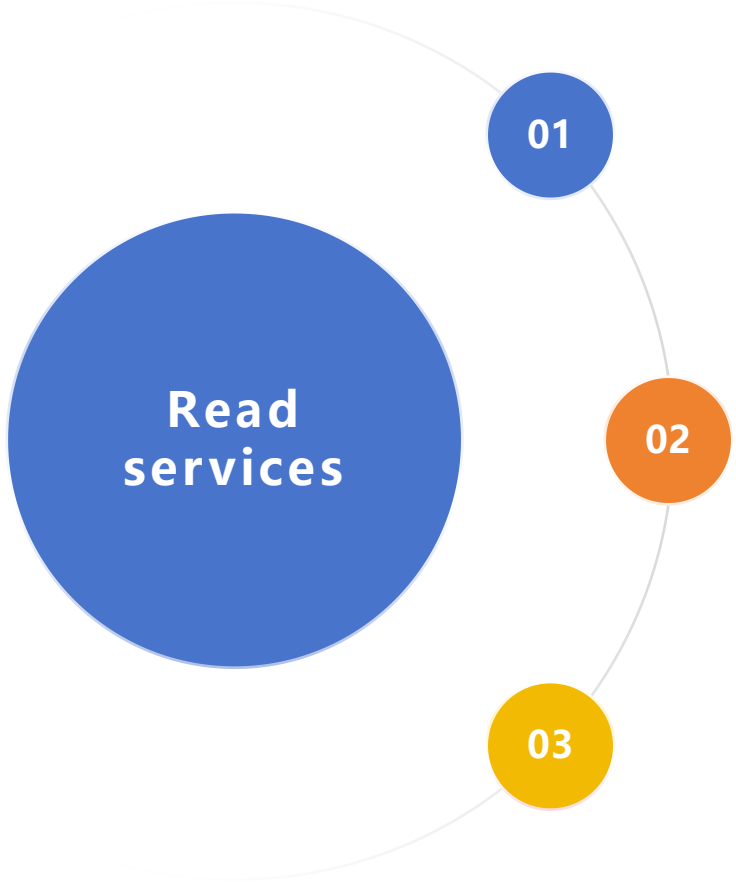
Microservice Component

1. Client & Jmeter
2. Write Servlet
3. Read Servlet
4. Ratelimiter
5. ServerWriteService
6. DB read service
7. Cache read service
8. Batch aggregation Service

Deployment on AWS



Data Model Design



GET/resorts/{resortID}/seasons/{seasonID}/day/{dayID}/skiers

Get Number of unique skiers at resort/season/day

GET/skiers/{resortID}/seasons/{seasonID}/days/{dayID}/skiers/{skierID}

Get total vertical for the skier for the specified sku day

GET/skiers/{skierID}/vertical

Get the total vertical for skier of specified resort, if not specified, return all season

Data Model Design

- Request Body: LiftRide (day, liftID), resort_id, season_id, day_id, skier_id
  1. Each resort can store multiple seasons data
  2. Each resort has multiple lifts
  3. Each skier can go to multiple resorts in any days in any seasons and take mutiple lifts

- Mapping Relation between entitles:
  1. Resort : Season → 1 to N
  2. Resort : Lift → 1 to N
  3. Skier: LiftRide → 1 to N
  4. LiftRide : Lift → N to 1

Trade Off between SQL and NO SQL

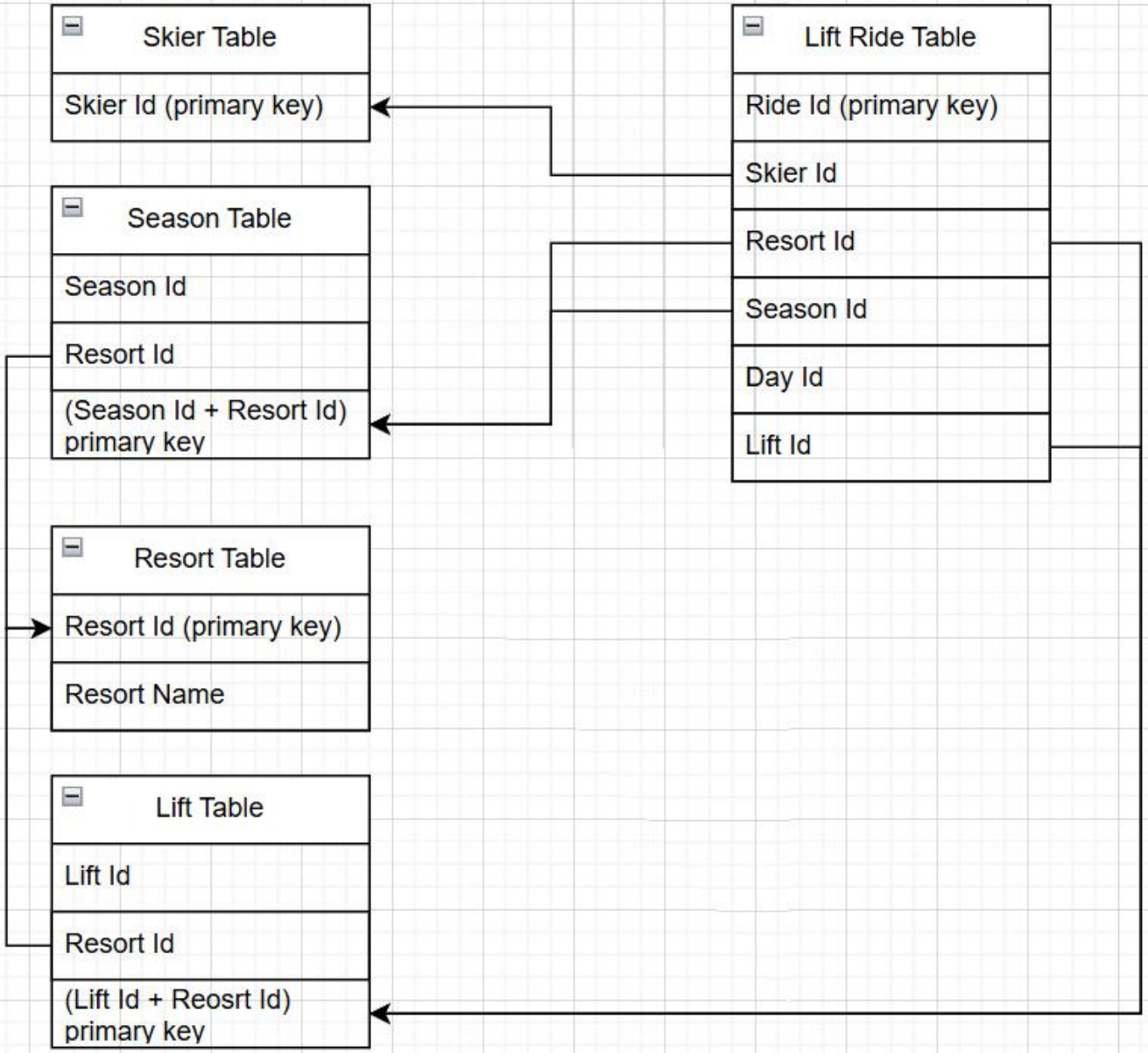
- Strong Data Integrity with Relationships.
  - Data model includes multiple interrelated entities.
  - Referential integrity is enforced through foreign key constraints.
  - NoSQL DB do not support joins or relational constraints, leading to data duplication or inconsistent references.
- Complex Querying Requirements (Aggregation)
  - Read operation required grouping, filtering, and joining across multiple tables

Trade Off between SQL and NO SQL

NOSQL database can do complex query by assigning partion key + sort key

But, not Scalable, not Efficient

Redundant data still exists



Database Schema

Hybrid Solution

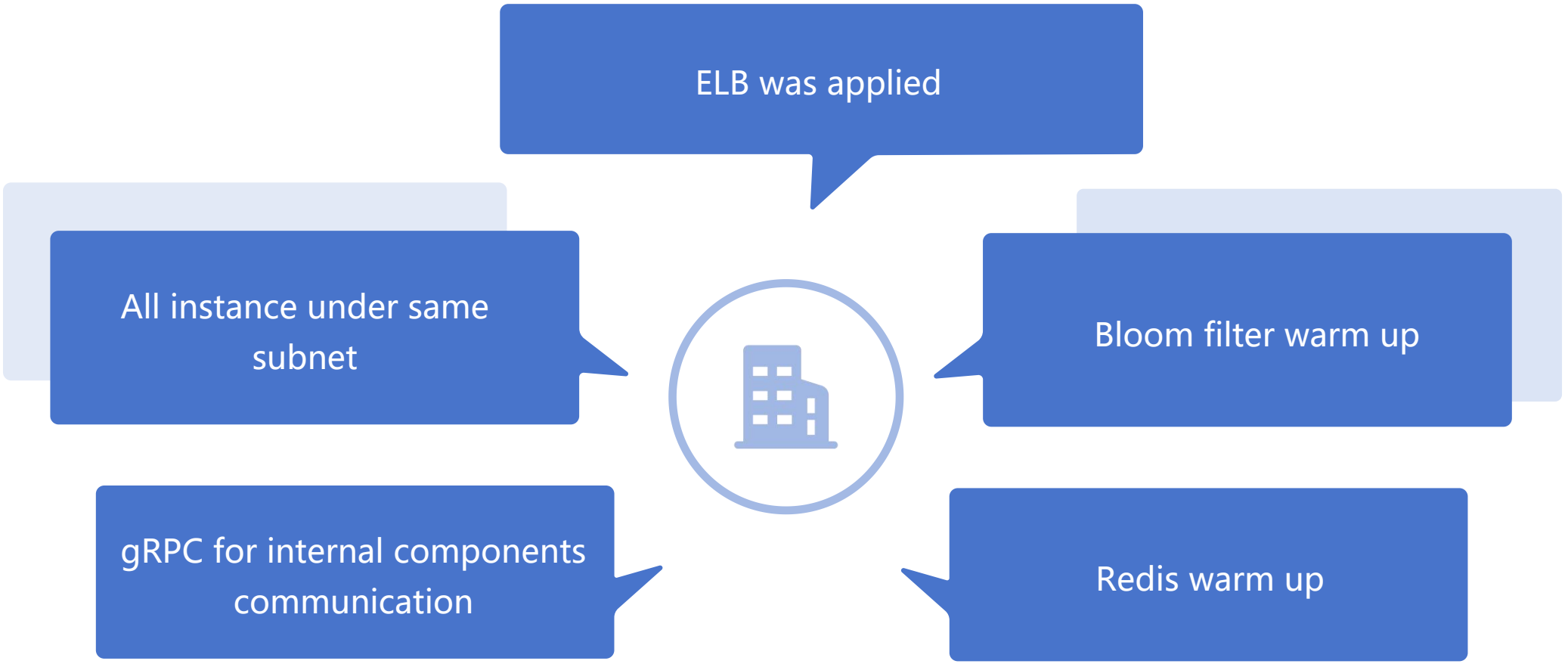
- **Multi-tierd cache strategy**
- In Memory LRU cache in read-service
- Redis as second layer cache service
- We also implmented a bloom filter to prevent “Invalid” request
- Why Bloom Filter:
    1. Guarantee no false negatives, but allows false positives.
    2. Light weight data structure
    3. Fast look up
    4. Prevent waste on computing & I/O

Some Optimization

- Decouple bloom filters with Redis
- **Before**: Each ReadService queried Bloom filters via RedisBloom
  - Bloom filters were initialized and stored in Redis
  - We do not want to too much access of Redis!
- **After**: Remove bloom filter from Redis, build and managed locally in memory
  - ReadServices periodically pull updated Bloom filter snapshots from BatchAggregation Service
  - ReadServices perform in-memory mightContain() checks without Redis calls
- **Why**
  - No Redis-Bloom Filter dependency → improved system stability
  - ReadServices become more fault-tolerant and loosely coupled

Some Optimization

Some Tricks


ELB was applied
All instance under same subnet
Bloom filter warm up
gRPC for internal components communication
Redis warm up

Tune The System

01 **Balance between availability and performance**

02 **We put all tunable parameters in config.properties file, easy to tune**

03 **Some major tunable parameters for writing**

- Number of queues & Channels
- Message fetch amount & write batch size & write flush interval
- Write Service thread number

04 **Some major tunable parameters for reading**

- DB max pool size
- Read Service thread number
- Aggregation cache interval

Key Metrics Monitered

01 Write throughput

02 Read throughput

03 P99 latency

04 LRU hit amount

05 Cache hit amount

06 DB hit amount

07 Bloom filter negative amount

Result

01

Write

- Throughput: 10585.371017254156
- Mean response time: 92.447995 ms
- Min response time: 39.0 ms
- Max response time: 837.0 ms
- P99 response time: 321.0 ms

02

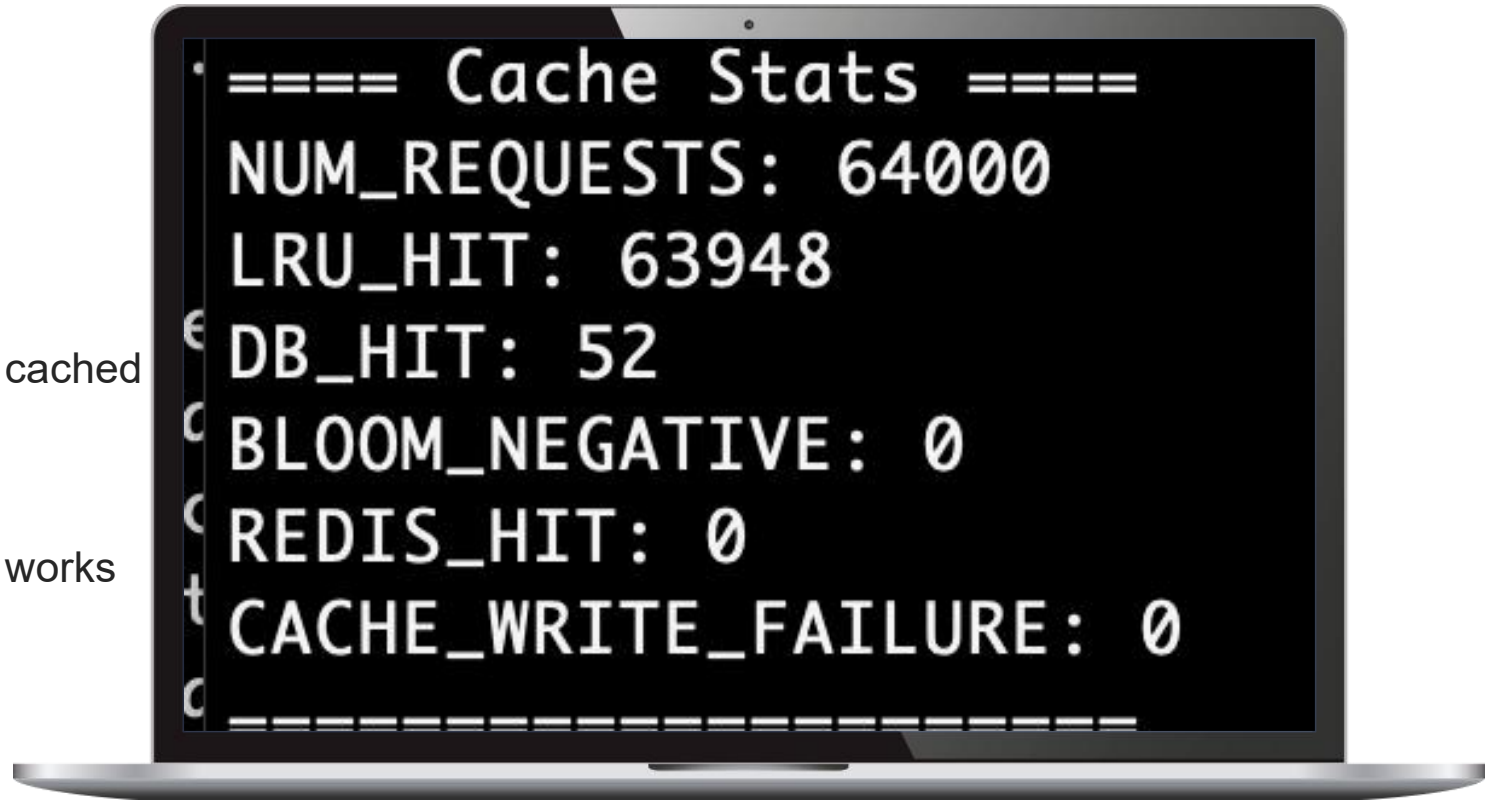
Read

- getAllVertical Throughput: 3795.52
- getSingleVertical Throughput: 4716.63
- getUniqueSkier Throughput: 4607.96

GET/resorts/{resortID}/seasons/{seasonID}/day/{dayID}/skiers

for getUniqueSkier api, most of read request were cached by LRU cache

10 resorts * 1 season * 3 days, total 30 keys, LRU works as expected!

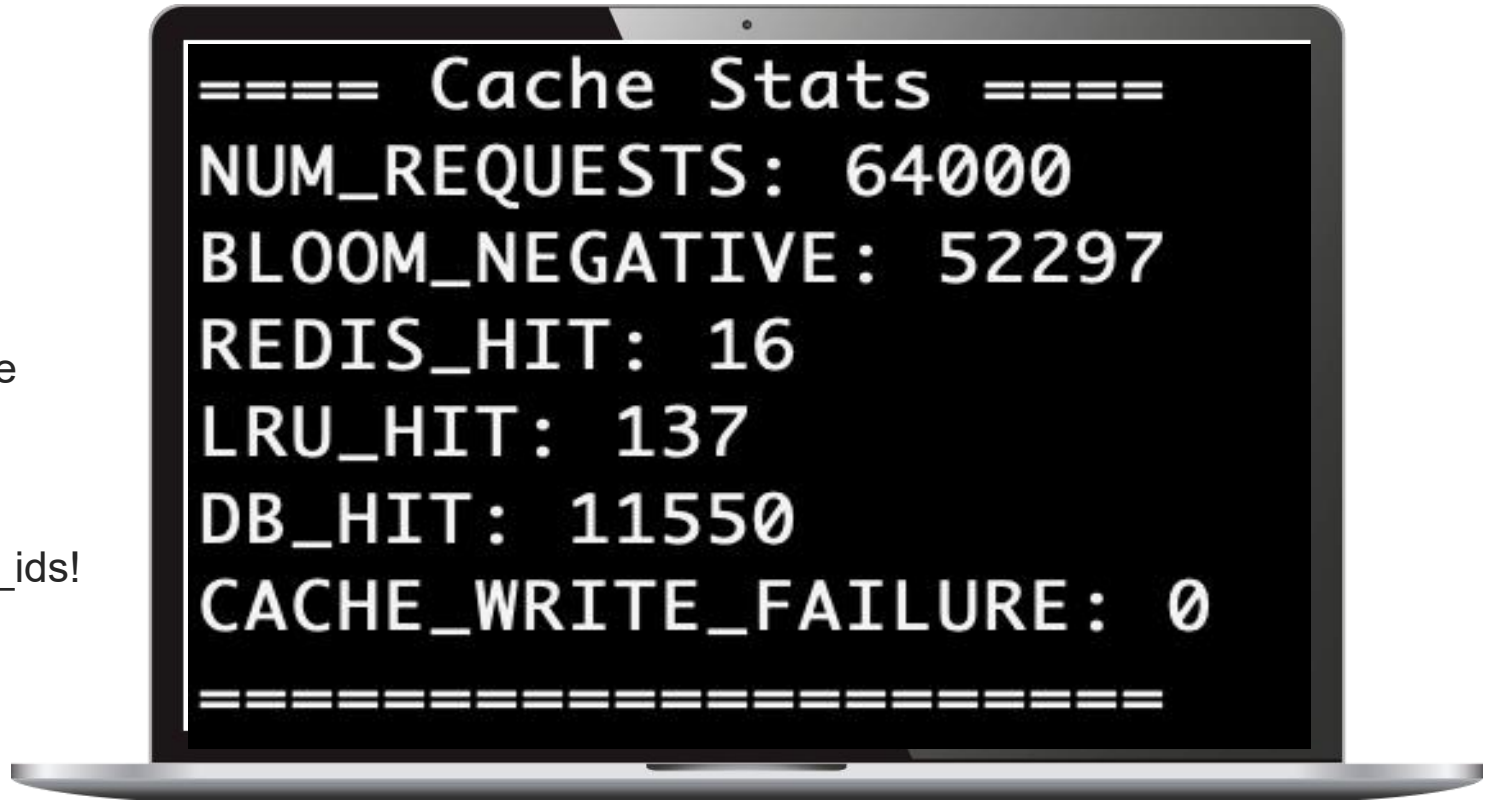


Why still 52 DB hit?!
This is because under concurrent condition, there is a delay in writing to Redis and LRU cache between different thread.

GET/skiers/{resortID}/seasons/{seasonID}/days/{dayID}/skiers/{skierID}

for getSingleVertical api, most of read request were filtered by bloom filters

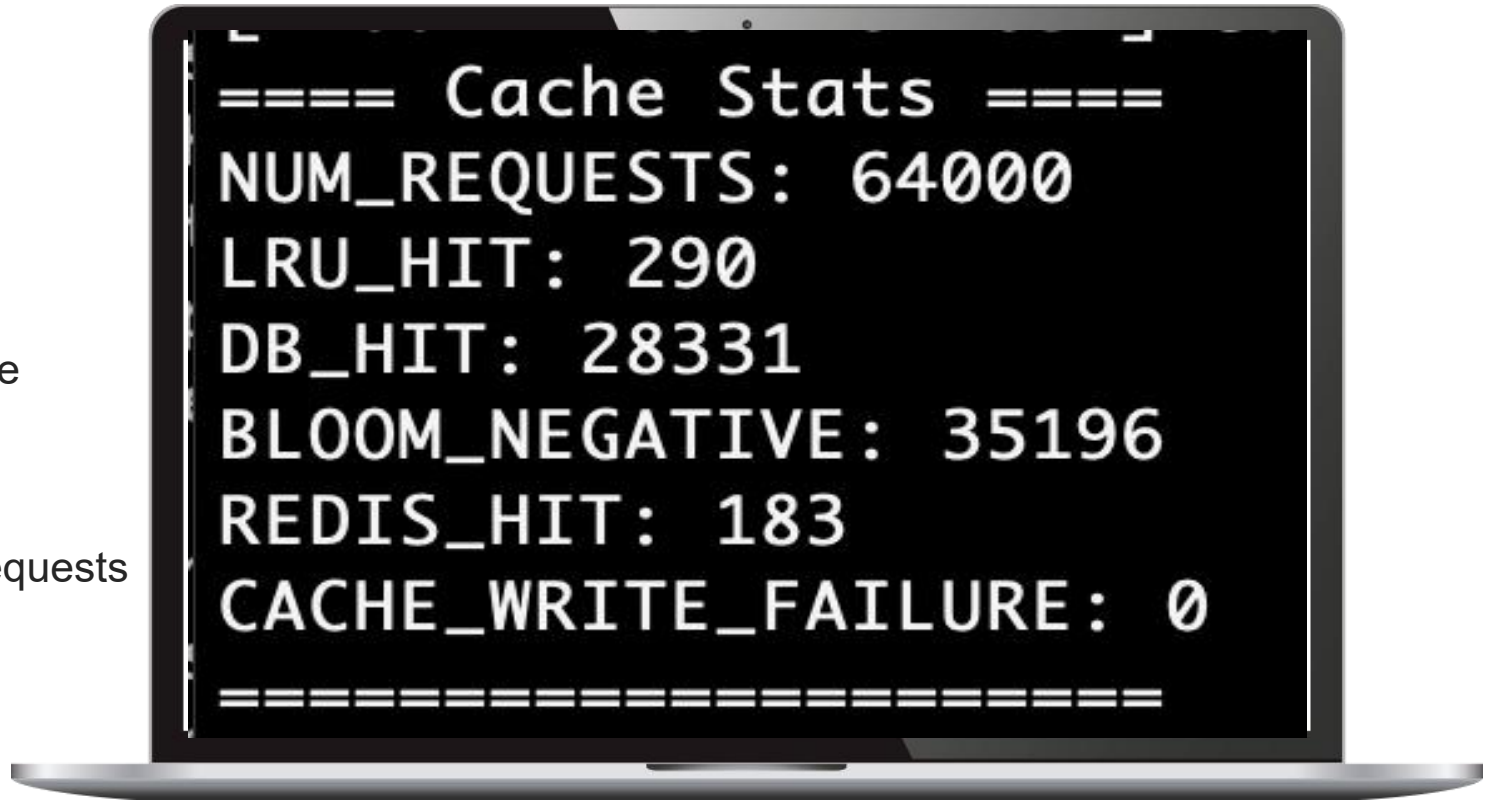
This make sense because too much random skier_ids!



GET/skiers/{skierID}/vertical

for getTotalVertical api, most of read request were filtered by bloom filters

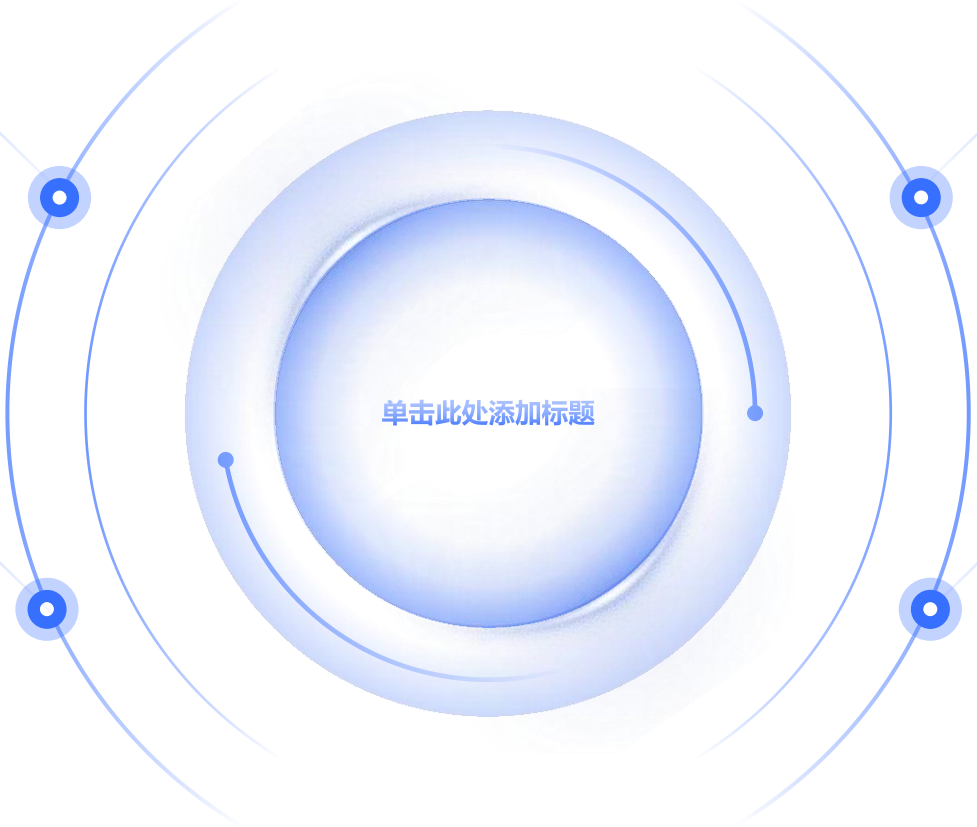
Less random skier_ids than last api, but most of requests still filtered by bloom filters, as expected



Overall

Design Trade-off:
Availability & Performance > Consistency

The system is designed to be eventually consistent



Stale data may exist during:

1. LRU cache refresh cycles
2. Redis cache write-back delays
3. Bloom filter update intervals

However, This trade-off gave us:

1. Faster read responses
2. Higher throughput under concurrent load
3. Better fault tolerance

Future Improvement

Distributed cache service (build a cluster)

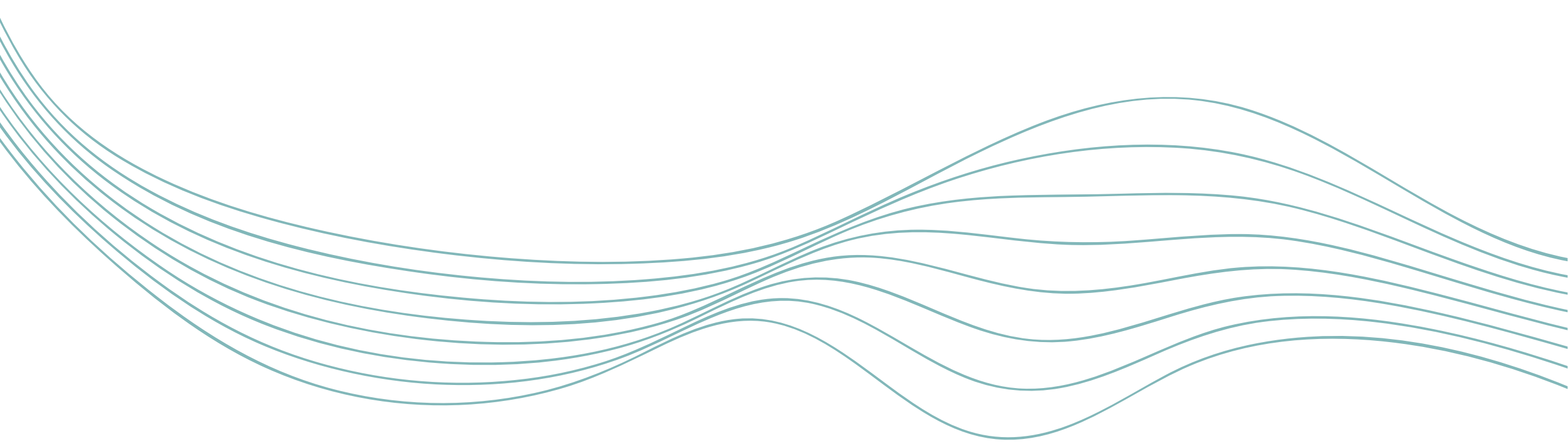
Hybrid solution of multiple DB

Enable automatic recovery and back up

Maybe tried more powerful machines

Write-Behind Strategy for Cache Updates, write goes to DB first, then asyn flush to Cache

Infra buidlings: observability, data warehouse & data lake, ETL pipeline etc.



Thanks for watching Q&A