Scalable Systems 1

Sasha

(Alexandr

RAEVSKIY)

Today's Agenda

Difference between DS/Algo and Scalable Systems

Problem solving framework recap

Online processing: Design Instagram clone

Online processing: Design Uber clone

Difference between DSA and System Design

{ik} | INTERVIEW KICKSTART

What changes when we move from DS/Algos to SysD?

In DSA, for a given computational problem

- Design a correct & efficient algorithm
- Code it up

Assumptions:

- Only one user
- Program will run only once
- Unbounded CPU and RAM to solve the problem

What changes when we move from DS/Algos to SysD?

In system design,

- Your DS/Algos program is now a service in System Design.
- The users of the service known as **clients**
  - May want to execute program remotely over a network.
  - Will need to *request* your **server** machine to run the program using the inputs they send over the network.
- Your server machine
  - Needs to send the output to of a request as a *response* over the network.
  - Needs to always be actively listening for requests.
- Your program needs to be **scalable** if it can handle a huge growth in the load (number of concurrent requests) without sacrificing the *response time*.

  The standalone program is converted into an **online service** and we will see how to build such an online service.

Scalability and its types

Scalability is property of a system to handle growing amount of traffic.

The scaled program with client-server will act as an online service

In case the program is scaled offline, it is known as batch processing

When the input is not fixed and comes as a stream, it is known as stream processing

Top-Down steps for System Design in an interview

{ik} | INTERVIEW KICKSTART

- Functional Requirements
- Technical Requirements
- Microservices
- Logical diagram
- Deep dive into MSvc (x1-x2 ms)
  - App, storage, cache tiers
  - Scale app, storage, cache tiers
- Recap

Step 1: Gather all requirements of the system

1. Gather functional requirements.
   a. This is the detailed problem statement.
   b. Describe user's view of the system in plain English
   c. Shows that you can communicate and unpack an open-ended problem.
   d. Ask clarifying questions.

2. Design Constraints / Scalability requirements
   a. Number of users, transactions per second etc
   b. The interviewer may give them or may ask the candidate.
   c. Can be collected at a later step as well.

Step 2: Defining Microservices of the System

A Microservice

- Is an *independently deployable* service modeled around a business domain.
- *Loosely coupled*, you can change one microservice and deploy it into production without affecting other microservices.
- Each microservice is a *black box to other* microservices.
- Small and focused on *doing one thing well*: Owned by a single (2 pizza) team
- Communicate with each other using language-independent network APIs (e.g. REST with HTTP)

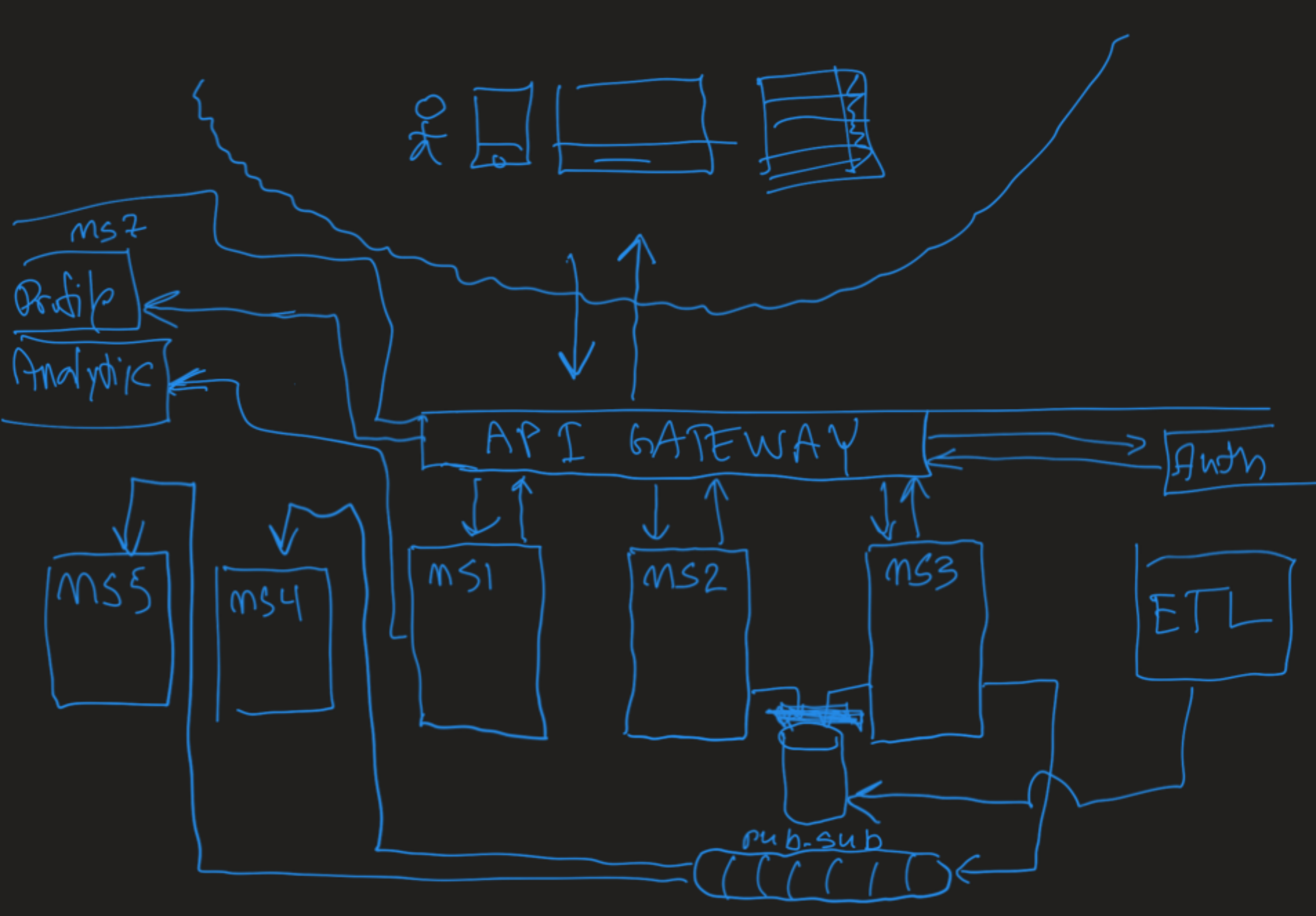
Step 2: Defining Microservices of the System

Bucketize the functional requirements into microservices.

1. Cluster the requirements such that each cluster can be handled by a different team.
2. Rule of thumb: If the APIs and data models for two functional requirements do not look the same, put them in different buckets.
3. This is subjective, depends on the individual
4. After doing this, you will know whether it is a depth-oriented problem (~1 microservice) or breadth-oriented problem (many microservices)

Step 3: Design the Logical Architecture

1. The logical architecture is a block diagram with one block for each microservice
2. Draw and explain the data/logic flow between them
3. Rules of thumb:
   a. If the client (user or microservice) is waiting for response from the microservice, use HTTP/REST APIs
   b. If the client microservice does not expect an immediate response from the server microservice, use a message queue (pub-sub) which is its own microservice.
   c. If data transfer is offline, you may use batched ETL (Extract Transform Load) jobs


ms7
Profile
Analytic
API GATEWAY
Auth
ms5
ms4
ms1
ms2
ms3
ETL
pub-sub

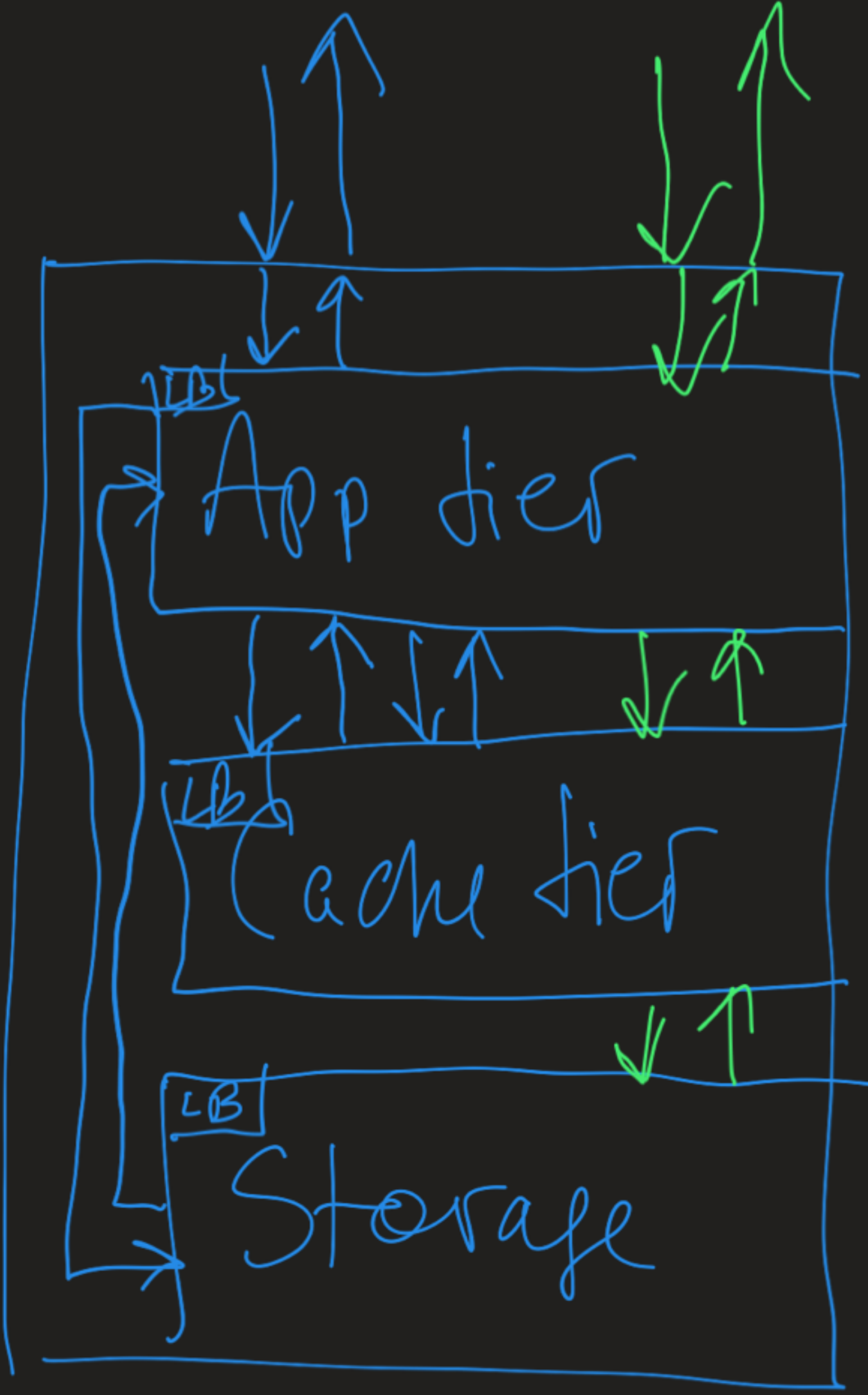
Step 4: Deep dive into each microservice

High-level microservice architecture:

- In general each microservice can have, upto three tiers:
  - Backend tier with business logic (App server) -- this is where your single-machine code conceptually goes
  - Cache tier for faster response time (Cache) - data here is a subset of the database tier, but can be stored in any format/data structure
  - Storage tier for data storage and retrieval (Database): store in row-major/col-major/files (for movies and other unstructured formats)

Note: We don't worry about front-end tier

ms:



Step 4: Deep dive into each microservice

1. If there are many microservices, discuss with the interviewer which ones to focus on.
2. Budget your time per microservice, since you have only a constant amount of time
3. If there is only 1 microservice, there might be a coding part to the interview.
4. For each microservice
    a. Solve each tier logically (as if solving for a single server)
    b. Identify Data Model (what data needs to be stored) to match the functional requirements
    c. Discuss how data will be stored in storage and cache tiers
    d. Propose APIs to match the functional requirements
    e. Propose workflow/algo for the the APIs in each tier

Note: Scale is not in the picture yet.

Step 4: Deep dive into each microservice

- Propose flow across tiers within the microservice
    - Most of technical design interviews have meat in this phase
    - Non-deterministic, every candidate will come with his or her own proposal
    - Changes from problem to problem
    - This constitutes the ‘most thinking’ portion of the interview
    - Most chances to flunk here unless candidate is prepared

Step 5: Identify the need for scale

In each microservice, identify the need for scale

- A deterministic set of reasons can be posed across all interviews
  - Need to scale for throughput (CPU/IO)
  - Need to scale for API parallelization
  - Need to remove hotspots
  - Availability and Geo-distribution
  - Need to scale for storage (storage and cache tiers)

- The constraints or numbers change from problem to problem
- Solve algebraically first and then put numbers. Algebraic formula is same for all problems
- If phase 1 takes more time, don't spend time on calculations/estimations here unless asked for

Scaling Options: Vertical scaling

- Easiest way to scale to higher load
- One powerful machine (with lots of CPUs, RAM chips and disks), at a single location
- Cost eventually grows faster than linearly.
- Has a ceiling

Performance of one machine depends on:
- Number of CPU cores, speed of each CPU core, competing processes on the same CPU, amount of RAM, type of disk (SSD vs Magnetic Disk), number of disks (parallel access), network connectivity, GPUs.

Scaling up easy with cloud computing (up to 96 cores)

Scaling Options: Horizontal scaling

Add more machines. But need to change your program and coordinate tasks on multiple machines across the network.

If the “web application” threads are stateless, can push the state into a database.

This allows us to add more servers and a proxy (load balancer and/or cache).

A few shortcomings:

1. Complex distributed systems (managing consistency, availability, debugging)
2. If not architected properly, can result in fragmented servers, improper division of workload

Step 6: Propose the distributed system

Scale the tier and propose the distributed system

- This is the most deterministic portion of the solution
- Steps
  - Draw a generic distributed architecture per tier
  - If app server tier and stateless, just round robin requests
  - If cache or storage tier
    - Partition the data into *shards* or buckets to suit requirements of scale (changes from problem to problem)
    - Propose algorithm to place shards in servers (*consistent hashing*) (same across all problems)
    - Explain how APIs work in this sharded setting (problem specific)
    - Propose *replication* (same across all problems)
    - Propose CP or AP (algorithms for CAP theorem do not change from problem to problem)
- This is how each microservice looks within a single data center, either on-premise or VPC in cloud

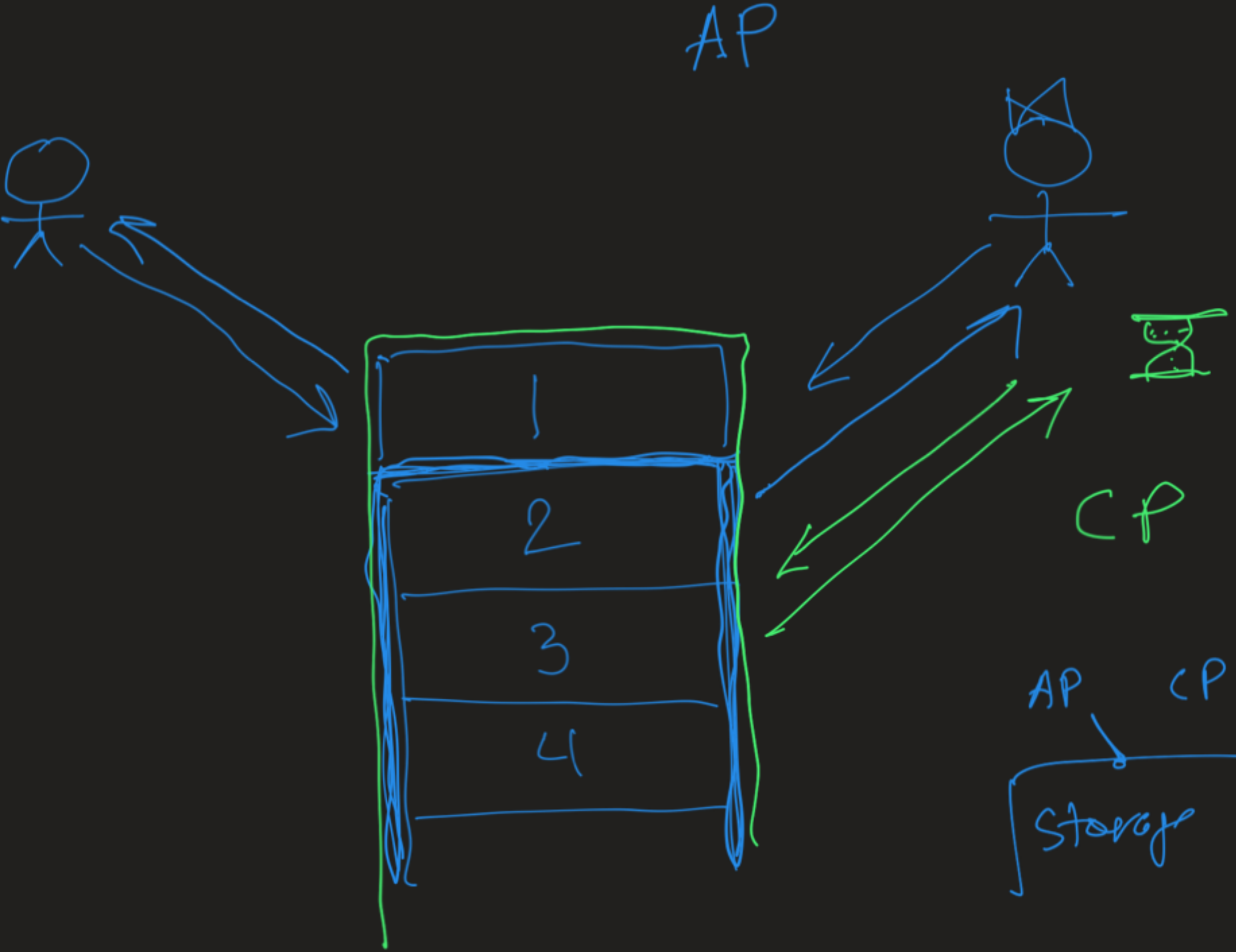
CAP theorem

Consistent

Available

Partitioned

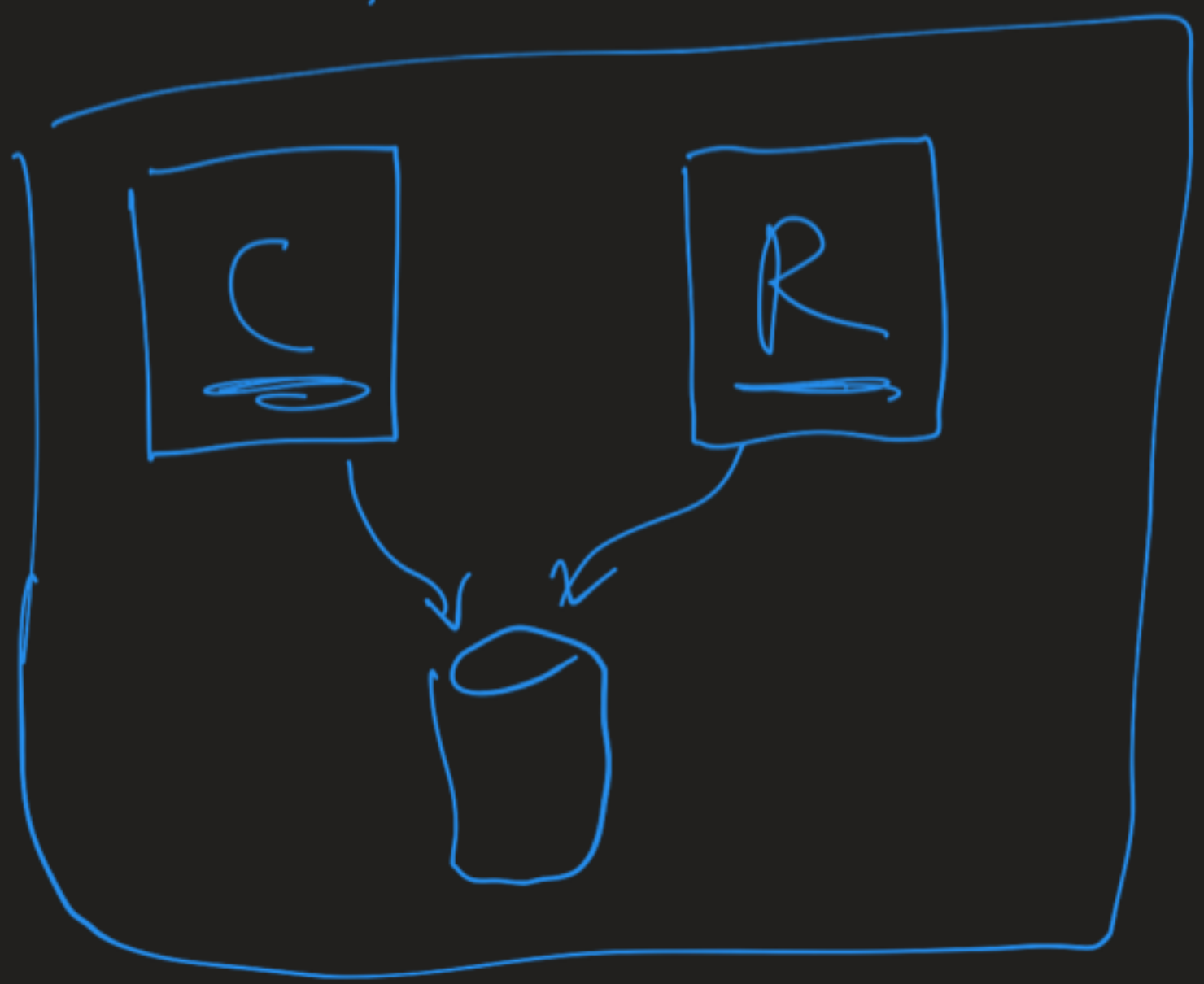
AP



CQRS

- Logic differs
- Scale

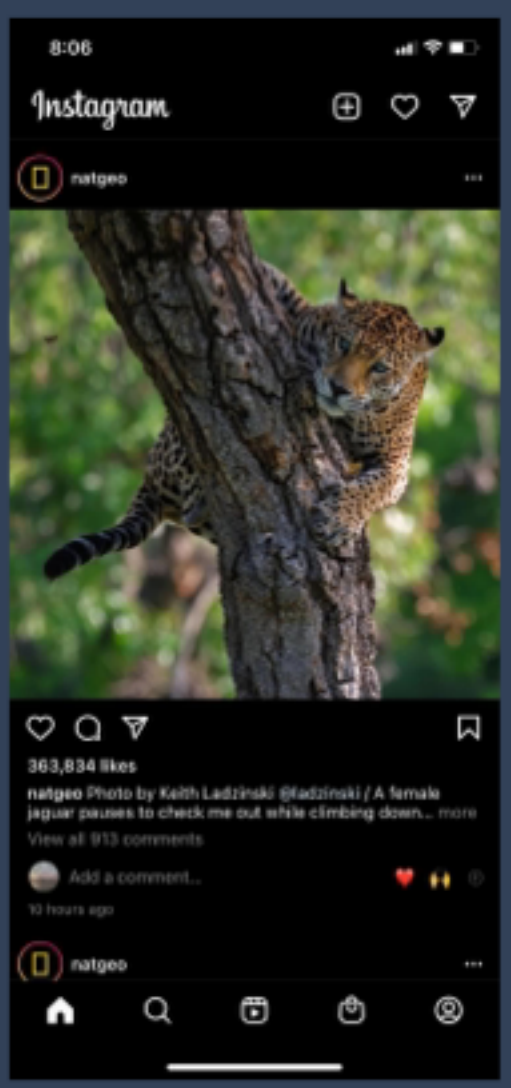
CRUD

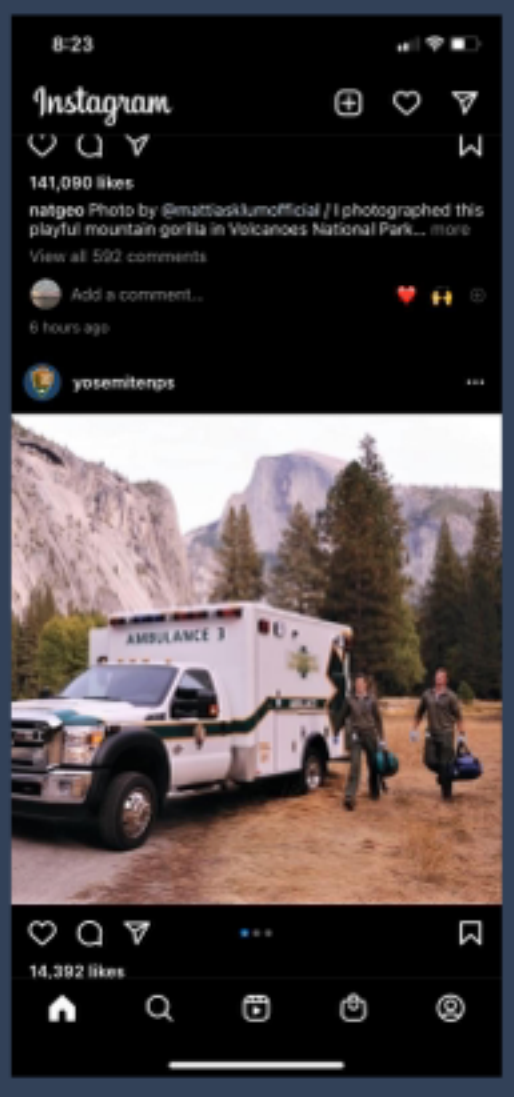


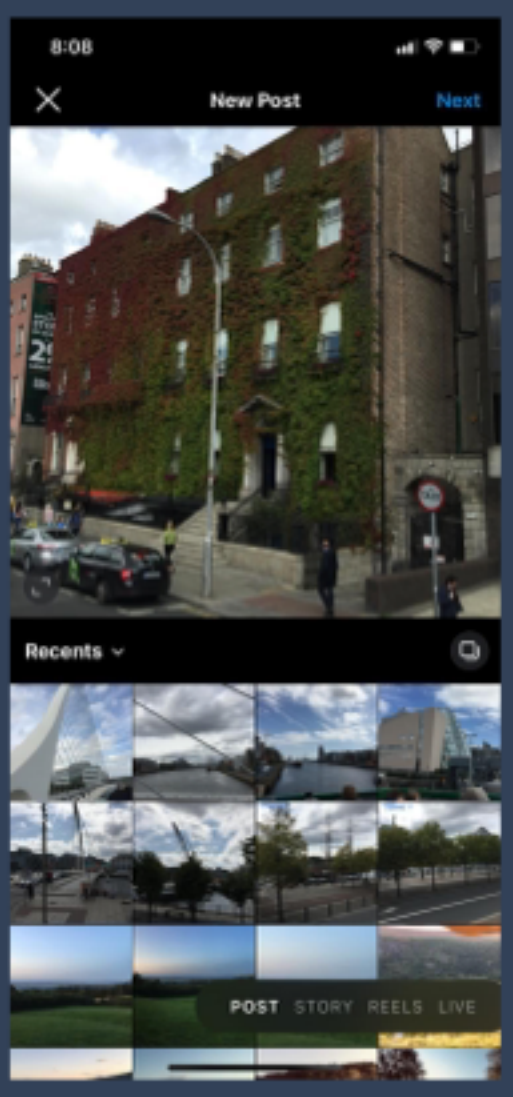
Problem 1

Online processing: Design Instagram clone

Instagram Application

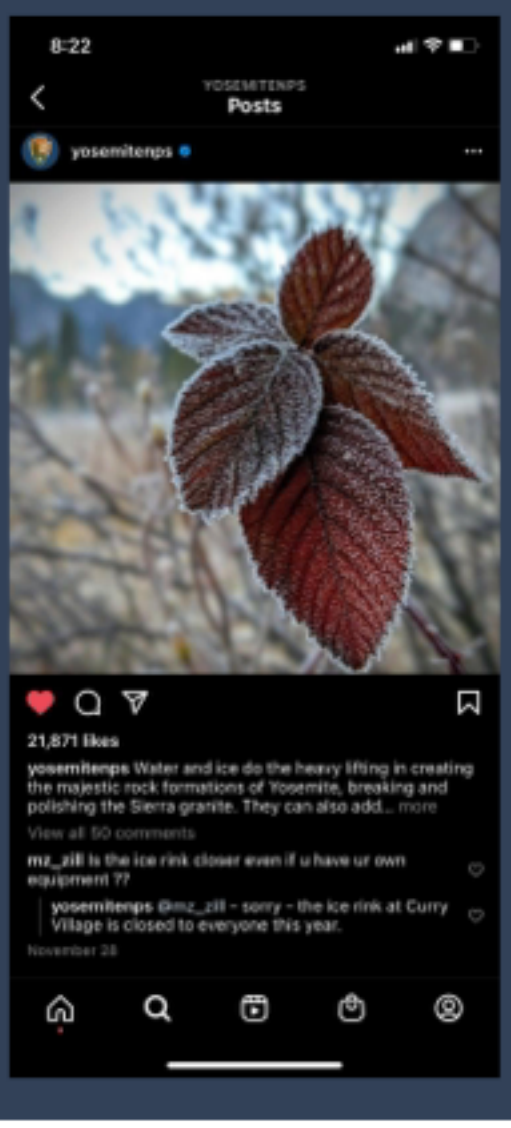






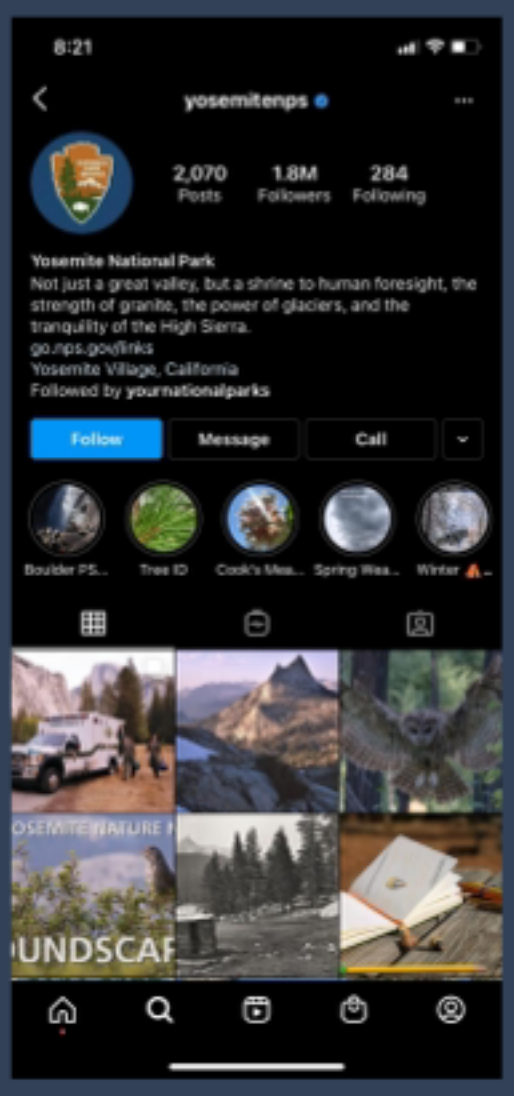
{ik} INTERVIEW KICKSTART

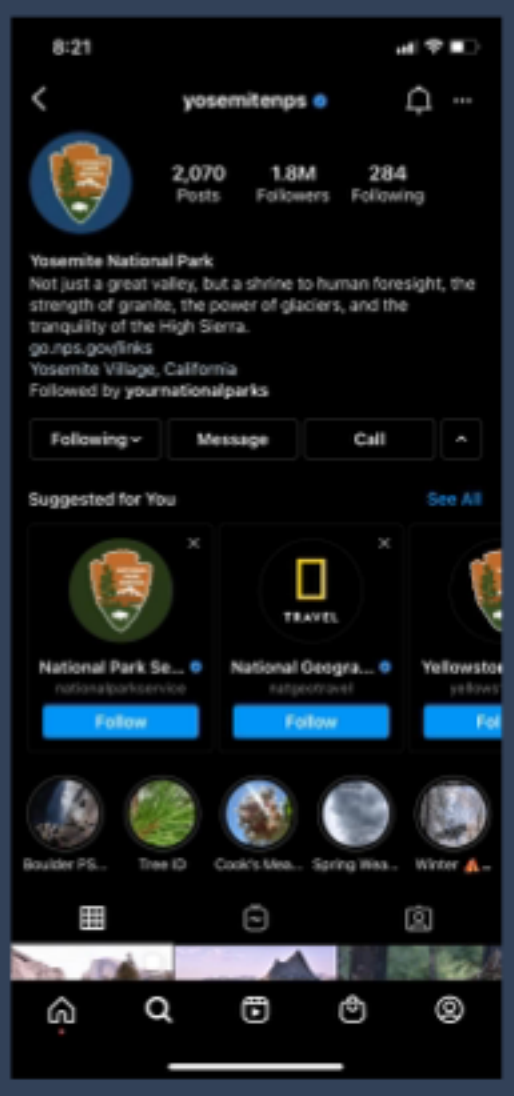
Instagram Application



{ik} | INTERVIEW KICKSTART

Instagram Application



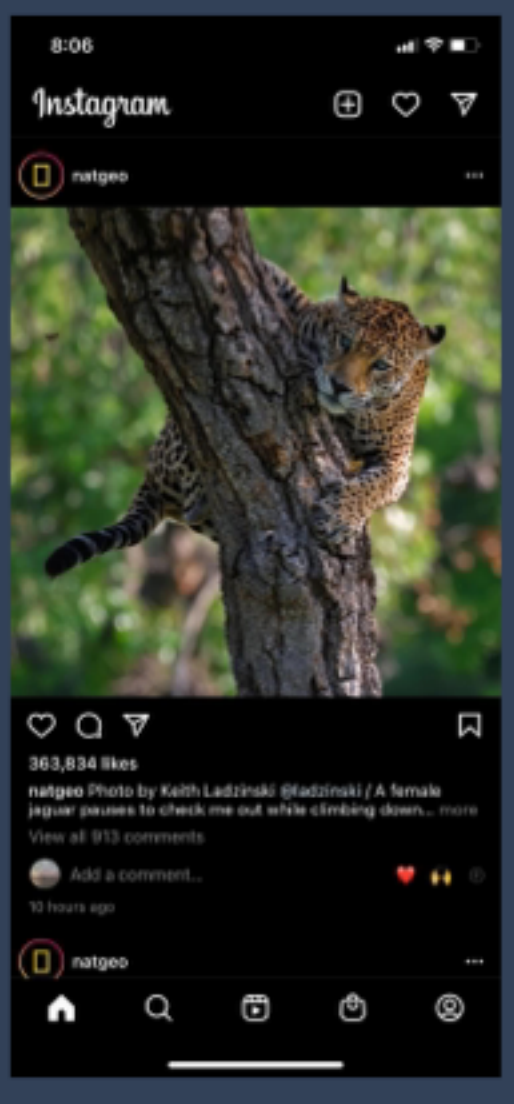


{ik} | INTERVIEW KICKSTART

Step 1: Gathering Requirements

Functional Requirements:

1. View, upload photos.
2. Like, comment on photos.
3. Search for photos based on tags or titles.
4. Follow/unfollow other users.
5. Generate a news feed, based on who the user follows.



Step 1: Gathering Requirements

Non Functional Requirements

- Service needs to be highly available (prefer availability over consistency).
- Acceptable latency of X milliseconds to generate the news feed, Y milliseconds to view a photo.
- The systems needs to be highly reliable (no lost photos)

1. Total users ~= 1 billion
2. Daily active users ~= 500 million
3. New photos per second ~= 1000
4. Photo views per second ~= 20000
5. Average photo file size ~= 200kb

Identify APIs according to Functional Requirements

1. Users should be able to view, upload and download photos.
   a. viewPhoto(photo_id, user_id)
   b. viewPhotos(user_id)
   c. postPhoto(user_id, photo, location, time_stamp, title, tags[])

2. Users should be able to like, comment on photos.
   a. likePhoto(photo_id, user_id)
   b. postComment(photo_id, user_id, time_stamp, comment)

3. Users should be able to follow/unfollow other users.
   a. follow(follower_id, followee_id, time_stamp)
   b. unfollow(follower_id, followee_id, time_stamp)

APIs according to Functional Requirements

1. System should generate a news feed for the user, based on who the user follows.
   a. generateFeed(user_id, time_stamp, batch_size, batch_range[])

2. Users should be able to search for accounts or photos based on tags or keywords.
   a. searchAccounts(accName, user_id)
   b. searchPhoto(user_id, tags[]/keyword)

3. User Authentication, account creation. (out of scope)
4. Live/Stories (out of scope)

How to organize the information in memory?

1. Fetch a photo given a photo id
2. Fetch User's information
3. Likes, Comments
4. Follow/Unfollow a user

What key do you need?

How do you translate that to a DB Model?

Data Model according to Functional Requirements

1. Users should be able to view, upload and download photos.
   a. PhotoTable: Primary Key = PhotoId(int64).
      Other columns = UserId(int64), description(varchar 64), ImgStorePath(varchar 64), ThumbnailPath(varchar 64), PhotoLatitude(int), PhotoLongitude(int), TimeStamp(datetime)
   b. UserTable: Primary Key = userId(int64)
      Other columns = name(varchar 32), email(varchar 32), dob(datetime), zipcode(int), createTime(datetime), lastLogin(datetime)

2. Users should be able to like, comment on photos.
   a. LikesTable columns: UserId(int64), PhotoId(int64), TimeStamp(datetime)
      Primary Key = UserId + PhotoId
   b. CommentsTable columns: PhotoId(int64), UserId(int64), Comment(varchar 128) TimeStamp(datetime)
      Primary Key = PhotoId + UserId + TimeStamp

Data Model according to Functional Requirements

1. Users should be able to search for photos based on tags or accounts based on name.
   a. Search on account can use UserTable, indexed on name.
   b. Search on tags: Reverse index on tags->photo id (part of offline system)

2. Users should be able to follow/unfollow other users.
   a. FollowTable columns: FollowerId(int64), FolloweeId(int64), TimeStamp(datetime)
      Primary Key = FollowerId + FolloweeId

3. News Feed generation:
   a. UserNewsFeed table: Primary Key = FeedItemId(int64).
      Other columns = UserId(int64), PhotoId(int64), PhotoOwnerID(int64), Description(varchar 64), ImgStoreOrCDNPath(varchar 256), FeedCreationTimestamp(datetime), NumLikes(int32), NumComments(int32))

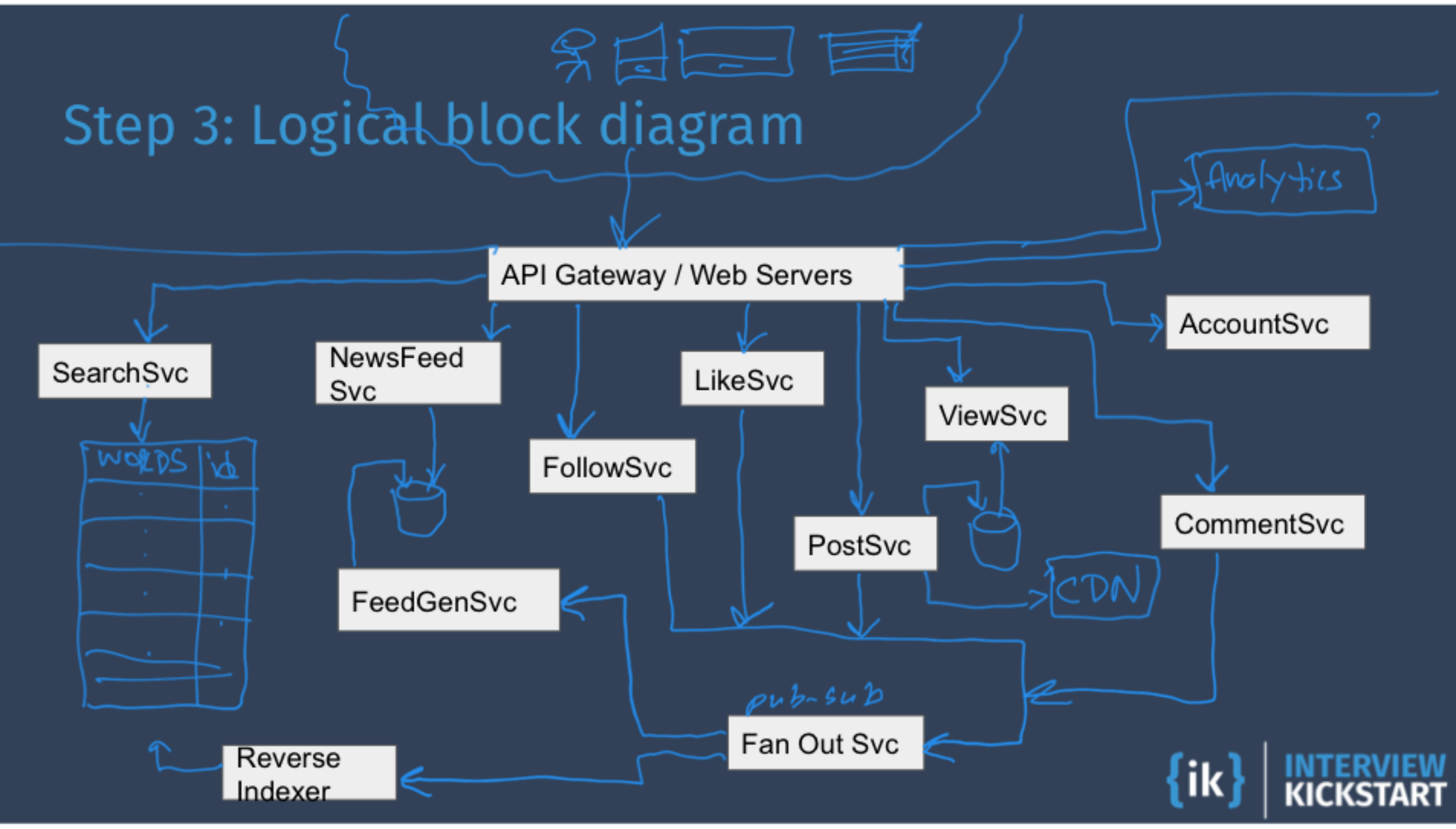
- Microservices:
  - View Svc ✓
  - Upload Svc ✓
  - Generate News feed ✓
  - Comment Svc ✓
  - Read Write Likes ~
  - ~~Edit Svc~~
  - Tag Svc ✓
  - Search Svc ✓
  - User Profile Svc ✓
  - Follow Svc ✓
  - ~~Live Stream Svc~~
  - Add Delete Follower Svc ~
  - ~~Advertising Svc~~
  - ~~Share Svc~~
  - ~~Admin Svc~~
  - ~~Photo Collage Svc~~
  - Likes Svc ✓
  - Analytics Svc ~
  - ~~Suggest User Svc~~

Step 2: Defining Microservices

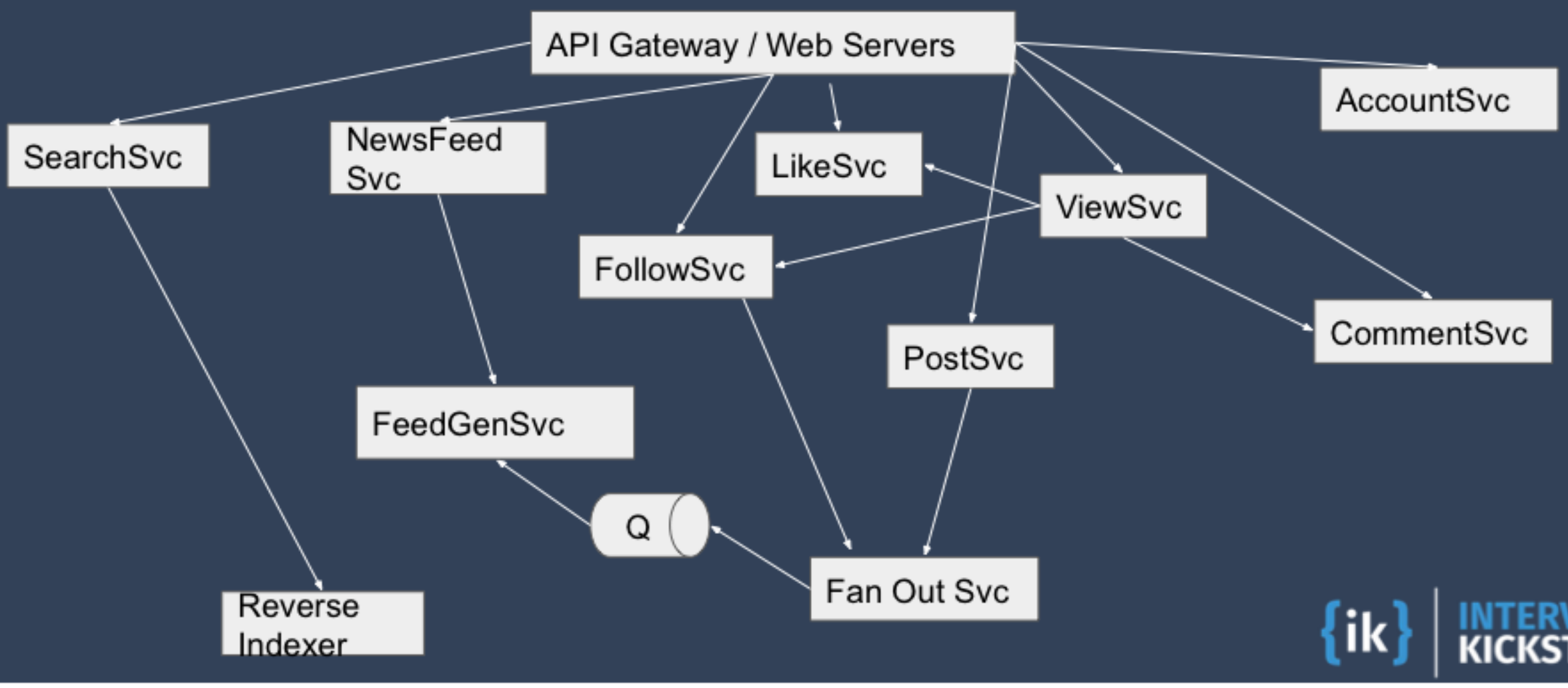
- ViewService
- PostService
- LikeService
- FollowService
- CommentService
- NewsFeedService
- *FeedGenerationService*
- SearchService
- AccountService

Breadth Oriented Problem

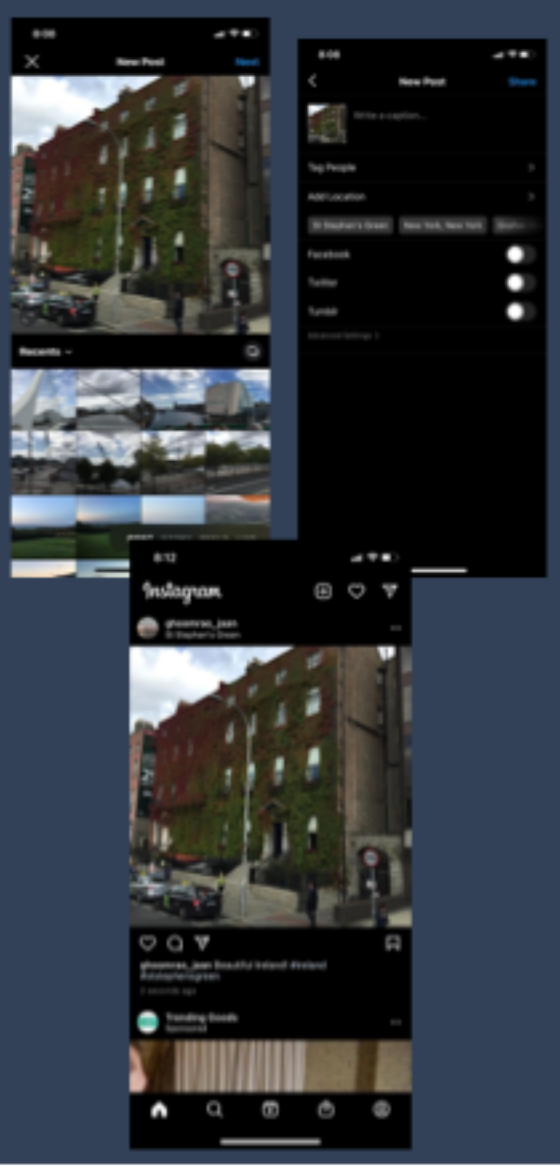
Step 3: Logical block diagram



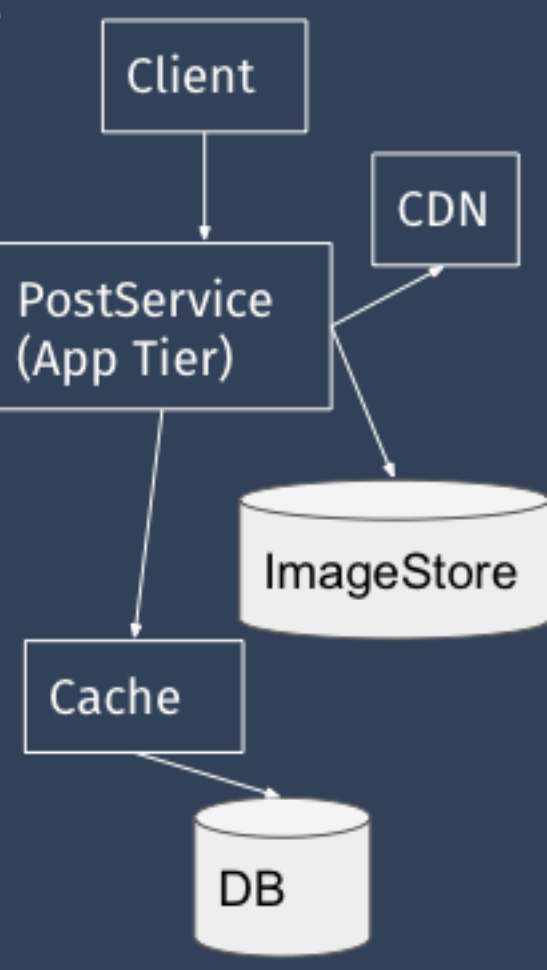
Step 3: Logical block diagram



Step 4: PostService



- Tiers: App server tier, Cache or In-memory tier, Storage tier
- Data Model: K-V pair
  - K: PhotoID: V: desc, img store path, other properties
- How to store in Cache tier - Hashmap
  - To store recent photos by celebs
  - K: celebID, V: a list of recent photos
- How to store in Storage tier - as a row oriented K-V store
- APIs
  - postPhoto(userID, location, time_stamp, ...)
- Algorithm in APIs
  - Generate photo id
  - Create(K.V) - CRUD
- Flow of APIs
  - Phone app to convert photo in proper resolution and format
  - Stores the img in img store, create meta-data rec in db
  - Update the celeb cache
  - Async fanout to update news feed, upload to cdn, gen reverse idx



Step 5: Recall scaling up the solution

- For each microservice
  - Check whether each tier needs to scale
  - A deterministic set of reasons can be posed across all interviews
    - Need to scale for throughput (CPU/IO)
    - Need to scale for API parallelization
    - Need to remove hotspots
    - Availability and Geo-distribution
    - Need to scale for storage (storage and cache tiers)
  - The constraints or numbers change from problem to problem
  - Solve algebraically first and then put numbers. Algebraic formula is same for all problems
  - If phase 1 takes more time, do not spend time on calculations unless asked for.

Step 5: PostService:

- For each microservice
  - A deterministic set of reasons can be posed across all interviews
    - Need to scale for throughput (CPU/IO) - Yes
    - Need to scale for API parallelization - No
    - Need to remove hotspots - No
    - Availability and Geo-distribution - Yes
    - Need to scale for storage (storage and cache tiers) - Yes
  - Solve algebraically first and then put numbers
    - Total users ~= 1 billion
    - Daily active users ~= 500 million
    - New photos per second ~= 1000
    - Average photo file size ~= 200kb
  - Let's look at each tier separately.

Step 5: PostService:

- App Tier
  - Let t = CPU work time in ms from a single thread
  - So number of operations per second handled by single thread = 1000/t
  - Number of concurrent threads in a commodity server = 100-200 (determined by experiments)
  - So the number of API calls a server can handle per second ~ 100000/t, operating at full capacity.
  - But typical servers operate at 30-40% capacity, so roughly **30000/t API calls per server**.
  -
  - Assuming internet speed of 8 Mbps = 1 mega-byte per sec
  - For a 200 KB photo, we will take around 200 ms
  - Plug in t = 200 ms in above formula = 30000/200 = 150 rps per server
  - For 1000 photos per sec ~= 7 servers.

Step 5: PostService:

- DB Tier
  - Image store estimation
  - Total space required for 10 years of photos (w/o replication or growth)
    = 10 years * ~400 days per year * ~100k sec in a day * 1000 pic per s * 200 kb
    = 80,000,000,000,000,000 b = ~80 Pb
  - (replication to be considered in step 4c)
  - Meta-data estimation
  - PhotoTable: PhotoId(int64) + UserId(int64) + title(varchar 64) + ImgStorePath(varchar 64) + ThumbnailPath (varchar 64) + PhotoLatitude(int) + PhotoLongitude(int) + TimeStamp(datetime) + numLikes(int) + numComments(int)
    = 228 bytes per row * 400 billion photos ~= 100 TB w/o replication.

Step 6: Recall proposing the distributed architecture

- For each microservice
  - If a tier needs scale, scale the tier and propose the distributed system
- This is the most deterministic portion of the solution
- Steps
  - Draw a generic distributed architecture per tier
  - If app server tier and *stateless*, just round robin requests
  - If cache or storage tier
    - Partition the data into *shards* or buckets to suit requirements of scale (changes from problem to problem)
    - Propose algorithm to place shards in servers (*consistent hashing*) (same across all problems)
    - Explain how APIs work in this sharded setting (problem specific)
    - Propose replication (same across all problems)
    - Propose CP or AP (algorithms for CAP theorem do not change from problem to problem)
- This is how each microservice looks within a single data center, either on-premise or VPC in cloud

Step 6: PostService: Reasons for partitioning / sharding

1. If the size of the data is very large
2. If the query throughput is very high (and includes writes)

PhotoTable ~= 100 TB w/o replication.

Assuming 1 shard could hold 1TB, we'll need ~100 shards (w/o replication).
But what should we shard on?

a. UserID
b. PhotoID

Step 6: PostService:

Given a key, which node should I go to (ignore replication for now)? And once I am at the right node, how do I access the relevant record?

Option 1: Assign records to nodes randomly. Would this work?

Option 2: Based on key range

Option 3: Based on the “hash of the key” range

Rebalancing partitions

Suppose the load increases, and you add more CPUs to handle the load.

Or suppose the data size increases, and you add more RAM/Disk to store it.

Or suppose a node fails, and other machines need to take over its responsibilities.

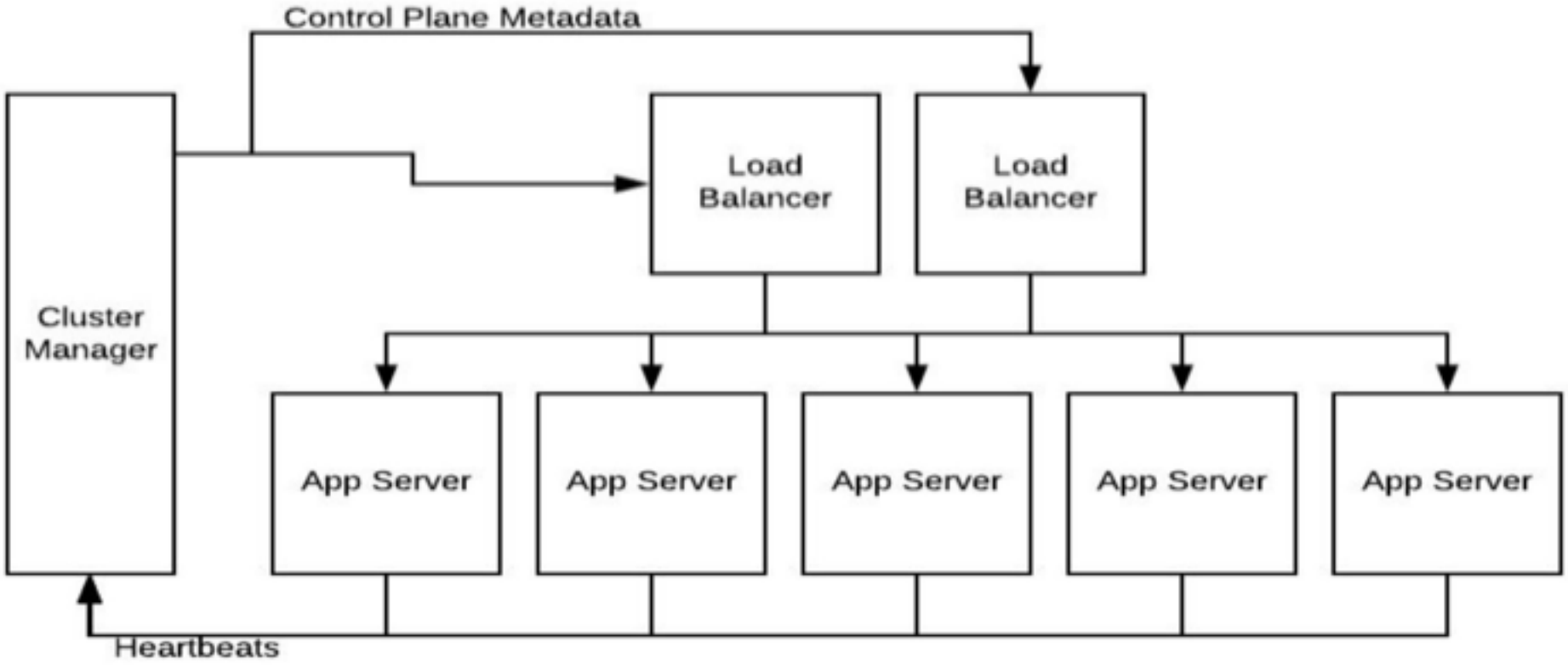
Key ---> Partition ---> Node

If Partition ⇔ Node, then when a node fails, most of the keys need to be moved from node node to another.

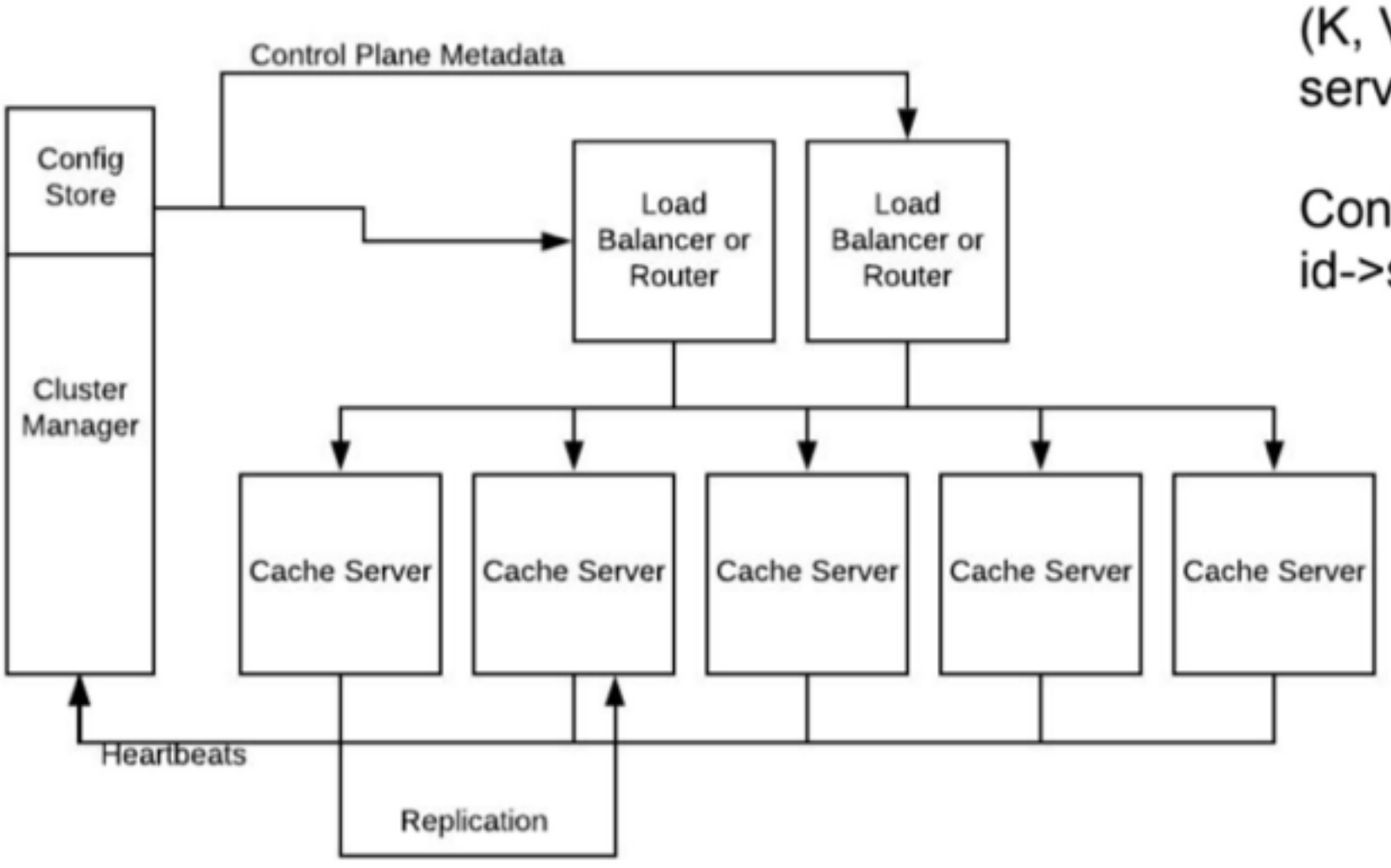
Step 6: PostService

- Sharding
  - Horizontal sharding
  - Map subsets of photos to a single shards
- Placement of shards in servers
  - Consistent hashing
- Replication
  - Yes for availability as well as for throughput
  - To reduce latency for geographically distributed users
- CP or AP?
  - AP: does not need to be strictly nanosecond level consistent
  - Best configuration: N = 3 or more, R = 2, W = 2, high throughput, lowest latency

Distributed Architecture - PostService - App tier



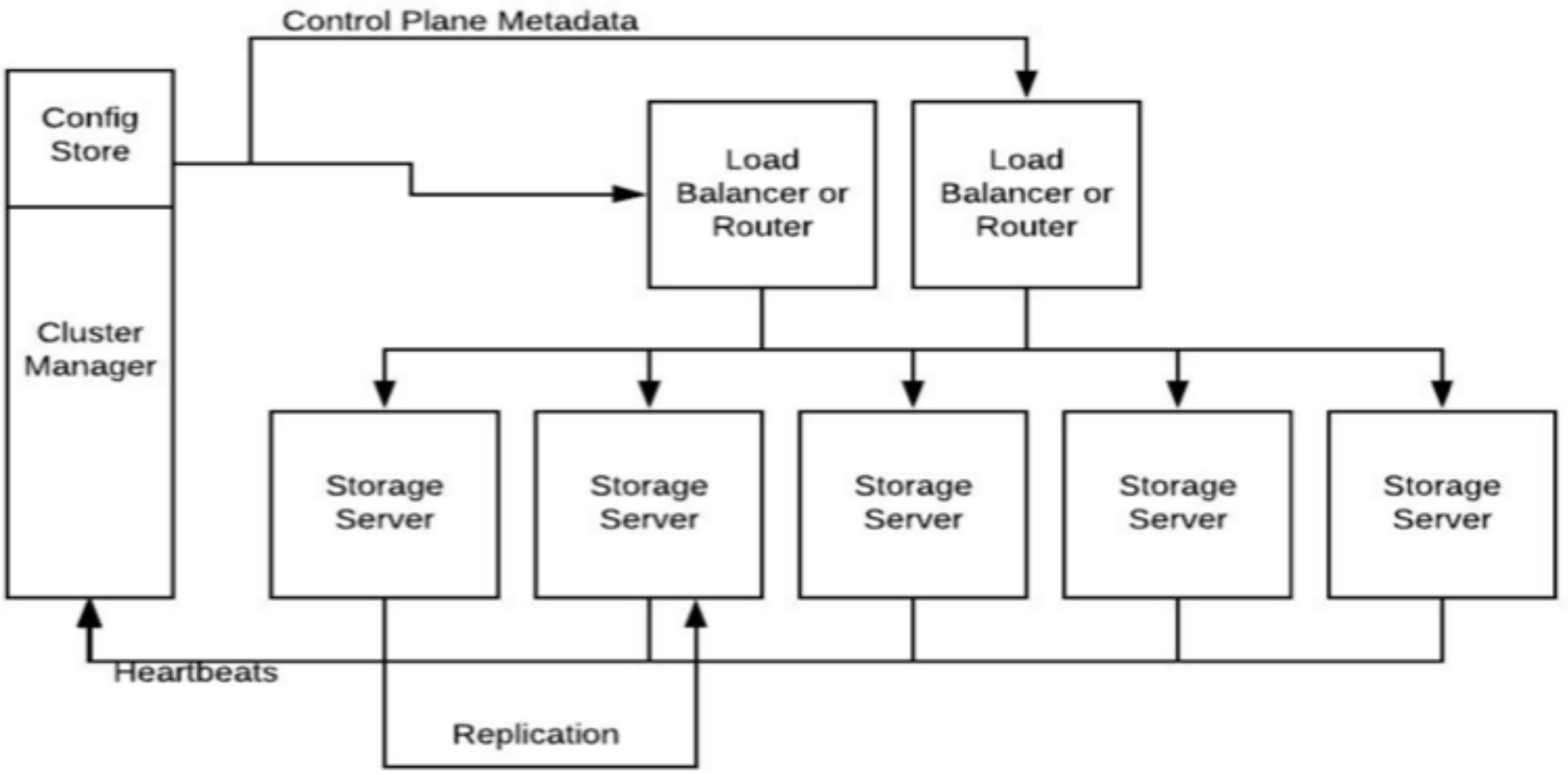
Distributed Architecture - PostService - Cache tier



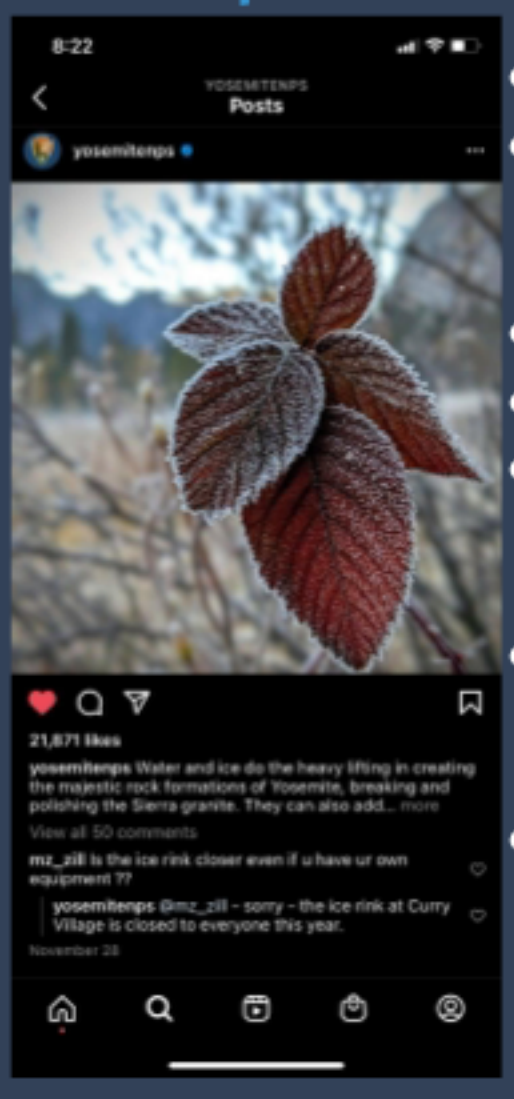
(K, V) -> partition/shard id -> servers

Config store maps shard id->servers

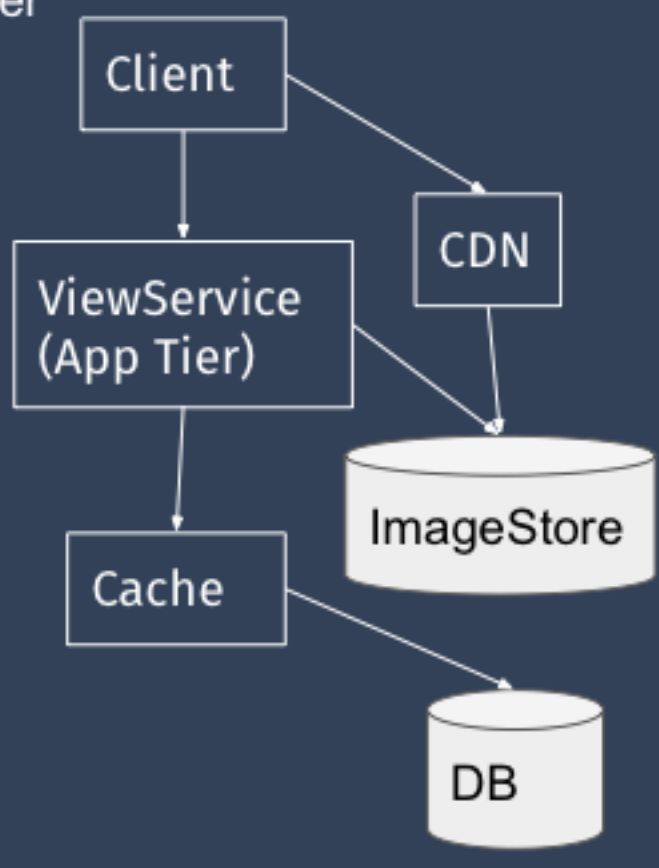
Distributed Architecture - PostService – Storage tier



Step 4: ViewService



- Tiers: App server tier, Cache or In-memory tier, Storage tier
- Data Model: K-V pair
  - K: PhotoID: V: desc, img store path, other properties
  - K: UserID: V: a list of photos
- How to store in Cache tier - Hashmap
- How to store in Storage tier - as a row oriented K-V store
- APIs
  - viewPhoto(PhotoID)
  - viewPhotos(UserID)
- Algorithm in APIs
  - Simple K-V workload to fetch basic data (CRUD)
  - Fetch and aggregate like status, comments
- Flow of APIs
  - Check if meta-data is available in cache
  - If yes, fetch and send it with img or cdn url
  - If not, read from db, aggregate likes, comments
  - Update cache, return to user



{ik} | INTERVIEW KICKSTART

Step 5: ViewService: Scale up the solution

- For each microservice
  - A deterministic set of reasons can be posed across all interviews
    - Need to scale for storage (storage and cache tiers) - Yes
    - Need to scale for throughput (CPU/IO) - Yes
    - Need to scale for API parallelization - No
    - Need to remove hotspots - No
    - Availability and Geo-distribution - Yes
  - Solve algebraically first and then put numbers
    - Total users ~= 1 billion
    - Daily active users ~= 500 million
    - Photo views per second ~= 20000
    - Average photo file size ~= 200kb
  - Let's look at each tier separately.

Step 5: Need for scale in ViewService:

- App Tier
  - Let t = CPU work time in ms from a single thread
  - So number of operations per second handled by single thread = 1000/t
  - Number of concurrent threads in a commodity server = 100-200 (determined by experiments)
  - So the number of API calls a server can handle per second ~ 100000/t, operating at full capacity.
  - But typical servers operate at 30-40% capacity, so roughly **30000/t API calls per server**.
  -
  - Since we are not sending the photo, but leveraging CDN, let's assume that each photo fetch takes around 50 ms.
  - Plug in t = 50 ms in above formula = 30000/50 = 600 rps per server
  - For 20000 photos per sec ~= 34 servers.

Step 5: Need for scale in ViewService:

- Cache Tier
  - Even though we did DB estimation for 10 years, we don't need to store a fraction of that amount in cache.
  - We can store last 5 days worth of photos' metadata in cache
  - So 5 days * 100,000 sec * 1000 pic per s ~= 500 million
  - 500 million * ~200 bytes per pic (meta data) ~= 100 GB
  - Theoretically it could fit in 2 machines with 64 GB RAM

Step 6: Distributed Architecture in ViewService

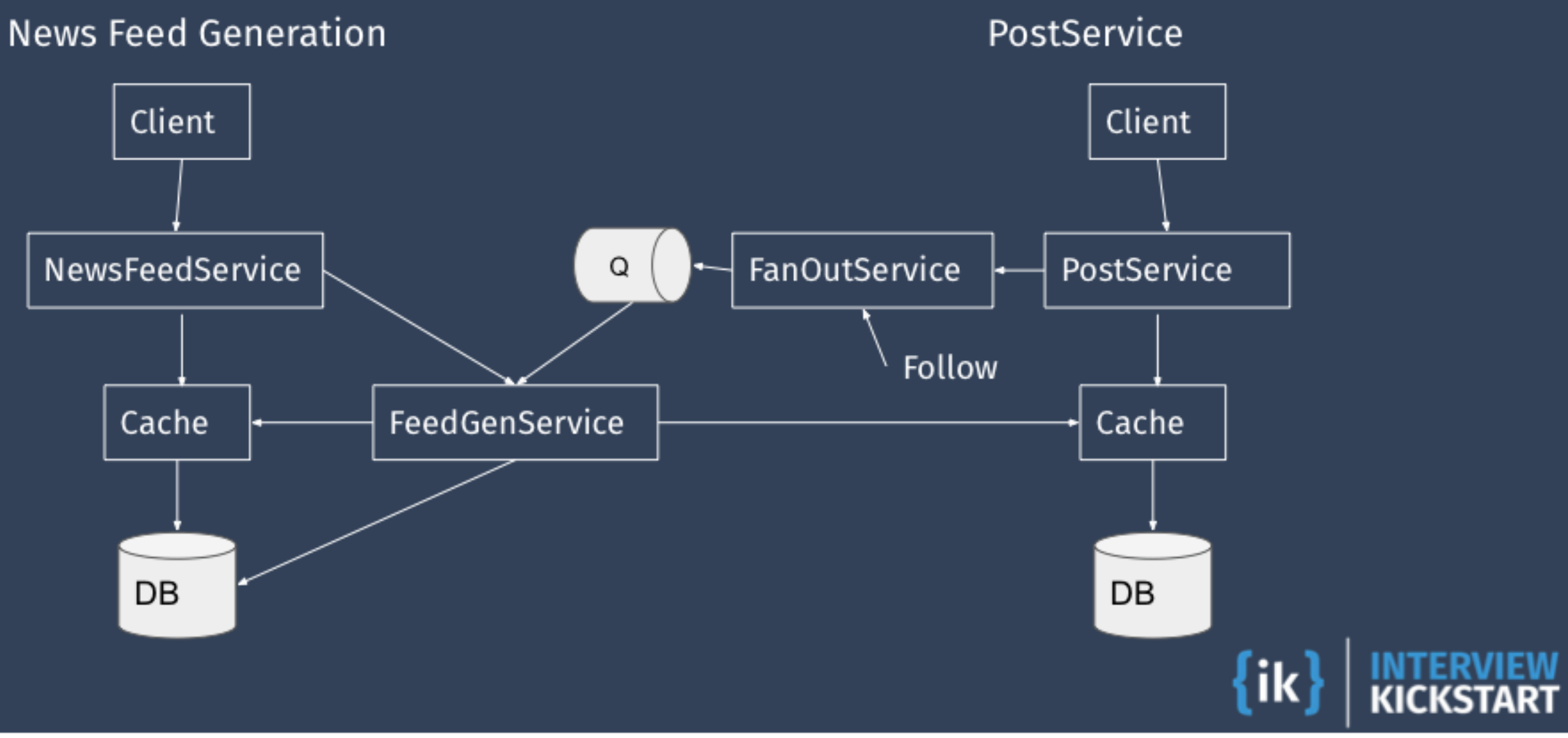
- Same as before
  - Stateless app servers
  - Shard the data
  - Consistent Hashing
  - CP or AP

News Feed

Things Get a little complicated?

How is this problem different from Tiny URL?

Logical Architecture (pretending we have 1 server)



Step 4: NewsFeedService

1. System should generate a news feed for the user, based on who the user follows.
    a. NewsFeedService should fetch top X (say 100) latest, most popular and relevant photos of the people that the user follows.
        i. See if the info is available in cache. If not:
        ii. Get a list of people the user follows.
        iii. Fetch metadata of each user's latest 100 photos.
        iv. Rank the photos (recency, what user liked, location etc).
        v. Return top 100 of that subset.
        vi. Store the list in cache
    b. Alternatively, FeedGenService could pre-generate the same and store it in UserNewsFeed table and in cache.
        i. Generate new data based on last feed generation timestamp in the table.

Step 4: NewsFeedService

1. How should the user fetch the news feed?
    a. Pull model: Clients pull the news feed from servers manually, or at a regular interval.
        i. Pro: Less load on system while following celebrities.
        ii. Con: Empty pull requests if no data.
    b. Push model (long poll or websockets): Servers push new data to users when it becomes available.
        i. Pro: No wasted cycles from client in empty responses.
        ii. Con: Too much data to push for updates from celebrities.
    c. Hybrid: Pull model for pulling data from celebrities and push model for rest of us mere mortals.

Step 4: NewsFeedService

Cache Tier

How to organize the news feed in cache?

Step 4: NewsFeedService

DB Tier

UserNewsFeed table: Primary Key = FeedItemId(int64).
Other columns = UserId(int64), PhotoId(int64), PhotoOwnerID(int64), Description(varchar 64), ImgStoreOrCDNPath(varchar 256), FeedCreationTimestamp(datetime), NumLikes(int32), NumComments(int32))

Let's assume we store 3 days worth of news feed items per user, 1000 feeds per day.

How do you cap it at 3 days?

Step 5: Need for Scale in NewsFeedService:

- For each microservice
  - A deterministic set of reasons can be posed across all interviews
    - Need to scale for throughput (CPU/IO) - Yes
    - Need to scale for API parallelization - No
    - Need to remove hotspots - No
    - Availability and Geo-distribution - Yes
    - Need to scale for storage (storage and cache tiers) - Yes
  - Solve algebraically first and then put numbers
    - Total users ~= 1 billion
    - Daily active users ~= 500 million
    - Photo views per second ~= 20000
    - Avg num of timeline fetches a day by each active user ~= 10
    - Average photo file size ~= 200kb
  - Let's look at each tier separately.

Step 5: Need for Scale in NewsFeedService:

- App Tier
  - Timeline fetch req = 500 million daily active users * 10 fetches a day = 5 billion a day
  - = 5,000,000,000 / 100,000 (approx sec in a day) = 50,000

  - Using previous formula, roughly **30000/t API calls per server**.
  - Since we are not sending the photo, but sending a batch of meta-data, let's assume that each fetch takes around 100 ms.
  - Plug in t = 100 ms in above formula = 30000/100 = 300 rps per server
  - For 50000 photos per sec ~= 167 servers.

Step 5: Need for Scale in NewsFeedService:

- Cache Tier
  - Let's assume we store 500 news feed items per active user.
  - So 500 million users * 500 items * 300 bytes per item = 75,000,000,000,000 ~= 75 TB
  - For easier calculations, assume 75 GB RAM per server
  - So 75 TB / 75 GB ~= 1000 servers
  - How do you store it in these server?

Step 5: Need for Scale in NewsFeedService:

- DB Tier
  - Let's assume we store 3 days worth of news feed items per user, 1000 feeds per day.
  - So 1 billion users * 3000 items * 300 bytes per item = 900,000,000,000,000 ~= 900 TB
  - Assume 4.5 TB per DB shard
  - So 900 TB / 4.5 TB ~= 200 DB servers

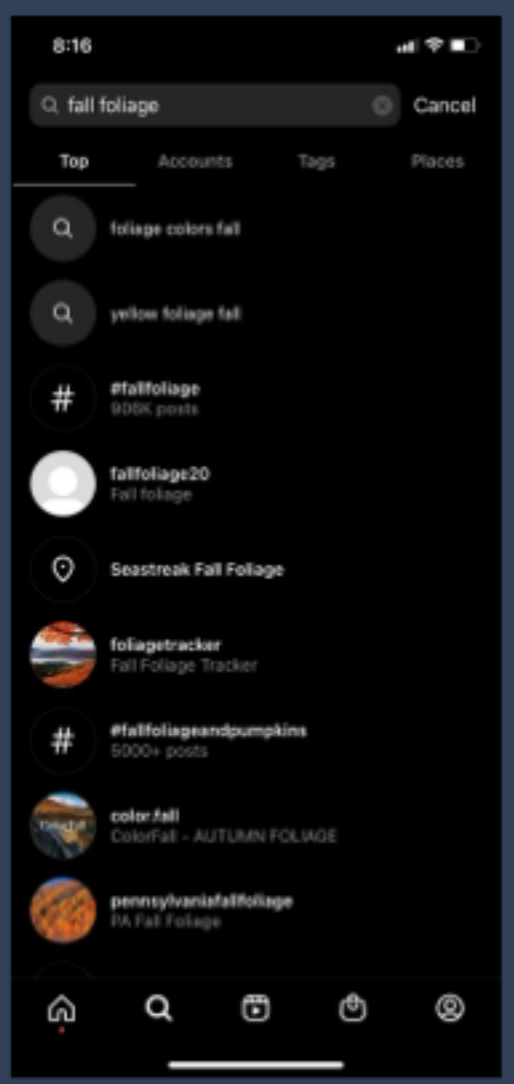
Step 6: Distributed Architecture for NewsFeedService

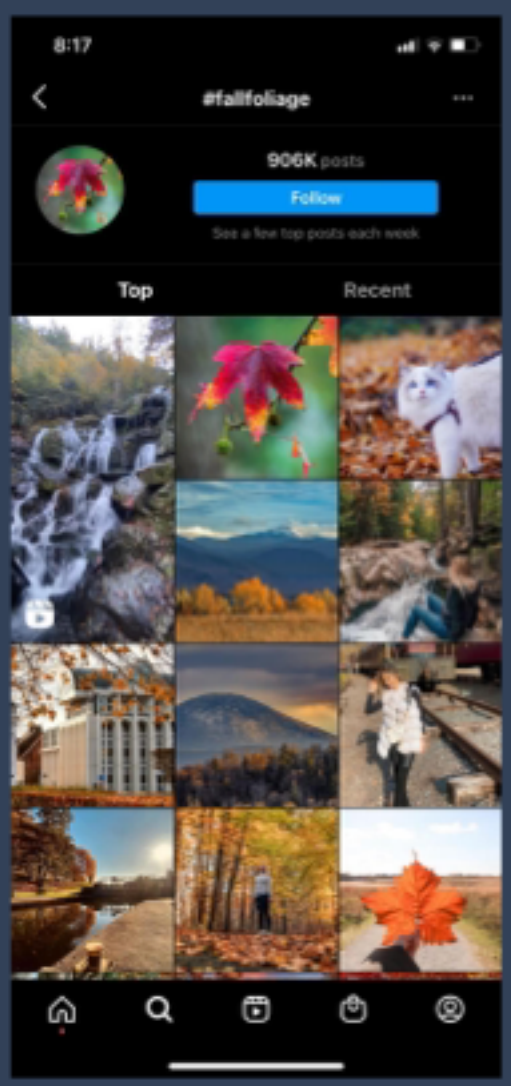
- For App tier, round robin would suffice unless you are holding the cache on the same servers
- For cache tier, shard based on User ID
- For DB tier as well, shard based on User ID
- Explain how APIs will work in this sharded setting

Relational vs NoSQL

joins	fast r & w

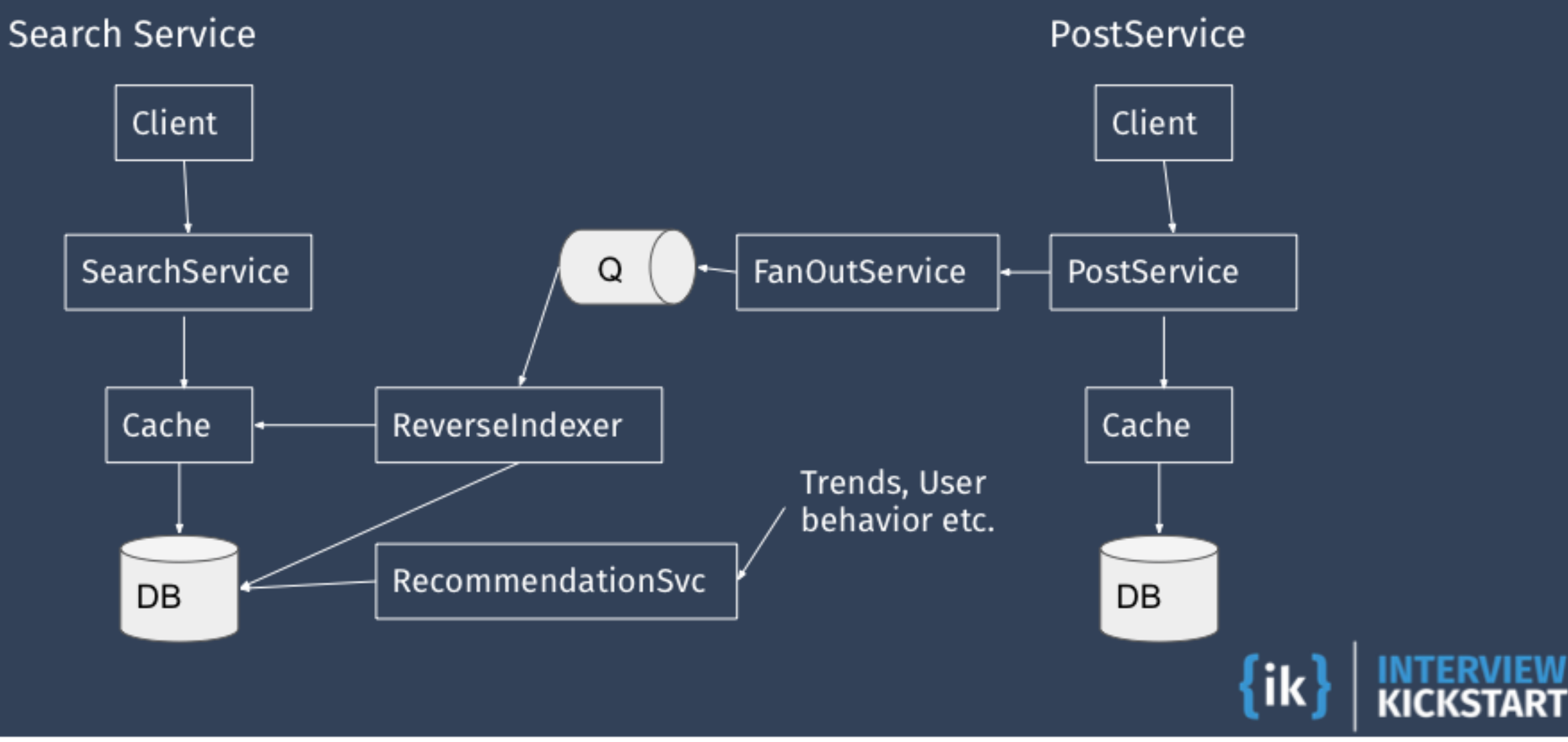
Search Service





{ik} INTERVIEW KICKSTART

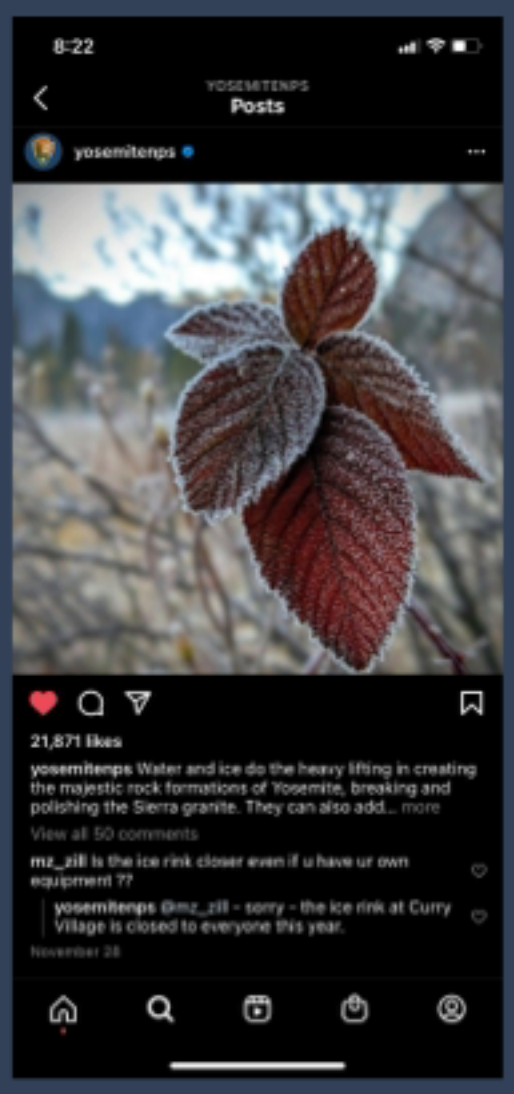
Logical Architecture (pretending we have 1 server)



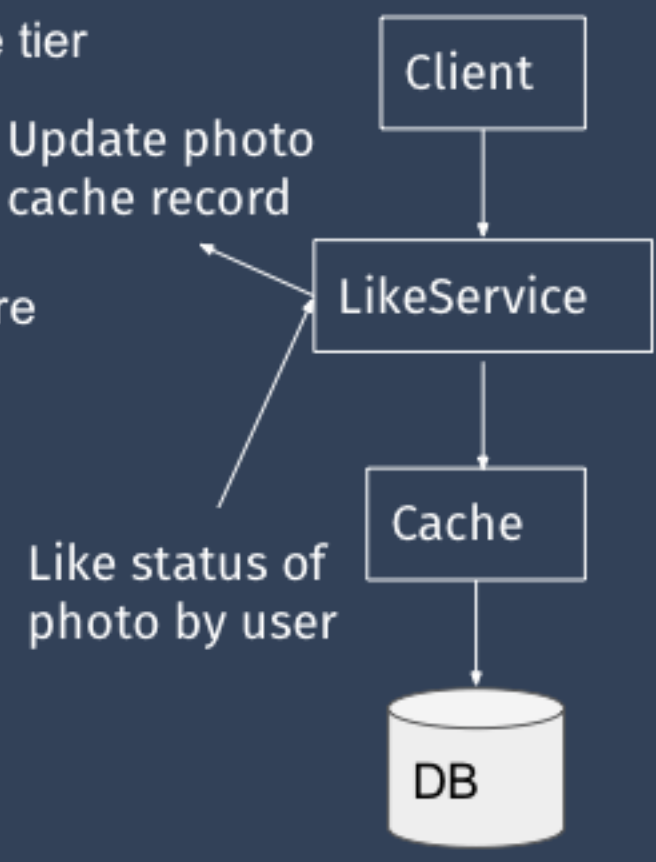
Step 4: Search Service

1. Search Service
    a. Search based on accounts.
        i. Can search in UserTable, using index on name.
        ii. Maintain results in cache to improve performance.
    b. Search on keywords
    c. Search on hashtags
        i. For both keywords and hashtags, an offline process will have to generate a reverse index of keyword/hashtag -> sorted list of photo ids.
        ii. Everytime a new photo is posted, this list needs to be updated, but doesn't need to get reflected immediately.
        iii. You could also mention recommendation service, which will generate recommendations based on user's previous likes, current trends etc. This is what you see when you first click on search button, w/o entering search term.

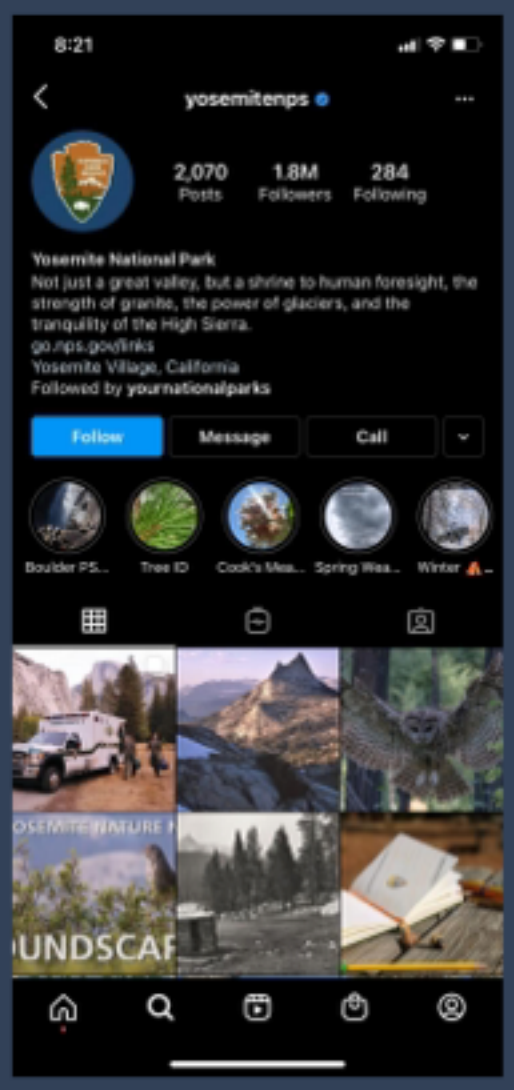
Step 4: LikeService



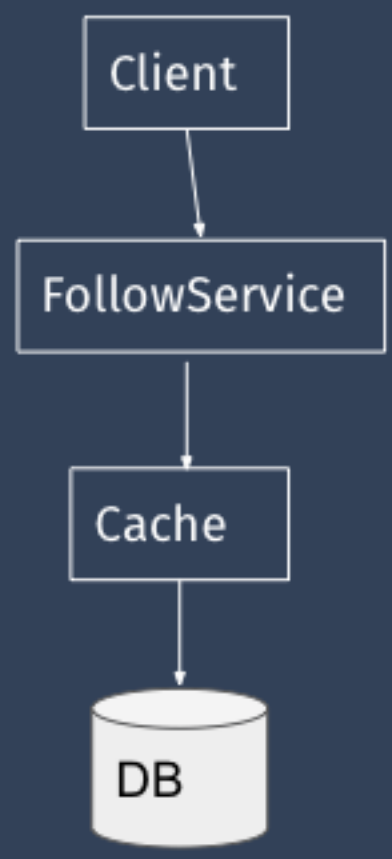
- Tiers: App server tier, Cache or In-memory tier, Storage tier
- Data Model: K-V pair
  - K: UserID+PhotoID: V: time_stamp
- How to store in Cache tier - Hashmap
- How to store in Storage tier - as a row oriented K-V store
- APIs
  - likePhoto(userID, photoID, time_stamp, ...)
  - getLikeStatus(userID, photoID)
- Algorithm in APIs
  - Create(K.V) - CRUD
- Flow of APIs
  - Create a record in the likes table
  - Update aggregate likes count of the photo
    - Offline process or sharded counter?
  - Async fanout to update recommendation logic



Step 4: FollowService



- Tiers: App server tier, Cache or In-memory tier, Storage tier
- Data Model: K-V pair
  - K: UserID+UserID: V: time_stamp
- How to store in Cache tier - Hashmap (userid->\# of followers)
- How to store in Storage tier - as a row oriented K-V store
- APIs
  - follow(userID, userID), unfollow(userID, userID)
  - doesUserFollow(userID, userID) + getFollowingUsers
- Algorithm in APIs
  - Create(K.V) - CRUD
  - update/delete(K,V) - CRUD
- Flow of APIs
  - Check if follow count is within limits
  - Create a record in follow table
  - Update cache with num of followers
  - Async message to update new feed generation logic
  - Similar flow for unfollow



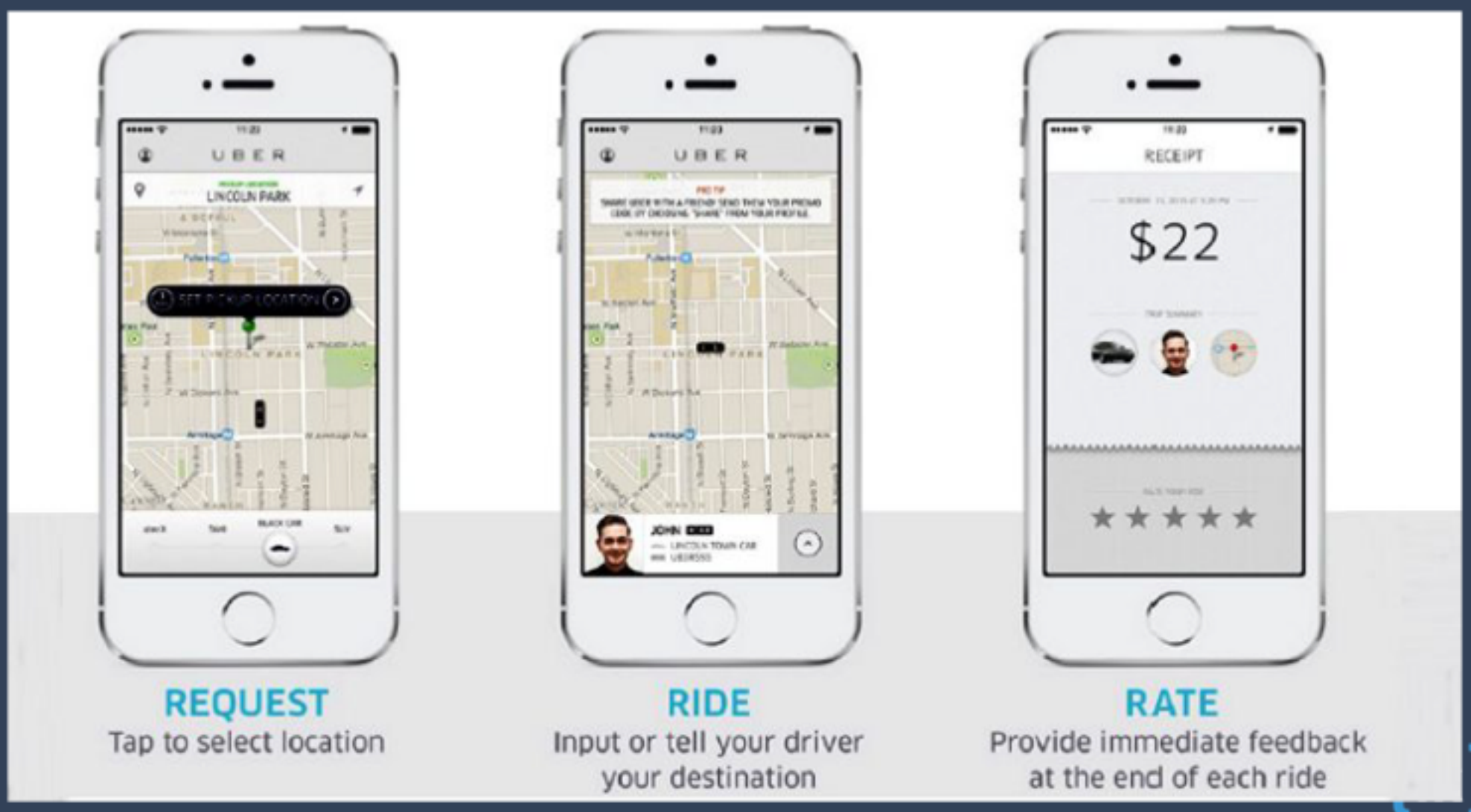
{ik} | INTERVIEW KICKSTART

Problem 2

Online System - Design an Uber clone

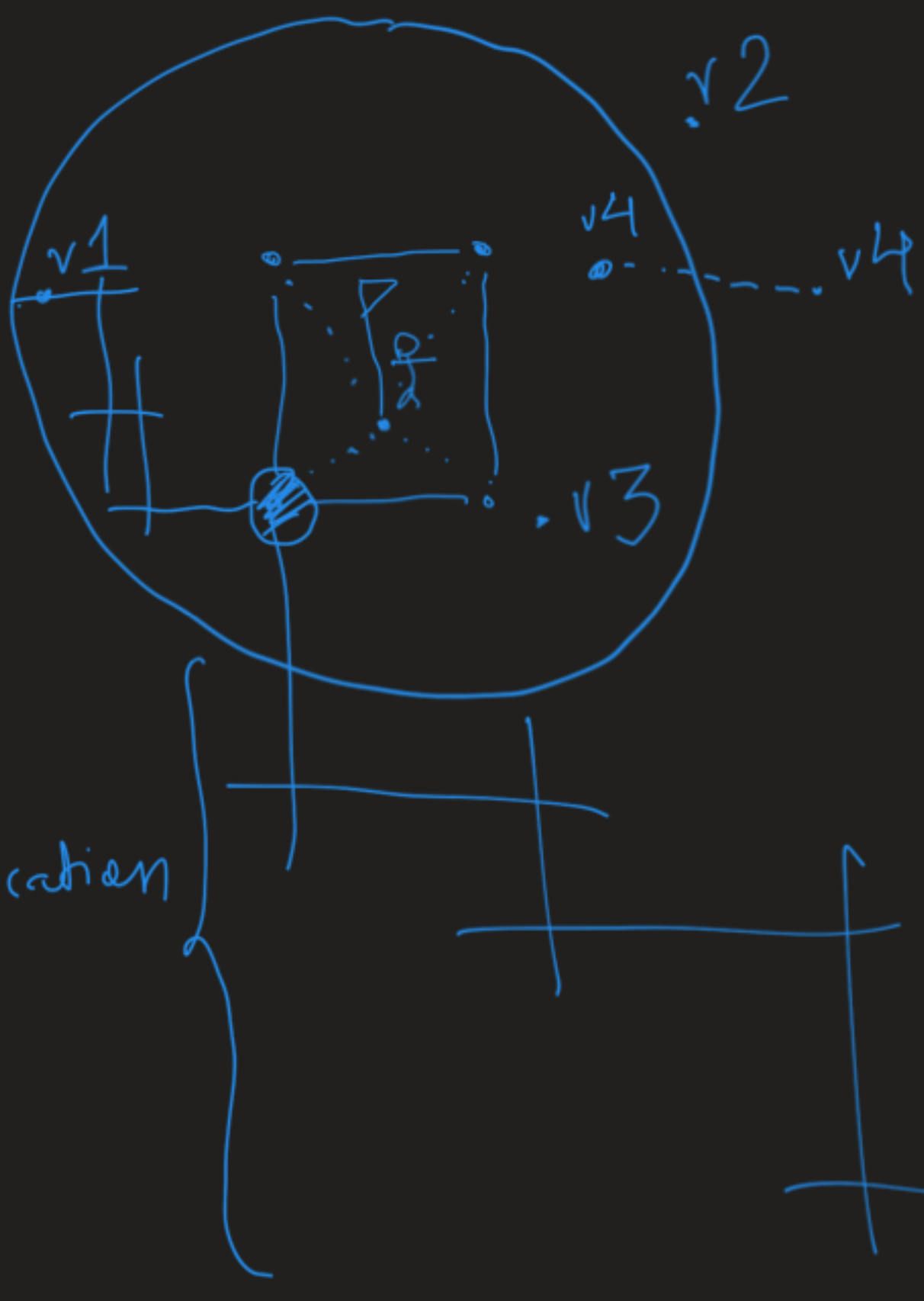
{ik} | INTERVIEW KICKSTART

Uber Screenshots



Step 1: Functional Requirements

- Rider can request for a trip, should be able to give options
  - Book in advance, or real time
- Rider should see cost and an ETA
- User Account Management for Riders and Drivers
- Location based Multiple vehicle Dashboard for Rider (show cars on a map around the user)
- Notify the driver (should be able to accept or reject a trip request)
- Notify the rider (to keep updated on the progress)
- Once trip is set, rider should be able to check the progress of the vehicle
  - Route the driver to the rider, route the driver to the destination during the trip
- Rider pays driver
- Rider and driver endorse each other
- Record historical location data for compliance purposes



* v4 in notification

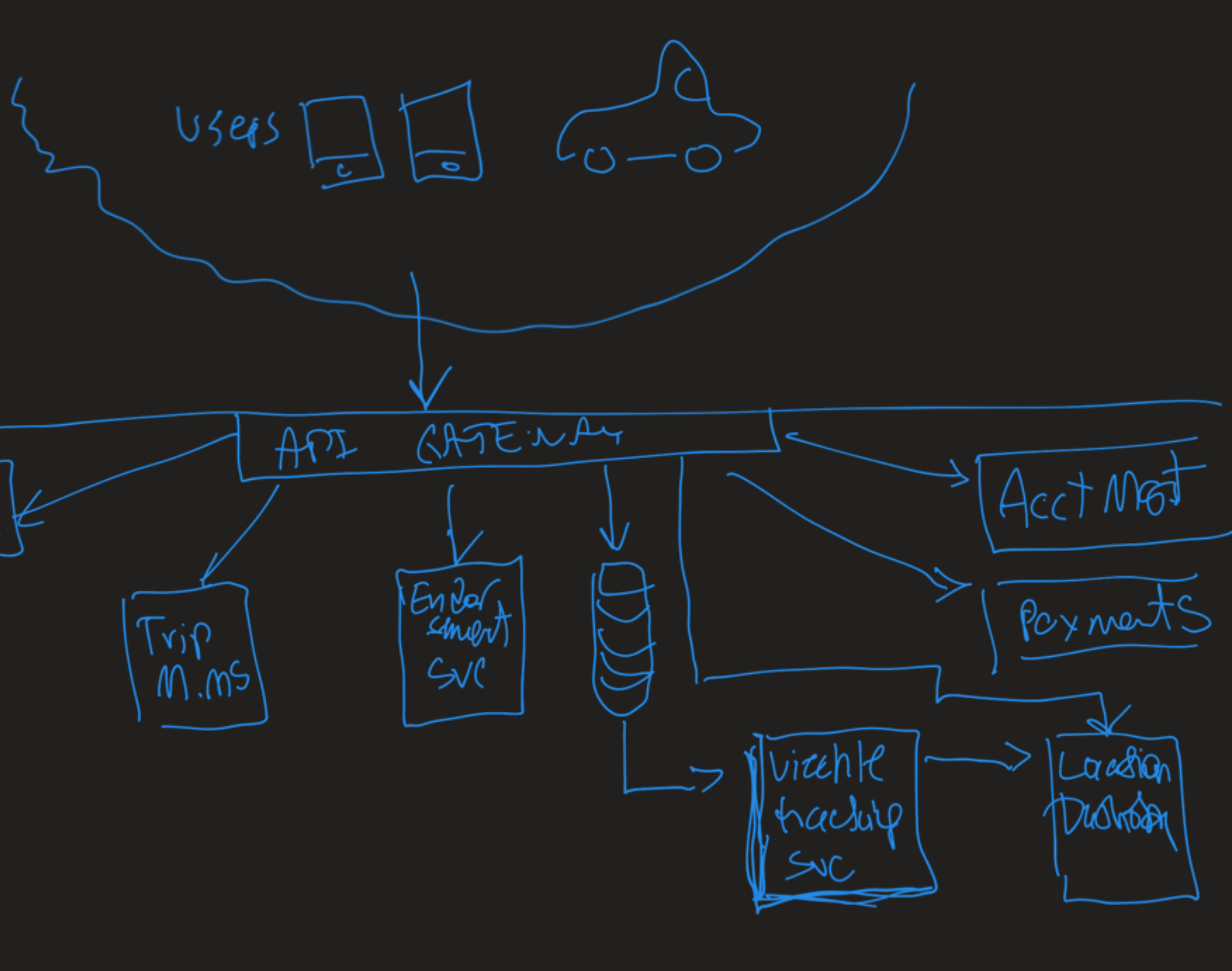
Step 1: Non-Functional requirements

- Number of riders and drivers: 80 million riders + 4 million drivers
- Number of vehicles registered ~ 4-5 million
- Number of trips per month: 40 million

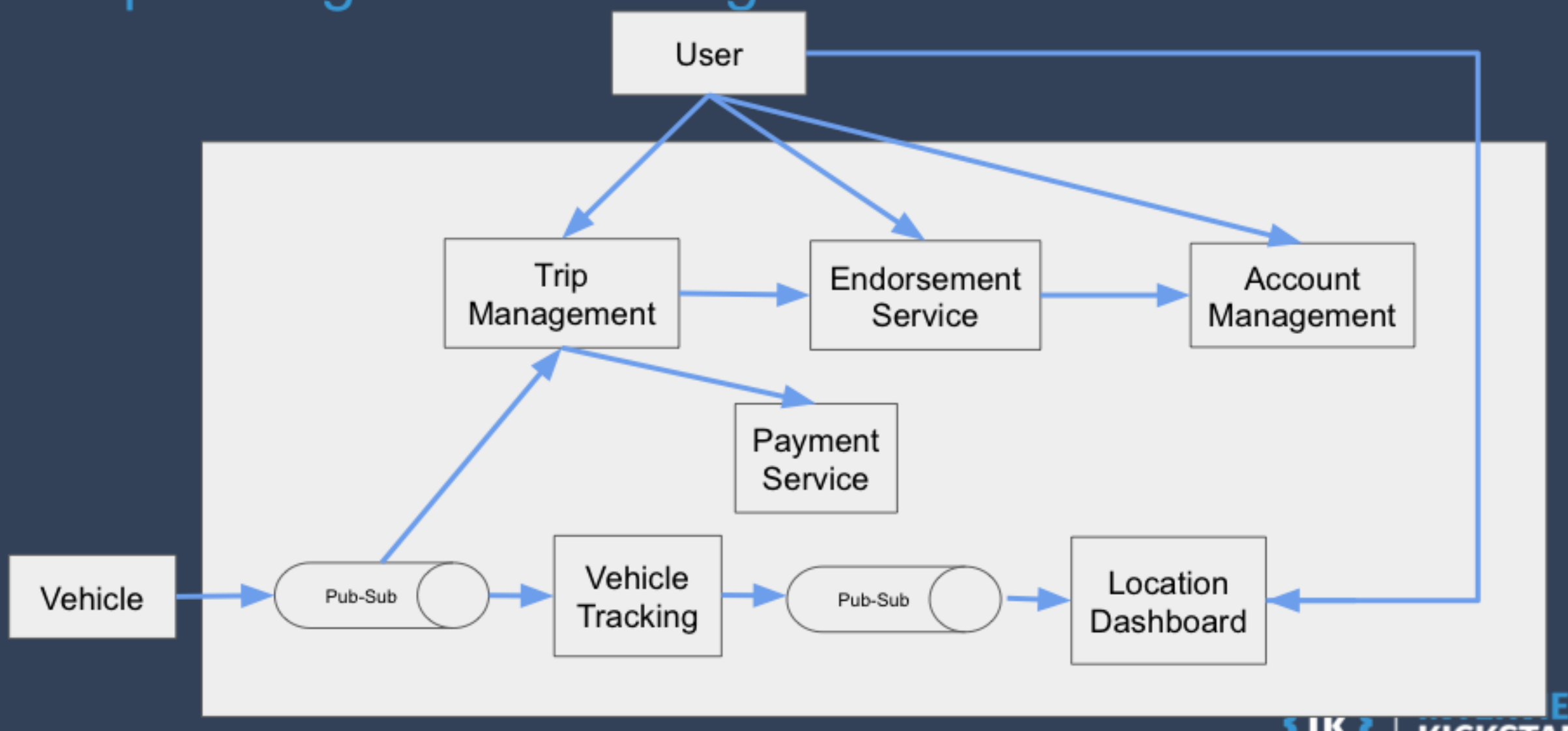
Step 2: Defining Microservices

- Account Management Microservice (Riders + Drivers)
- Vehicle Tracking Microservice
- Location based Dashboard Microservice
- Trip Management Microservice - ETA
- Payment Management Microservice
- Endorsement Management Microservice

Clearly, a breadth oriented problem


Users
API GATEWAY
Auth
Trip M.MS
Environment SVC
Vicente tracking SVC
Location Publisher
Acct Mgt
Payments

Step 3: Logical block diagram



Microservice to focus for Uber

- Vehicle Tracking Microservice
- Location based Dashboard Microservice
- Trip Management Microservice

geohash


A1
A2
B1
B2
A3
A4A
A4B
B3
B4
A4C
A4D
C1
C2A
C2B
D1
D2
C2C
C2D
C3
C4
D3
D4

s2 library

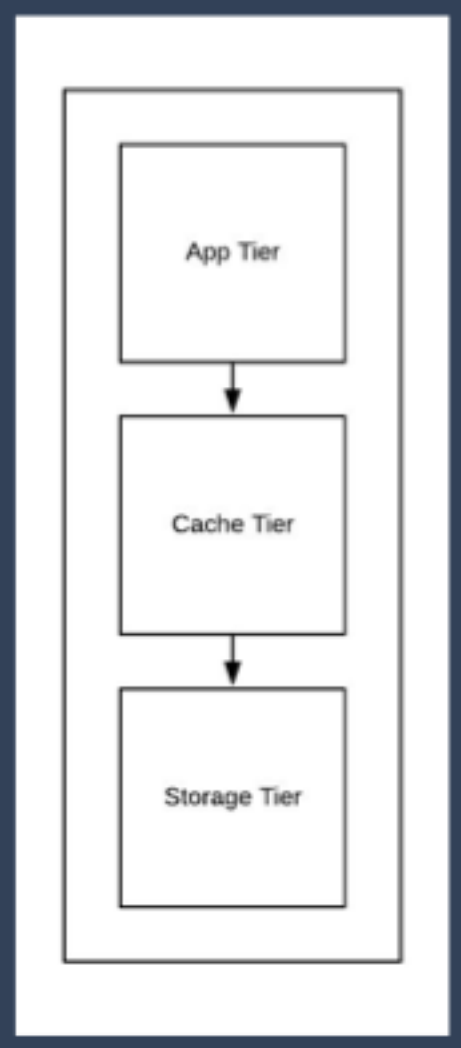
obj id	geohash	lat	long
1	AB1FØ5		
2	AB2F23		
3	BCF...		
4	...		

GeoHash:- How are number of available cabs found

- The earth is a sphere. It's hard to do summarization and approximation based purely on longitude and latitude.
- So Uber divides the earth into tiny cells using the Google S2 library. Each cell has a unique cell ID.
- When Uber needs to find the cabs near a location, a circle's worth of coverage is calculated centered on where the rider is located.
- Uber builds the map overtime by:
  - Identifying the missing roads and incorrect data.
  - Preferred access points at large locations such as airports, universities etc.

Step 4: Vehicle Tracking Microservice

- **Tiers**: App server tier, Cache or In-memory tier, Storage tier
- **Data Model**: K-V pair
  - K: VIN: V: *location*, ts, status, geo-hash, prev geo-hash, other properties
  - Loc_table: geohash, top_left_x, top_left_y, bot_right_x, bot_right_y
- **How to store in Cache tier** - Hashmap
- **How to store in Storage tier** - as a row oriented K-V store
- **APIs**
  - create(vin, lat, long) when registered
  - updateLoc (vin, lat, long)
- **Algorithm in APIs**
  - Simple K-V workload to update current location
  - Calculate geohash based on location (look up geo spatial index)
- **Flow of APIs**
  - App server tier consumes new vehicle registration from driver and location updates from pub-sub
  - Sends to cache server tier
  - Write back caching
  - Update location dashboard service if change in geo hash



Step 5: Vehicle Tracking Microservice

- Need to scale for storage ? Maybe, depending on how many days of history
  - Let's assume we have 2 different tables:
    - To persist current location for persistence if cache goes down
    - To save history for audit/compliance purpose - No need of second level granularity. We can assume we save 1 min snapshot for up to 1 year.
  - Amount of storage in cache and storage tiers: Number of vehicles*size of K-V pair per vehicle
  - For 1st table: 5 million vehicles * 40 bytes per vehicle (vin + ts + lat + long + 2 geo-hash etc) = ~200 mb
  - For 2nd table: 60 points an hr * 24 * 365 * 200 mb ~= 100 TB
- Need to scale for throughput ? yes
  - Number of update API calls per second: thousands per sec
  - Cannot be handled by any single server
- Need to scale for API parallelization ? no
  - APIs themselves are constant latency, so no need to parallelize
- Availability and Geo-location ? yes

Step 5: Vehicle Tracking Microservice

- App Tier
  - Let t = CPU work time in ms from a single thread
  - So number of operations per second handled by single thread = 1000/t
  - Number of concurrent threads in a commodity server = 100-200 (determined by experiments)
  - So the number of API calls a server can handle per second ~ 100000/t, operating at full capacity.
  - But typical servers operate at 30-40% capacity, so roughly **30000/t API calls per server**.
  -
  - For a simple location update + calculating geo-hash, assume it will take around 20 ms
  - Plug in t = 20 ms in above formula = 30000/20 = 1500 rps per server
  - 5 million cars updating location every 5 sec, assume 1 mil updates per sec.
  - For 1,000,000 updates per sec ~= 1,000,000 / 1,500 ~= 667 servers (distributed globally).

Step 5: Vehicle Tracking Microservice

- Cache Tier
  - In memory data per vehicle:
    VIN (~8b) + lat (4b) + long (4b) + 2 geohash (20b) + status (1b) + other (3b)
    ~= 40 bytes per vehicle
  - Number of vehicles ~= 5 million
  - Total memory requirement = 5,000,000 * 40 ~= 200,000,000 ~= 200 mb
  - Can fit into 1 server, but no point in storing NYC/SF/India car info in 1 server
  - Distribute the data across multiple data centers based on Geo location.

Step 5: Vehicle Tracking Microservice

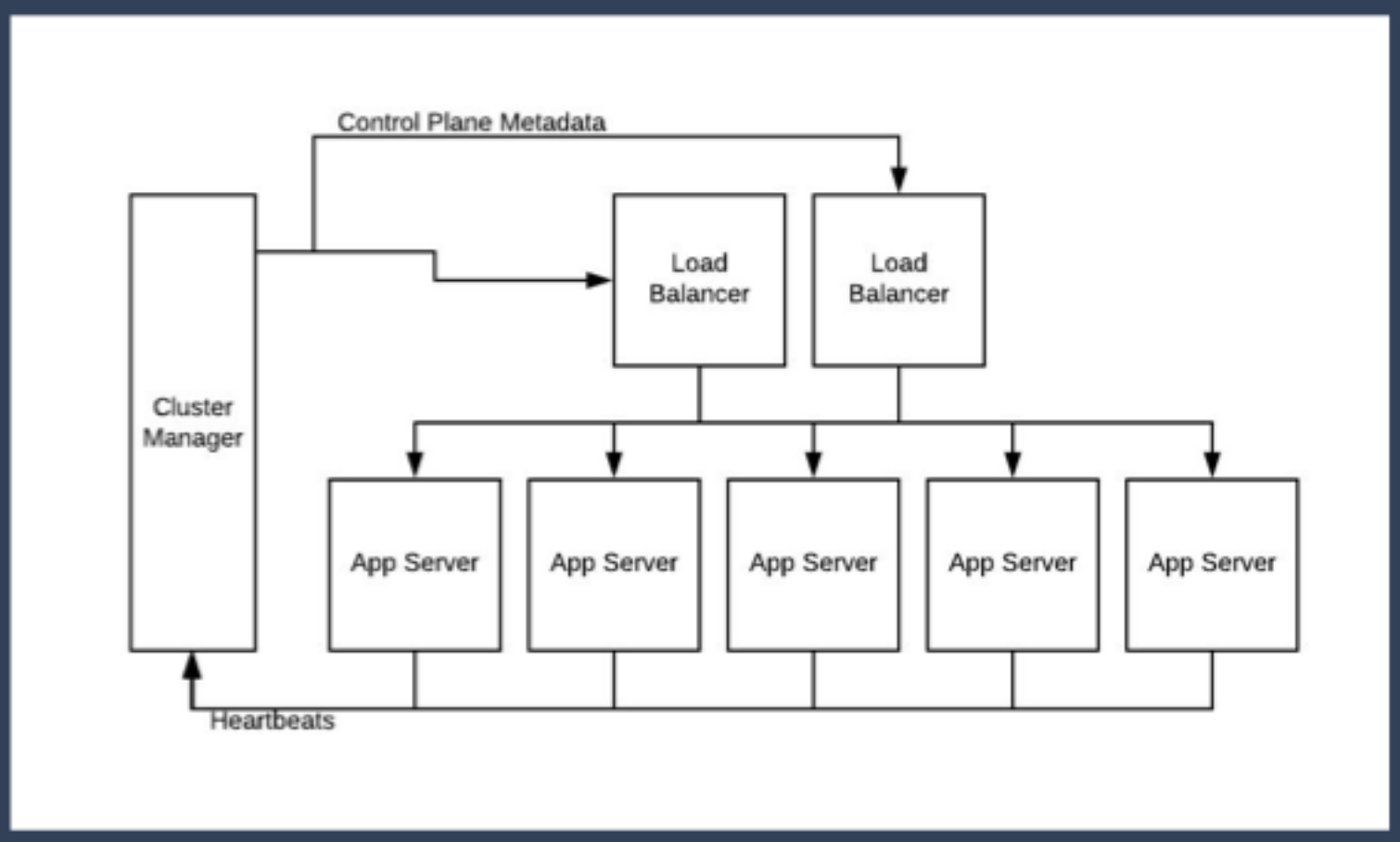
- DB Tier
  - In memory data per vehicle:
    VIN (~8b) + ts(4b) + lat (4b) + long (4b) + status (1b) + other (3b)
    ~= 25 bytes per vehicle (we don't need to store geo-hash in the db)
  - Number of vehicles ~= 5 million
  - Total memory requirement = 5,000,000 * 25 ~= 125,000,000 ~= 125 mb per data pt.
  - If you assume that you are persisting this info for compliance purpose, you don't need second level granularity
  - For minute level aggregation of 1 year data:
    125 mb * 60 * 24 * 365 ~= 125 mb * 500,000 ~= 62,500,000 mb ~= 63 TB
  - Assuming 4TB per db server, you will need ~16 shards (w/o replication)
  - Similar to cache, doesn't make sense to hold data for all vehicles worldwide in a single cluster, distribute it across multiple data centers.

Step 6: Vehicle Tracking Microservice

- Sharding
  - Horizontal sharding
  - Map subsets of vehicles to a single shards
    - Shard based on geo location
    - If you need further sharding, shard based on hash of VIN#
- Placement of shards in servers
  - Consistent hashing
- Replication
  - Yes for availability as well as for throughput
- CP or AP?
  - AP: does not need to be strictly nanosecond level consistent
  - Best configuration: N = 3 or more, R = 2, W = 2, high throughput, lowest latency

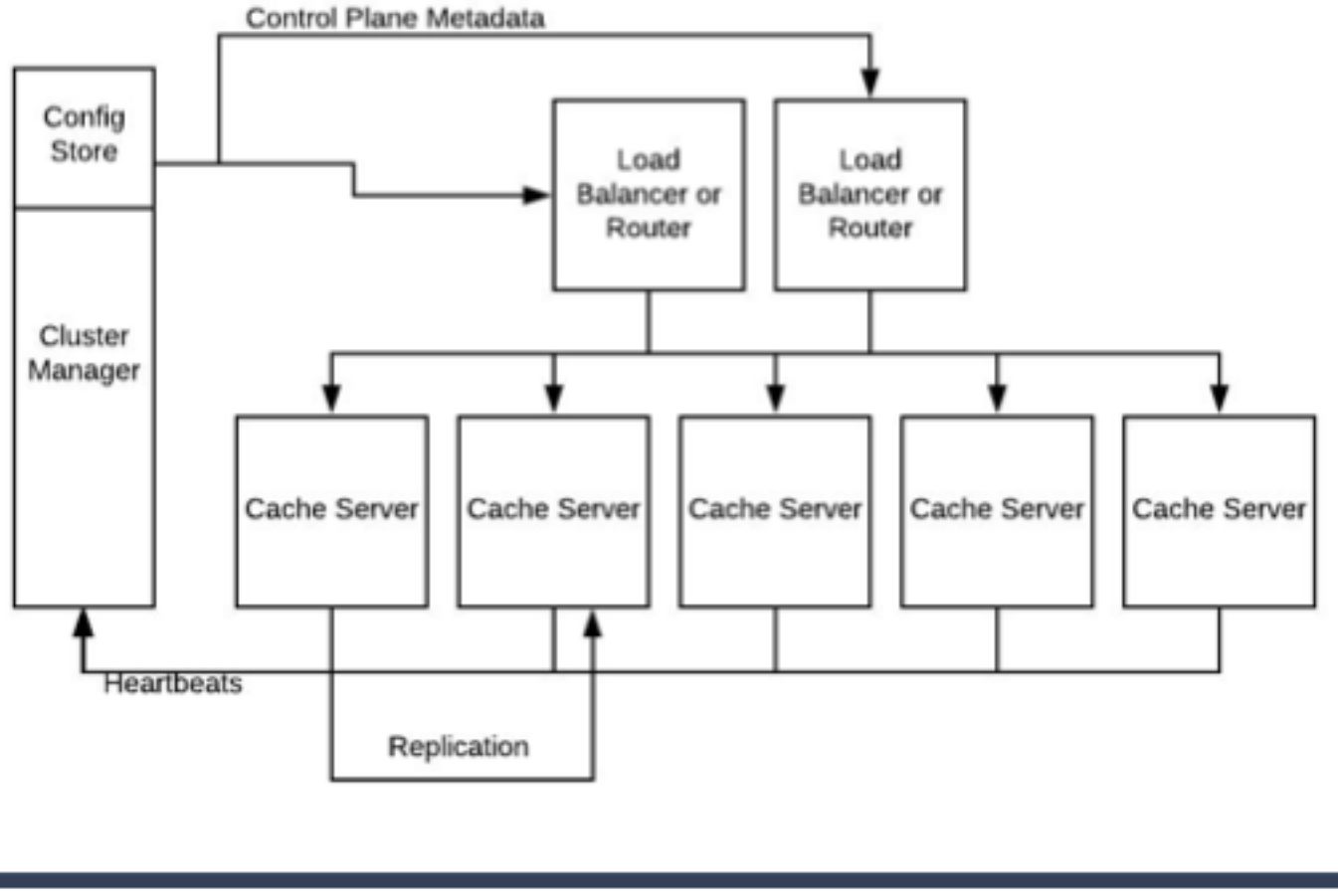
Step 6: Vehicle Tracking Microservice

Architectural layout for App tier



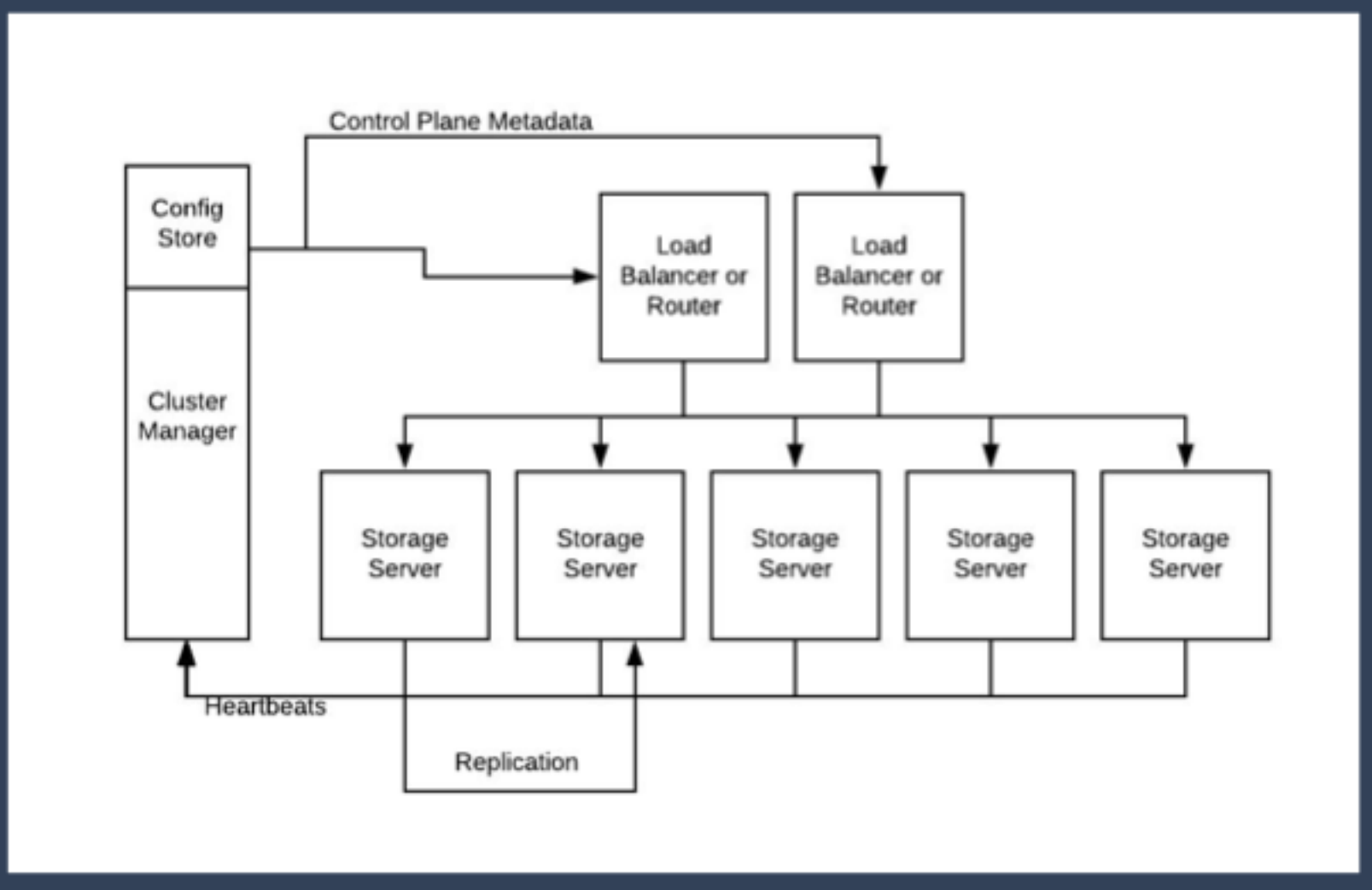
Step 6: Propose the Distributed Architecture

Architectural layout for cache tier



Step 6: Propose the Distributed Architecture

Architectural layout for storage tier

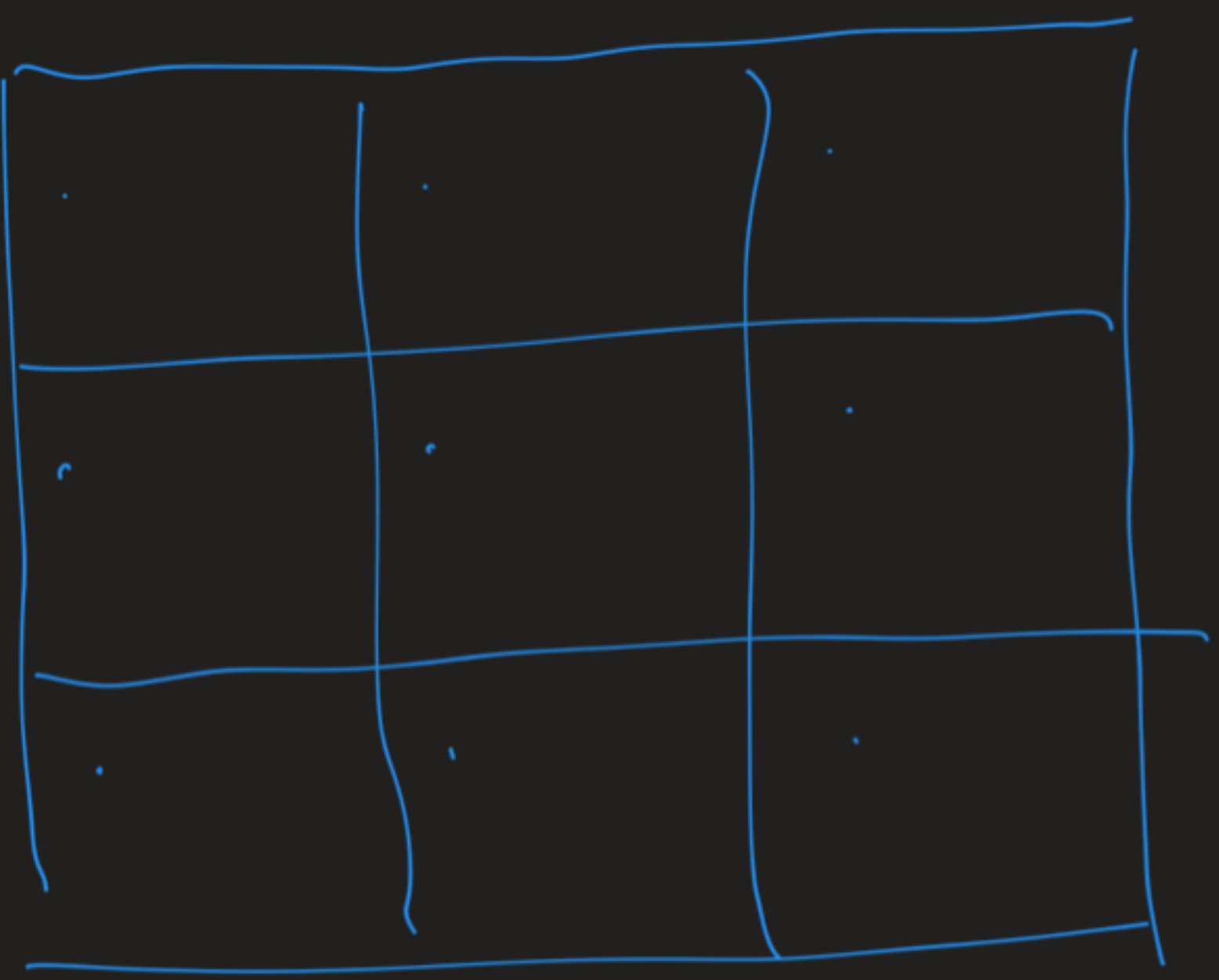


Step 4: Location based Dashboard Microservice

- **Tiers**: Cache tier, Storage tier (just for recovery)
- **Data Model:** K-V pair
  - K: Location Id: V: timestamp, *list of {vehicles + properties + location}*
  - Loc_adj_list: loc_id, north, south, east, west, ne, nw, se, sw
- How to store in in-memory tier:
  - Hash table: Geo hash as the key, list/map/ht of cars within that geohash as values
- How to store in storage tier:
  - Row oriented.
- APIs:
  - get(location id) -> all vehicles given location
  - get(location id, properties) ->search vehicles with certain properties given location
  - Subscribe(driver ids) -> to track the movement of cars in the vicinity

Step 5: Location based Dashboard Microservice

- Need to scale for storage ? yes
  - Amount of storage in cache and storage tiers: Number of unique location ids times size of each K-V pair
- Need to scale for throughput ? yes
  - Number of API calls per second: number of trips /sec * number of location views / trip + geo-hash update from cars.
  - Number of servers required per tier - calculations from steps discussed in a different deck
- Need to scale for API parallelization ? Maybe
  - APIs themselves are constant latency, so no need to parallelize
  - But internally, if you are requesting vehicles in the neighboring blocks, you can parallelize.
- Availability ? yes
- Geo-location based distribution ? yes



9t

1t + coordination

Step 5: Location based Dashboard Microservice

- Cache Tier
  - What do we need to store in memory?
    - Key: Geohash (6 bytes) + timestamp (4 bytes) + values: List/Map/Hash Table of VIN (~8b) + lat (4b) + long (4b) + status (1b) + type (1b) = 18 bytes per vehicle (round up to 20)
    - Assume around 20 cars in 1 cell ~= 400 bytes per cell
  - Number of cells of 6 char geohash ~= 32^6 ~= 1 billion
  - Total memory requirement = 1,000,000,000 * 400 ~= 400,000,000,000 ~= 400 gb
  - Assuming 1 server can store about 40 GB, you will need about 10 machines

  - Other way to calculate:
  - Area of US ~= 10 million sq. km
  - If you divide that into cells of 500m * 500m, you will get ~40 million cells

Step 6: Location based Dashboard Microservice

- Sharding
  - Horizontal + vertical sharding
  - Horizontal : Geo distributed (Shards in west coast containing geo hash near that region)
  - Vertical: Within a geo distribution, shard based on driver ids to distribute write ops:
    - Shard1 : GeoHash1 -> D1, D11
    - Shard2 : GeoHash1 -> D2, D12
- Placement of shards in servers
  - Consistent hashing
- Replication
  - Yes for availability as well as for API parallelization of fetching data from neighboring cells
- CP or AP?
  - AP: does not need to be strictly nanosecond level consistent
  - Best configuration: N = 3 or more, R = 2, W = 2, high throughput, lowest latency

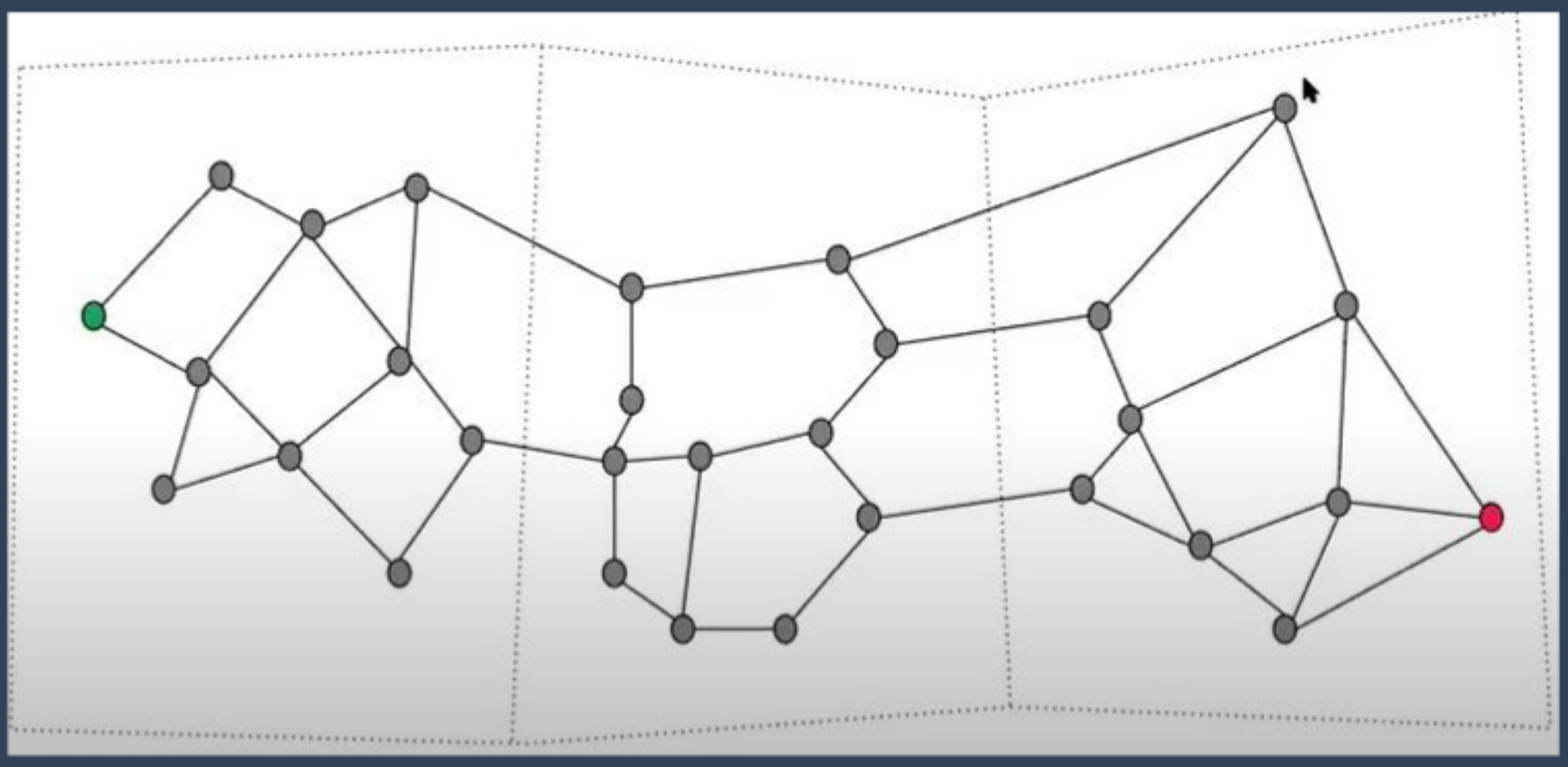
Step 4: Trip Management Microservice

- Could be divided into 2 parts
  - MapService to get map, calculate route, ETA
    - getMap, getRoute (let's assume they are handled by external APIs)
    - getTravelETA
  - TripInfo for operations around trip such as
    - requestTrip
    - acceptTrip
    - cancelTrip
    - rejectTrip
    - completeTrip
    - getTripStatus
    - tripHistory

How are ETA's calculated

- ETA is the estimated time of arrival of a cab.
- To calculate the ETA for a ride we need:
  - Track the available cabs nearby
  - Track the rides that are ready to finish the ride that might take less time
- Uber models the road network as a graph and has 4 layers of computations
  - Map data -> routing info (graph) -> traffic analysis -> ML
  - Gps data on road segments to calculate real time speed
  - Use traffic data to compute edge weights
  - ML to bridge gap between predicted ETA and actual ETA.

How are ETA's calculated



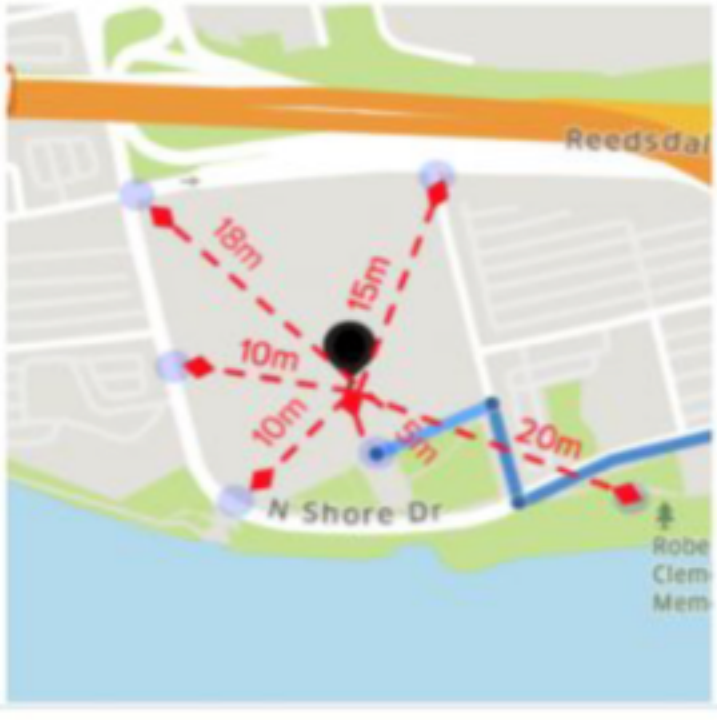
How are ETA's calculated

Identify

Identify pickup/dropoff coordinates for each location pin.

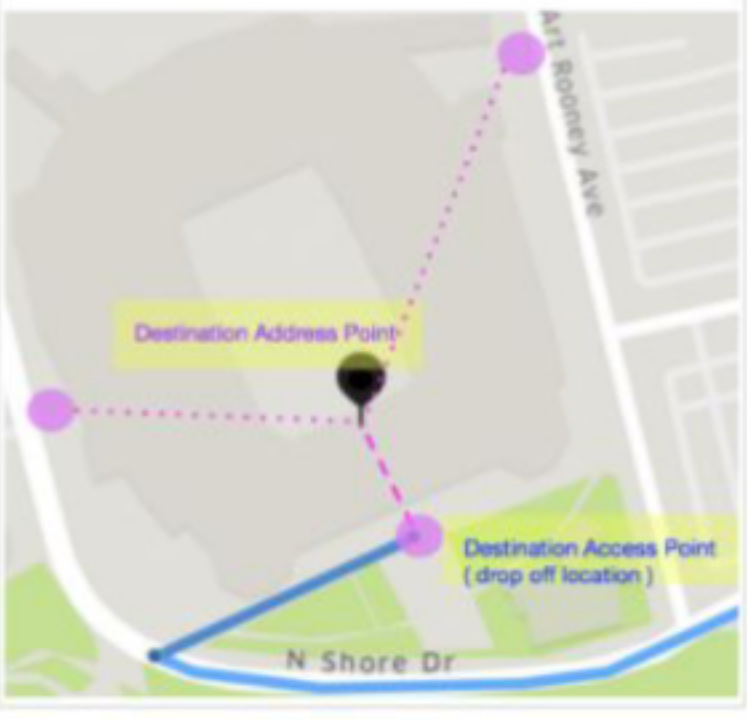
Compute

Compute geospatial proximity of location pin to pickup/dropoff coordinates.



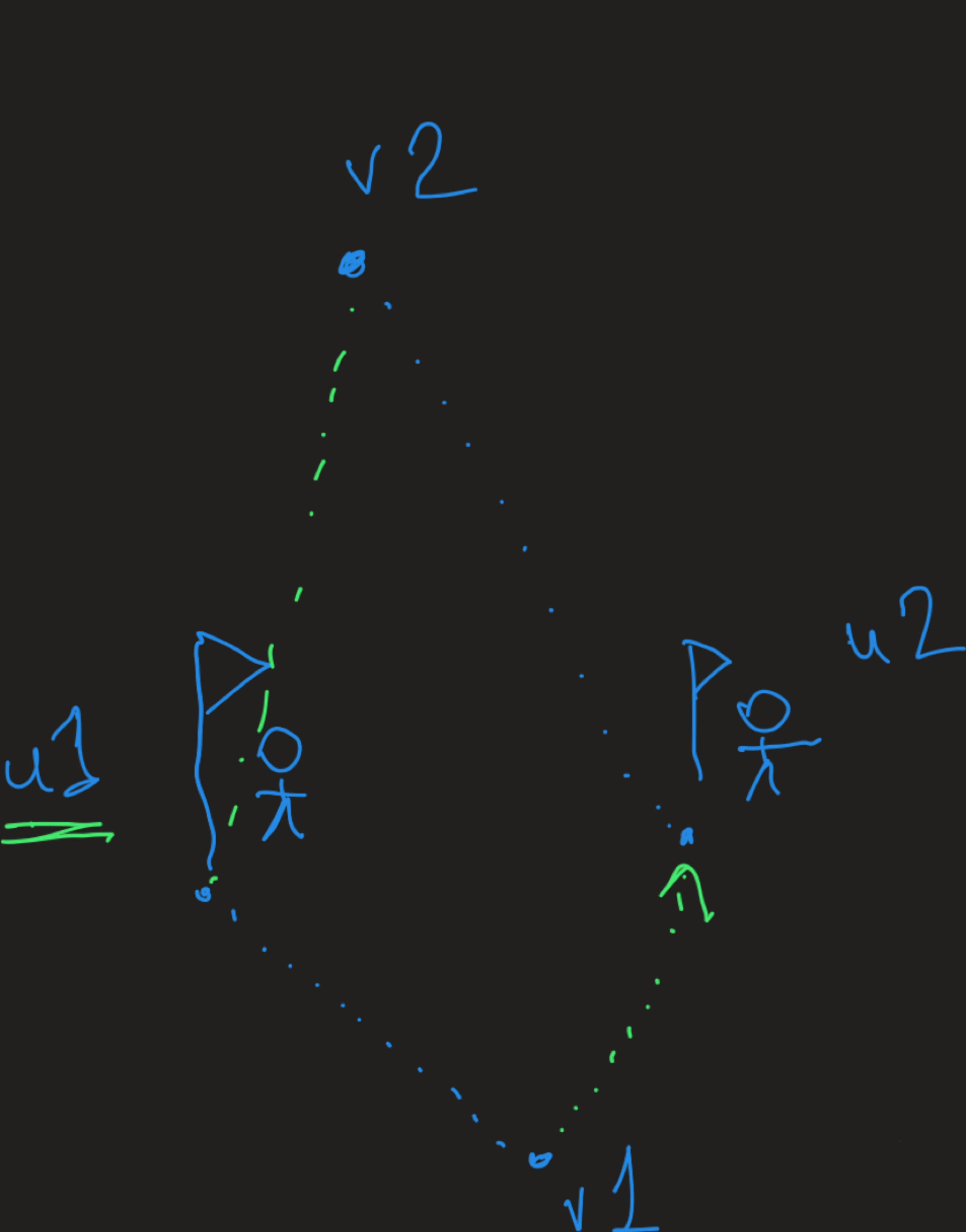
Set

Set the **closest** pickup/dropoff to be **preferred pickup-points**.



Step 4: Trip Management Microservice

- TripInfo related operations:
- Data Model: K-V pair
  - K: <Trip id>: V: *rider, driver, startTime, endTime, price, properties, state of trip*
  - K: <driver/Trip id>: V: *rider, driver, properties*
  - K: <rider/Trip id>: V: *rider, driver, properties*
- How to store in Storage tier
  - Row oriented K-V store
- APIs
  - create(K, V) for model 1 when trip requested
  - update (K, V) for model 1 when when trip changes state
  - Update (K. V) for models 2 and 3 when trip completed
  - subscribe(driver_id) for location updates.
- Flow of APIs
  - Request trip -> app server calls Location service to get list of vehicles
  - Logic to find the best match
  - Optimize for overall wait time
  - Create trip
  - For each vehicle, send notification to driver
  - If driver accepts, update trip
  - Get driver's ETA


v2
u1
u2
v1

Step 5: Trip Management Microservice

- Need to scale for storage ? no
  - Amount of storage in cache and storage tiers: Number of trips per second*size of K-V pair per trip * sizing for a few years
  - 4,800,000,000 trips in 10 years * 40 bytes per trip ~= 192 GB
- Need to scale for throughput ? no
  - Number of update API calls per second: few hundreds per sec
- Need to scale for API parallelization ? no
  - APIs themselves are constant latency, so no need to parallelize
- Availability ? yes
- Geo-location based distribution ? yes

Step 6: Trip Management Microservice

Architectural layout for every tier same as previous microservices

Servers needed only for replication

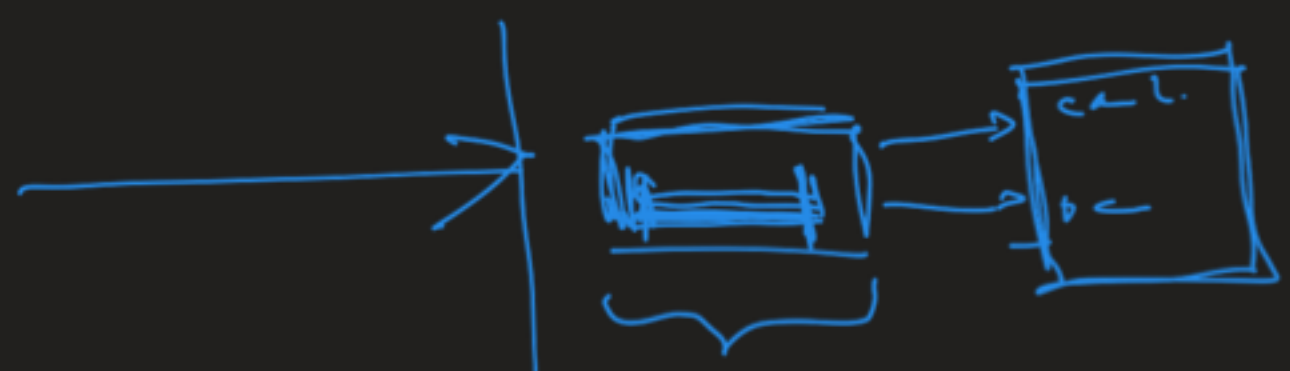
Price Surge in Uber - Optional

- The prices are increased when there is more demand and less supply.
- Surge is used to help to meet supply and demand as by increasing the price more cabs will be on the road when the demand is more.
- Uber uses “surge pricing algorithm” to determine the prices of rides based on the laws of supply and demand.
- This algorithm is ran every 5–10 minutes to determine the prices of rides for places of high demand and the price is determined by the availability of cars and the demand for the cars.
- The goal of the algorithm is to maximize the number of rides that can be provided (especially of times with low supply and high demand).

Uber - recap

Recap the solution

- RPS
- hit ratio
- scenes



getGeoHash(1, 1) -> String

Find Drivers Around Location

- Get geohash of my loc
- Get nearby locations -> ___
- select vins FROM G_table WHERE geoHash LIKE ('___%')

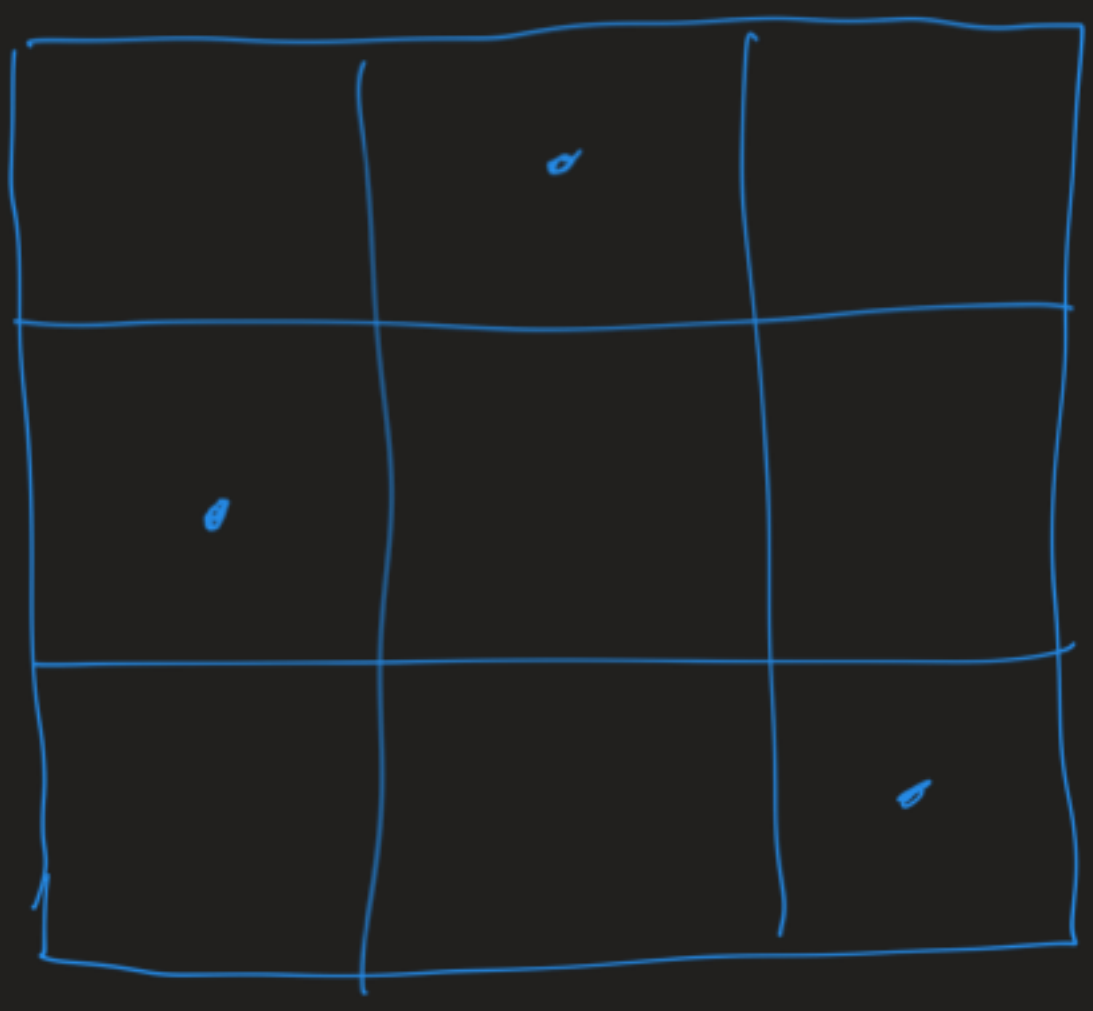
Extra Slides

Useful data

- People, population:
  - Americas (NA, SA, USA, Brazil)
  - Europe (DE, UK, ES)
  - Asia (India, China, JP)
  - Africa
  - Largest cities
- People, general:
  - Life expectancy
  - People per household
  - Median income (US)
  - Time online
- Metrics
  - CTR, CPM
  - Largest ad players
- Users
  - FB, Netflix, Spotify, etc
- Tech high level
  - Internet access, % per country
  - Via smartphone
  - Android vs iPhone
  - Main social networks
  - Mobile traffic %
  - # of apps
- Tech low level
  - KB, MB, GB, TB, PB, EB
  - Image sizes (L|M|H)
  - Video size (L|M|H)
  - Size of text
  - Latency
    - Across the globe, network
    - Read from HDD, SSD, Memory, Cache
    - To load a page in web/app
    - Above the Fold

Cloud Systems by Company

What	Amazon	Microsoft	Google
Compute	EC2	Virtual Machine	Computer Engine
Load Balancing	Elastic Load Balancing	Azure Load Balancer	Cloud Load Balancing
Serverless Compute	Lambda Functions	Azure Functions	Cloud Functions
Containers	Elastic Container Service	Azure Kubernetes	Google Kubernetes
Object storage	S3	Azure Storage	Google Cloud Storage



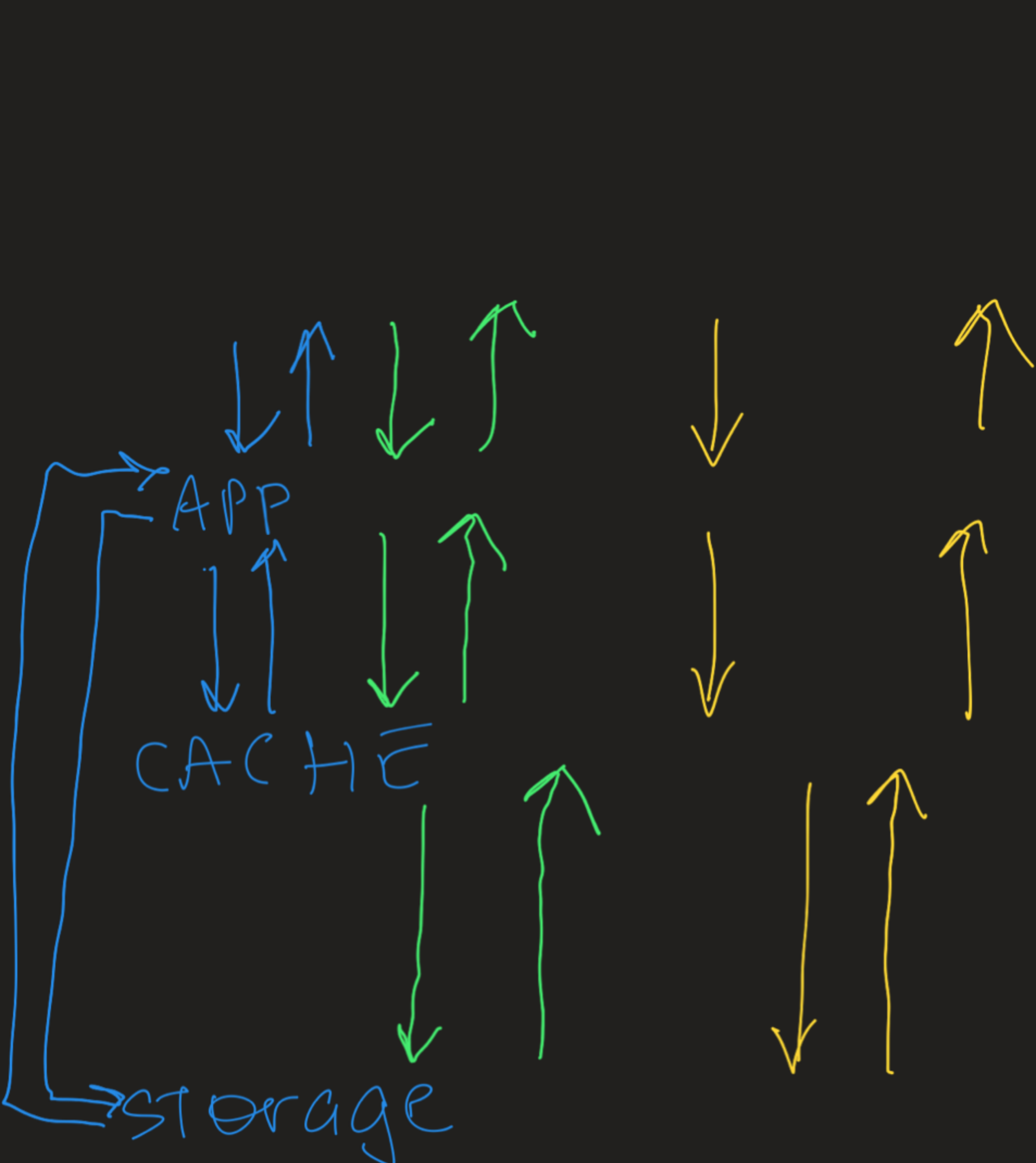
1. ✓

$t = gx$

2. ✓

coord time + $\frac{1}{2}x$

API parallèzation


APP
CACHE
storage

Cheat Sheet

- Redundancy - Replication and/or Snapshot.
- High write load/storage - sharding
- High read load - replication
- Scaling - Primary-Secondary or Leaderless
- High read/write concurrency - Duplicate record.
- Big objects - split into smaller chunks.
- Real time communication - Long polling / Websockets
- Search - index/primary key + reverse index
- Decoupling - message queues / pub-sub

Thank you!

Credits: Omkar Deshpande and Niloy Mukherjee for their help and for allowing to use a few of their slides, Vivek for cheat sheet inspiration.

{ik} | INTERVIEW KICKSTART