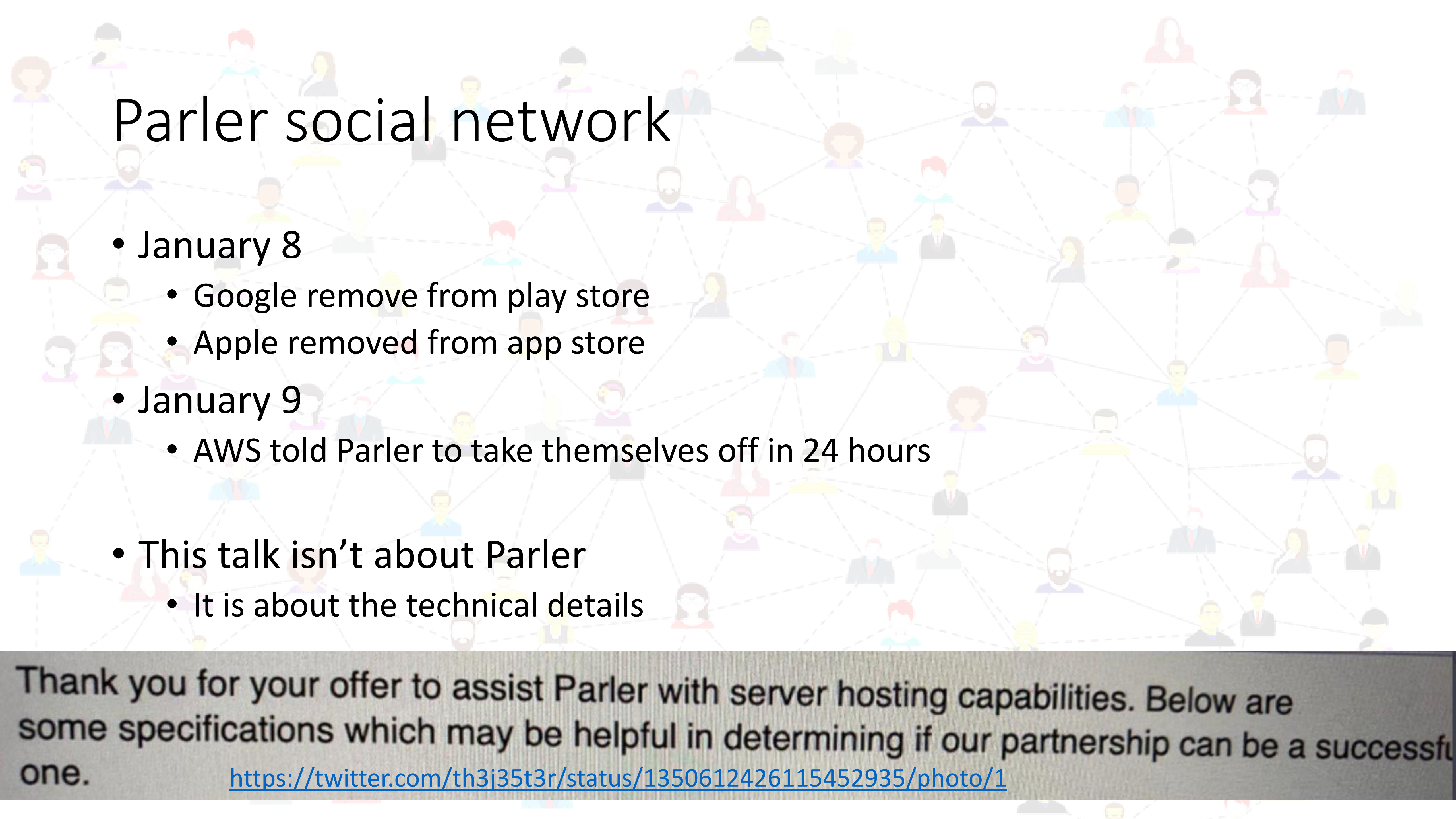
Architecting a large social network without breaking the bank

Oren Eini

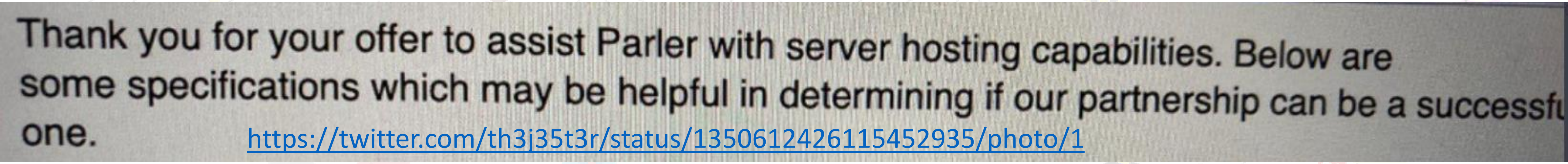
oren@ravendb.net



Parler social network

- January 8
  - Google remove from play store
  - Apple removed from app store
- January 9
  - AWS told Parler to take themselves off in 24 hours

- This talk isn’t about Parler
  - It is about the technical details



Thank you for your offer to assist Parler with server hosting capabilities. Below are some specifications which may be helpful in determining if our partnership can be a successfu one.

https://twitter.com/th3j35t3r/status/1350612426115452935/photo/1

Parler “secret” sauce

Purpose?	Specs	How many?	Total
Scylla (Cassandra clone)	64 cores & 512 GB 14 TB NVMe	40 nodes	2,560 cores 20 TB RAM 560 TB NVMe disks
PostgreSQL	96 cores 768 GB RAM 4 TB NVMe	100	9,600 cores 75 TB RAM 400 TB NVMe disks
Application servers	16 cores 64 GB	400	6,400 cores 25 TB RAM

To buy? 6 million USD

At AWS? 300K + / month

Break down the numbers

- AWS @ 300,000 a month
- ~13 million users by Jan 2021
- Per user cost 2.3¢ a month
- Make sense?

- If you are a VC, maybe?
- For a tech guy?

Can we do better? I certainly hope so!

- Twitter has ~340 million users
- 42% of users are active
- 10% of users generate 80% of content
- ~10K tweets / sec
- Max tweets / sec: 143,199 @ 2013

- 0.06% of accounts have > 20K followers
- 2.12% has > 1K followers
- Common user: 2 tweets a month

Goal in numbers: 50M users, 50% engagement

- 50,000,000 users
- 25,000,000 post each month
- 50% passive daily users (readers, not posters)
- 80% users under 5 posts a month
- 20% < 300 posts a month
- 1% daily active (max 500 posts / day)

- 50K users @ 150 posts / day = 225 M posts per month
- 5M users @ 300 post / month = 1.5 B posts per month
- 20M users @ 5 / month = 100 M posts per month

- **Total: 1.745 billion posts / month**

What is the load?

- **Total: 1.745 billion posts / month**
- 2,400,000 posts / hour
- 670 posts / second

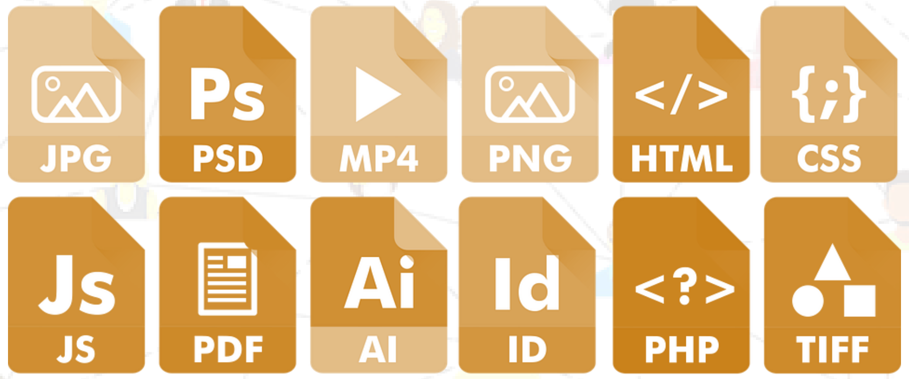
- Not consistent load

- Superbowl 2016:
  - Total 17 Million tweets
  - 152,000 tweets / minute at max

- Highly variable on events
  - Low hundreds to tens of thousands can happen in an instant.

Out of scope: Anything but text

- Ignoring images & videos
- Re-encoding videos
- Stripping EXIF metadata
- Sending large files to our users



- That is the CDN’s issue, nothing particularly interesting here
  - Still *important*, but mostly a solved problem

Service level agreement: Capacity

- 5,000 posts / second
  - 18 millions / hour
  - 432 millions / day
- P99.99 at < 200 milliseconds

We need to design, upfront

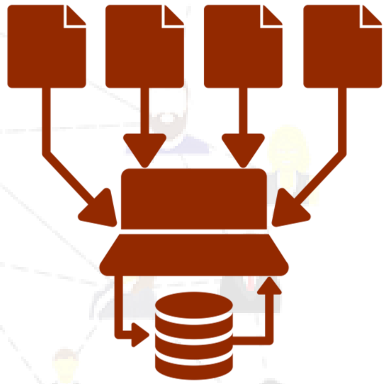
- Twitter: It’s all a tweet
  - @mentions
  - Retweets
  - Just properties / metadata
- Facebook: Posts & Replies
- Homogenous is easier, let’s go with that

- Favor reads over writes
- Read performance paramount
  - 100 ms has impact
- Favor user experience over guarantees
  - Also known as: cheating

Infrastructure components

- Core for everything
- Selected for:
  - Ease of scale
  - Ease of distribution
- Accepted limitation inherent to using them
- If it’s hard, let’s *not* do that

- Content Delivery Network
- S3 Compatible Object Store
- Distributed Key/Value store
- Distributed Queue



Adding a new post

- POST /posts/new {… }
  - If offline, add to local queue?
- Add to queue
- Return to client
- User see the post immediately posted
  - Client side behavior

Why?

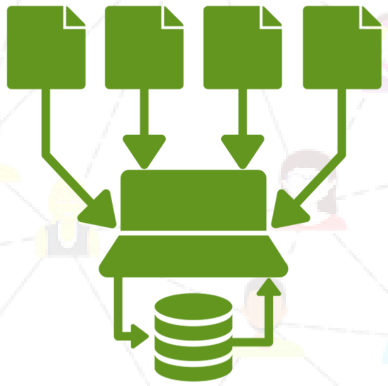
Core concept: Don’t commit to doing hard things

- Adding to a queue is simple
- Trivial to scale accepting to queue
- Gives us *time* to process
- Flatten out spikes in traffic

- Key observation: **What is the SLA for *other users* to see my posts?**
- The user who posted already see it “live”.

Design decisions to make our life easier

- All data is Post (json structure)
- Post has Id, only fetched by id
  - Meaning we have a simple scheme
  - Easy to scale, duplicate, share
- Posts are immutable
  - No edits
  - Can delete, however
    - “soft” delete, mind



- *Much* simpler data model to scale
- Can replicate data seamlessly (not need to figure conflicts)
  - Delete always win
- Key/Value is simple to shard / scale / replicate
- *Cheap* to operate

Identifier generation: How twitter does it?

- 63 bits number (signed long): Snowflake

Seq	Machine Id	Timestamp
12 bits	10 bits	41 bits
To generate multiple ids per millisecond	Max 1,024 machines	Milliseconds from epoch

  - Generates mostly sequential ids
  - Important for many optimizations
  - Can be generated independently on multiple machines
- **Explicitly**: Ids aren't secret
  - You can guess them… https://twitter.com/quinetweet/status/1309951041321013248

Identifier generation: Thoughts…

- Predictable id can be problematic:
  - Scanning attacks
  - Parler grab – 70 TB(!) of data
    - Distributed effort by activists
    - https://github.com/ArchiveTeam/parler-grab
- Even public posts are sensitive in bulk
- Requires complex setup:
  - Authentication
  - Rate limiting
- Can we avoid this?

What do we need from identifier?

24 bits	20 bits	20 bits	64 bits
Time in minutes (cycle @ 31 years)	Machine id (Enough for 1M)	Flags Public post, others	Crypto random

- Unique
- Nice to have:
  - Sorted
  - Don’t actually have to require it, though
- Unique ids:
  - Machine readable
  - Use Base58 ids: **4rzDn3T7Z2FLXqYxDTg5qqsbx57DRdkTvQTBcibc4da5**
  - 16 bytes, lots of space to play with
- Guessing a single item, probably fine
- Note, public post is in the *identifier directly*

Permissions at the architecture level

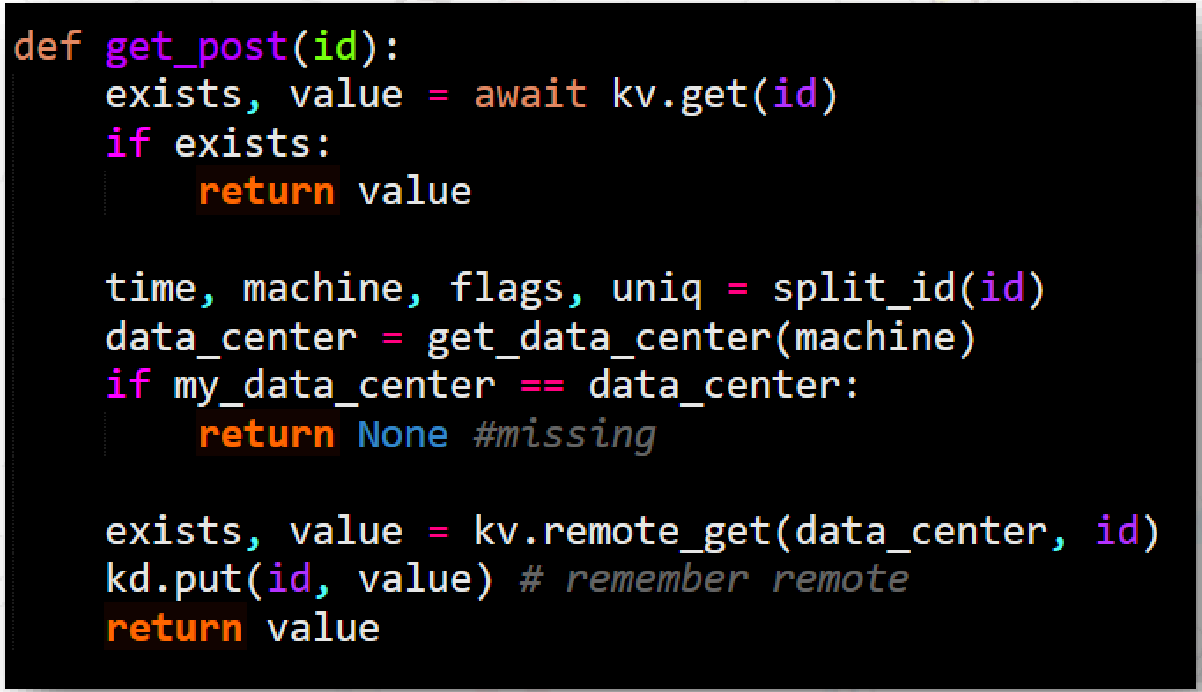
- Can assume most posts are public
- As a flag in the id, can natively skip permissions checks most cases
- If private, then we need to check if user can read post
  - Important optimization for vast majority of posts
- All popular posts are public, by definition
  - Simply the process significantly
- At the architecture level, making permissions explicit

Reading posts

- GET /read?post=**4rzDn3T7Z2FLXqYxDTg5qqsbx57DRdkTvQTBcibc4da5**&post=**7YWFiYdz8xcHE6jhQqQrUceyvLSkcDqpMq74i48kjxCV**
  - Batch oriented
  - Limit number of items in one shot
- Media – directly via CDN, not through our API
- Posts are in Key/Value store – always queried by id
  - Super cheap to manage / scale

Data distribution & immutability

- Posts never change
  - But may be deleted (One way)
- Easy to propagate changes

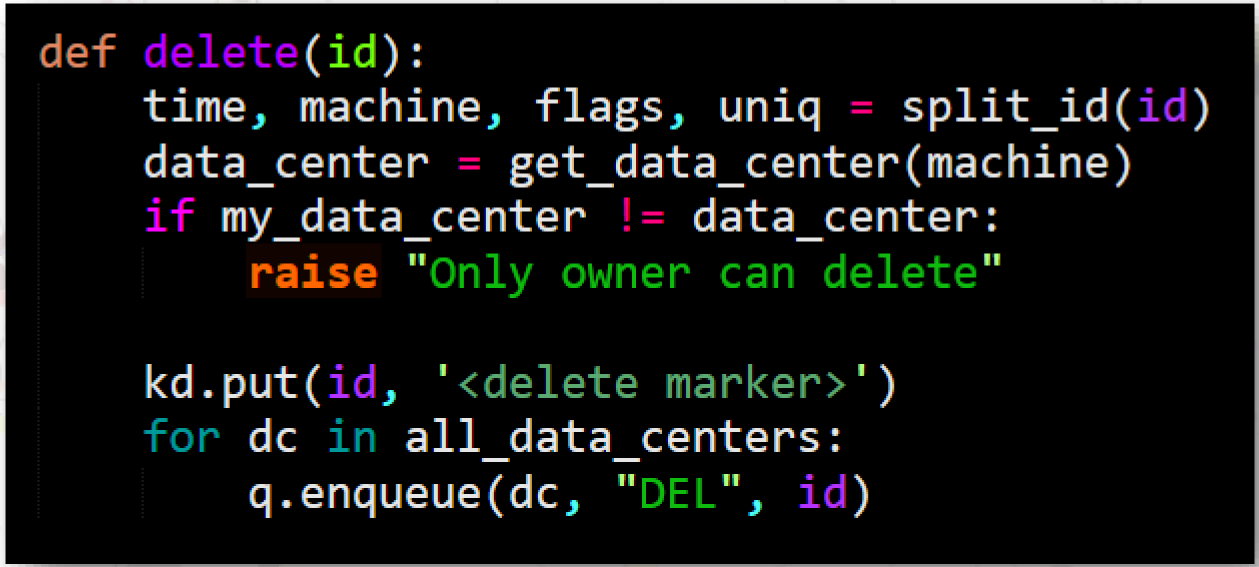


```
def get_post(id):
    exists, value = await kv.get(id)
    if exists:
        return value

    time, machine, flags, uniq = split_id(id)
    data_center = get_data_center(machine)
    if my_data_center == data_center:
        return None #missing

    exists, value = kv.remote_get(data_center, id)
    kd.put(id, value) # remember remote
    return value
```

Most posts are local

Handling deletes

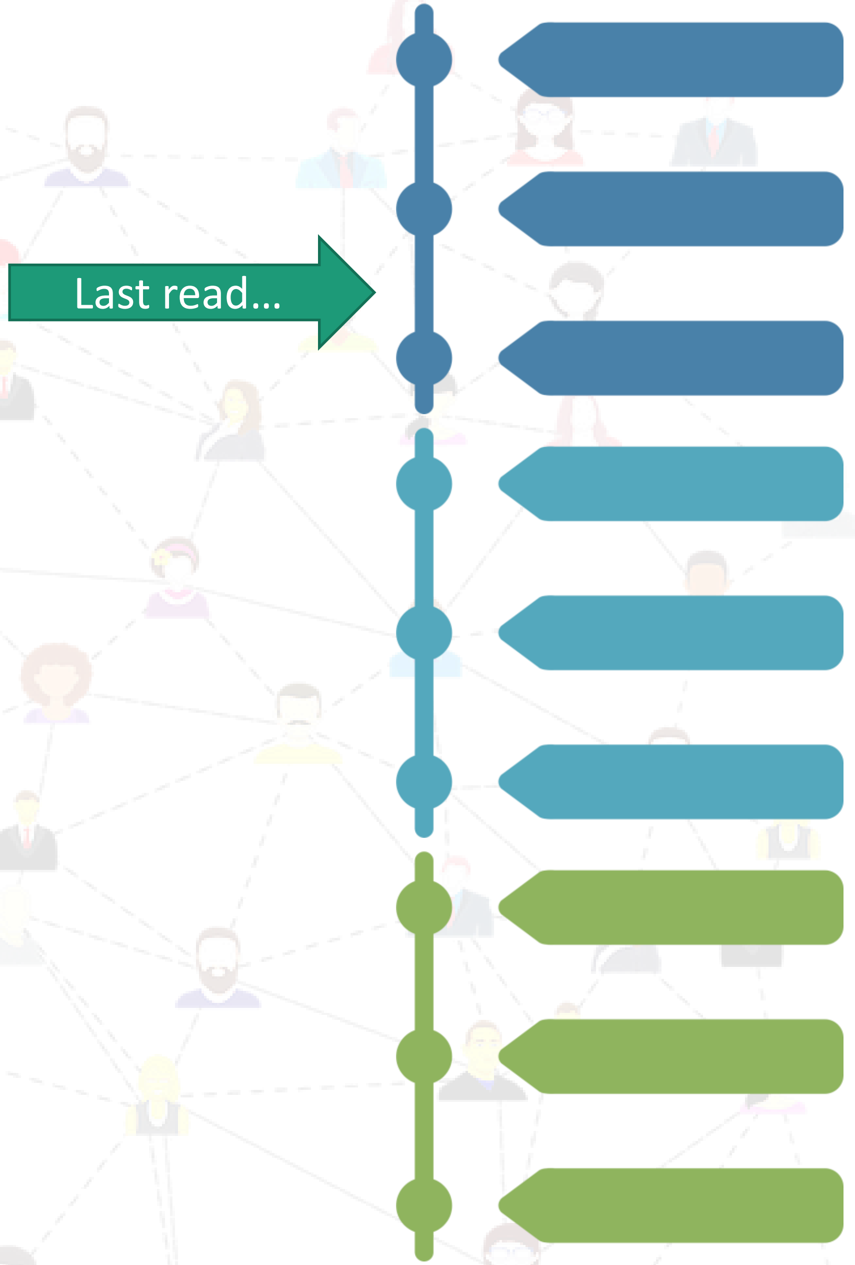
- Synchronizing state across distributed regions?
  - Complex
- Delete on owner datacenter
- Send delete command to all regions
- Assume that deletes are *rare*



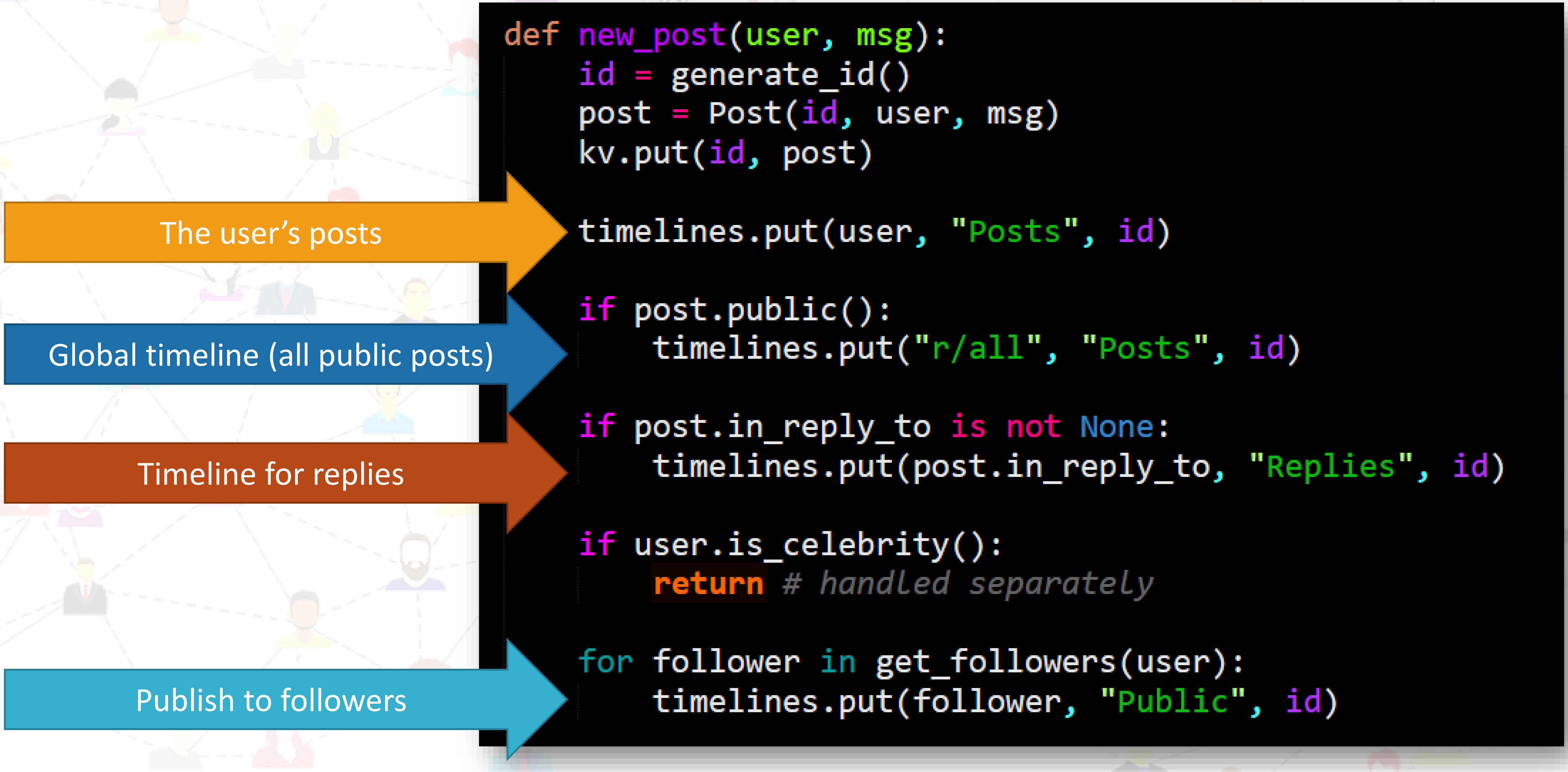
```
def delete(id):
    time, machine, flags, uniq = split_id(id)
    data_center = get_data_center(machine)
    if my_data_center != data_center:
        raise "Only owner can delete"

    kd.put(id, '<delete marker>')
    for dc in all_data_centers:
        q.enqueue(dc, "DEL", id)
```

What about the timeline?

- A single post, easy
- What about timeline view?
- The timeline abstraction & the last read location
    - Similar to Kafka's logs
- The public timeline
    - All activities of a user
- The private timeline
    - All the activities of the users followed by the user



Handling a new post

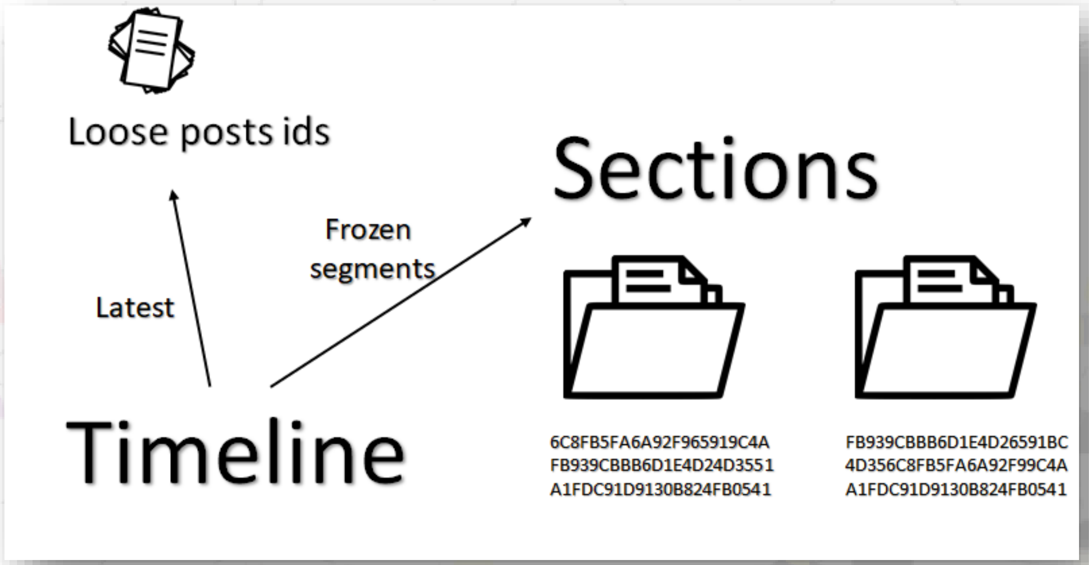


```
def new_post(user, msg):
    id = generate_id()
    post = Post(id, user, msg)
    kv.put(id, post)

    timelines.put(user, "Posts", id)

    if post.public():
        timelines.put("r/all", "Posts", id)

    if post.in_reply_to is not None:
        timelines.put(post.in_reply_to, "Replies", id)

    if user.is_celebrity():
        return # handled separately

    for follower in get_followers(user):
        timelines.put(follower, "Public", id)
```

timelines.put(timeline_id, name, post_id)

- Append only structure
- Going to have a *lot* of those
- Core abstraction

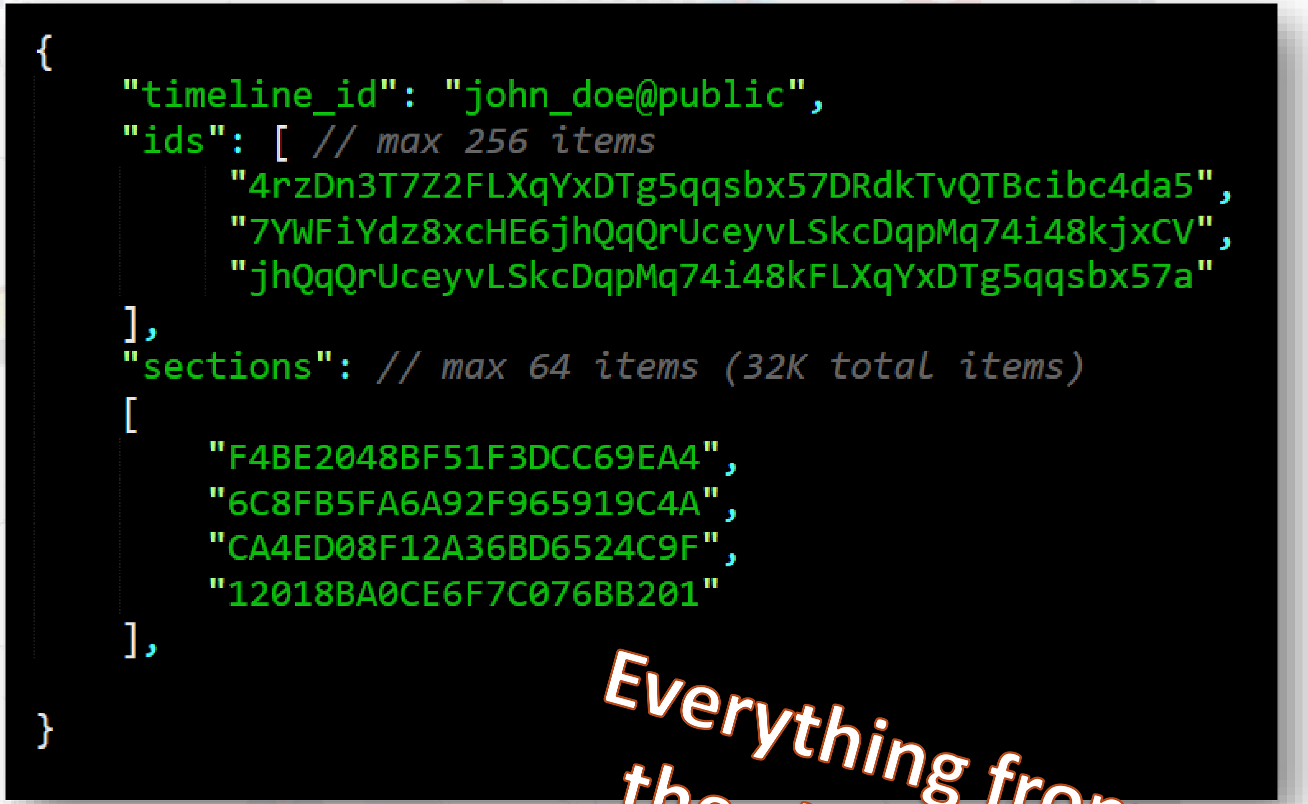
- Once reach certain size (~500 posts), create a section



How that works?

- ~500 post ids @ 20 bytes each
- With compression, assume ~8KB
- Immutable
- Last value points to previous section
- Unique section ids
- Once we have a section, throw on CDN
  - Basically, download a file ☺

- GET /timeline?id=john_doe@public



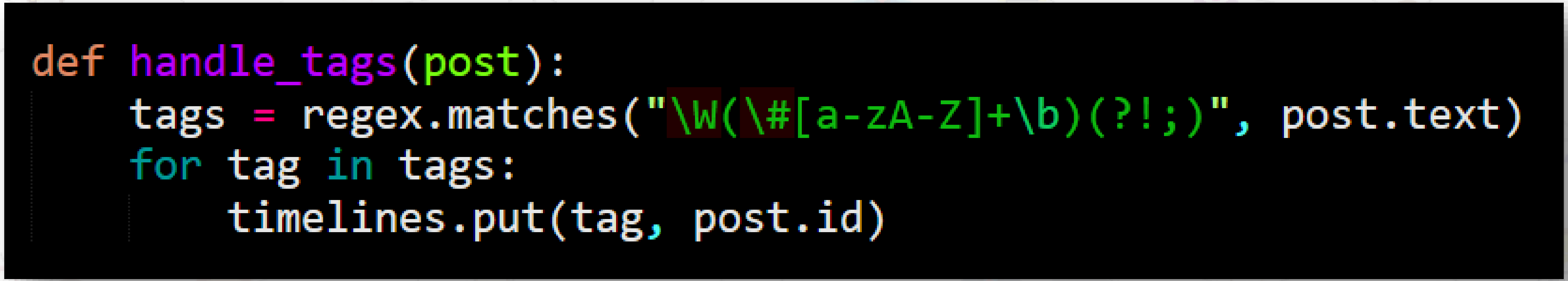
```
{
    "timeline_id": "john_doe@public",
    "ids": [ // max 256 items
          "4rzDn3T7Z2FLXqYxDTg5qqsbx57DRdkTvQTBcibc4da5",
          "7YWFiYdz8xcHE6jhQqQrUceyvLSkcDqpMq74i48kjxCV",
          "jhQqQrUceyvLSkcDqpMq74i48kFLXqYxDTg5qqsbx57a"
    ],
    "sections": // max 64 items (32K total items)
    [
         "F4BE2048BF51F3DCC69EA4",
         "6C8FB5FA6A92F965919C4A",
         "CA4ED08F12A36BD6524C9F",
         "12018BA0CE6F7C076BB201"
    ],

}
```

Everything from the client side

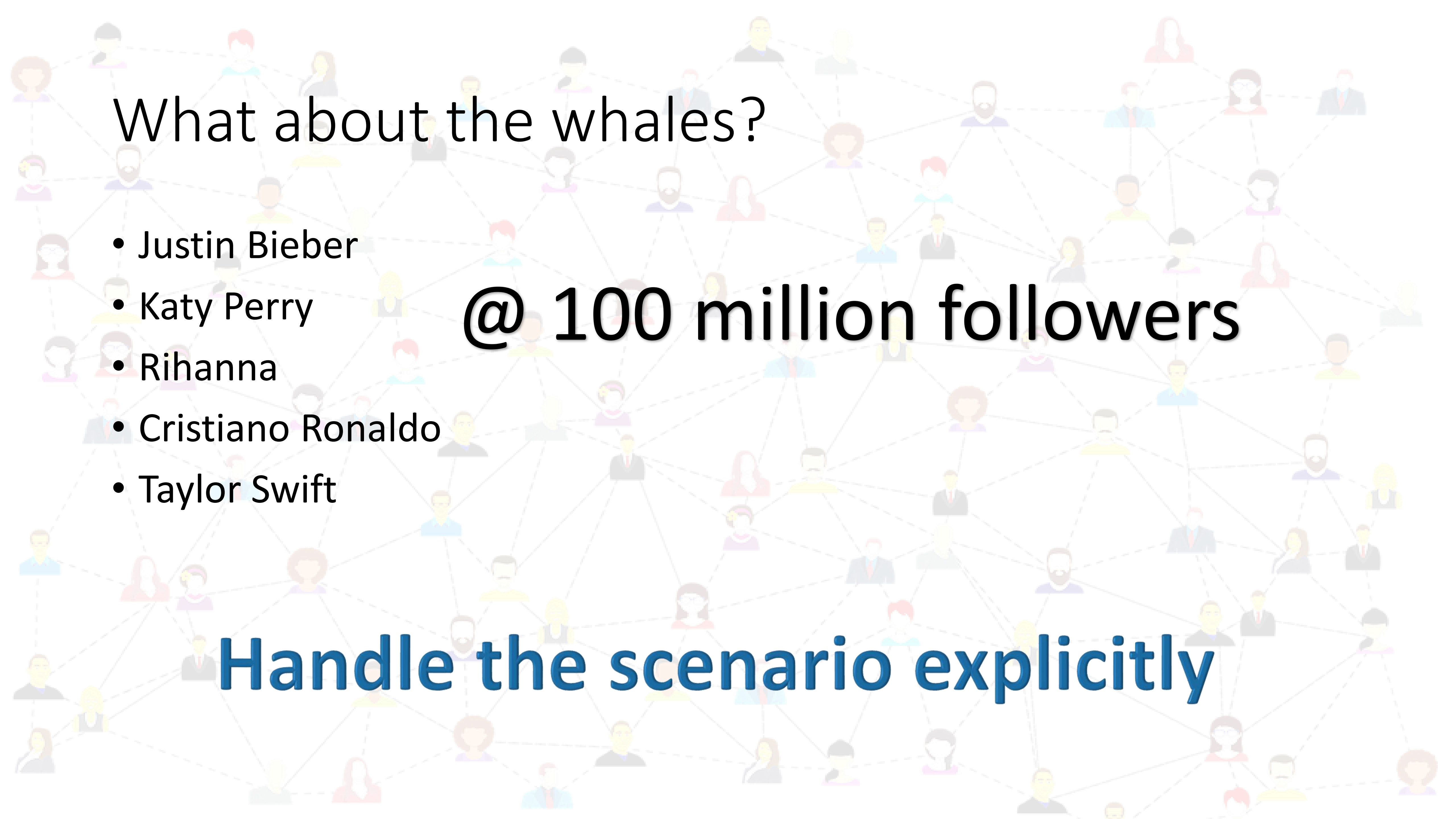
- Read the timeline
- Get recent ids: GET /read?post=…
- Get sections, parse and then load again

Implementing hashtags

- That is just another time line



```python
def handle_tags(post):
    tags = regex.matches("\W(\#[a-zA-Z]+\b)(?!;)", post.text)
    for tag in tags:
        timelines.put(tag, post.id)
```



What about the whales?

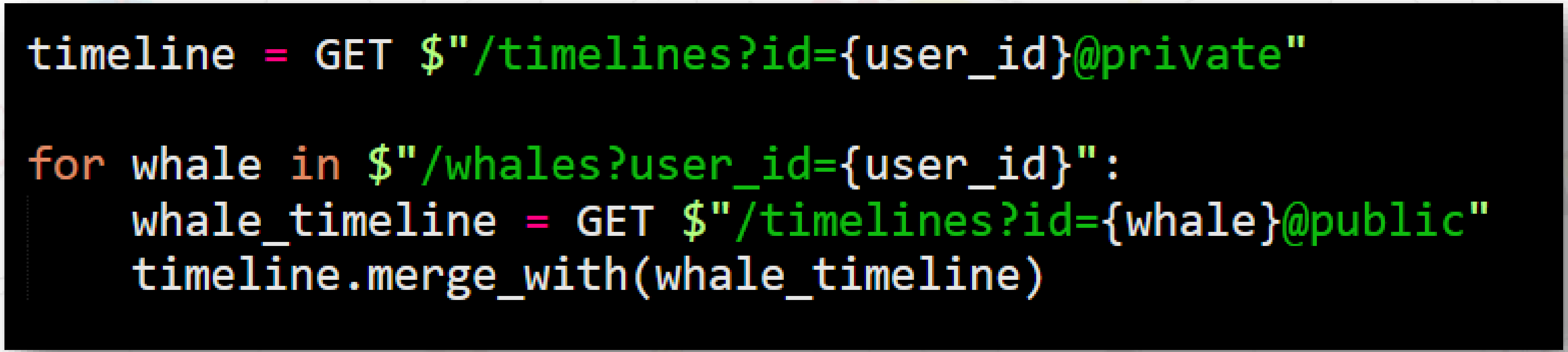
- Justin Bieber
- Katy Perry
- Rihanna
- Cristiano Ronaldo
- Taylor Swift

@ 100 million followers

Handle the scenario explicitly

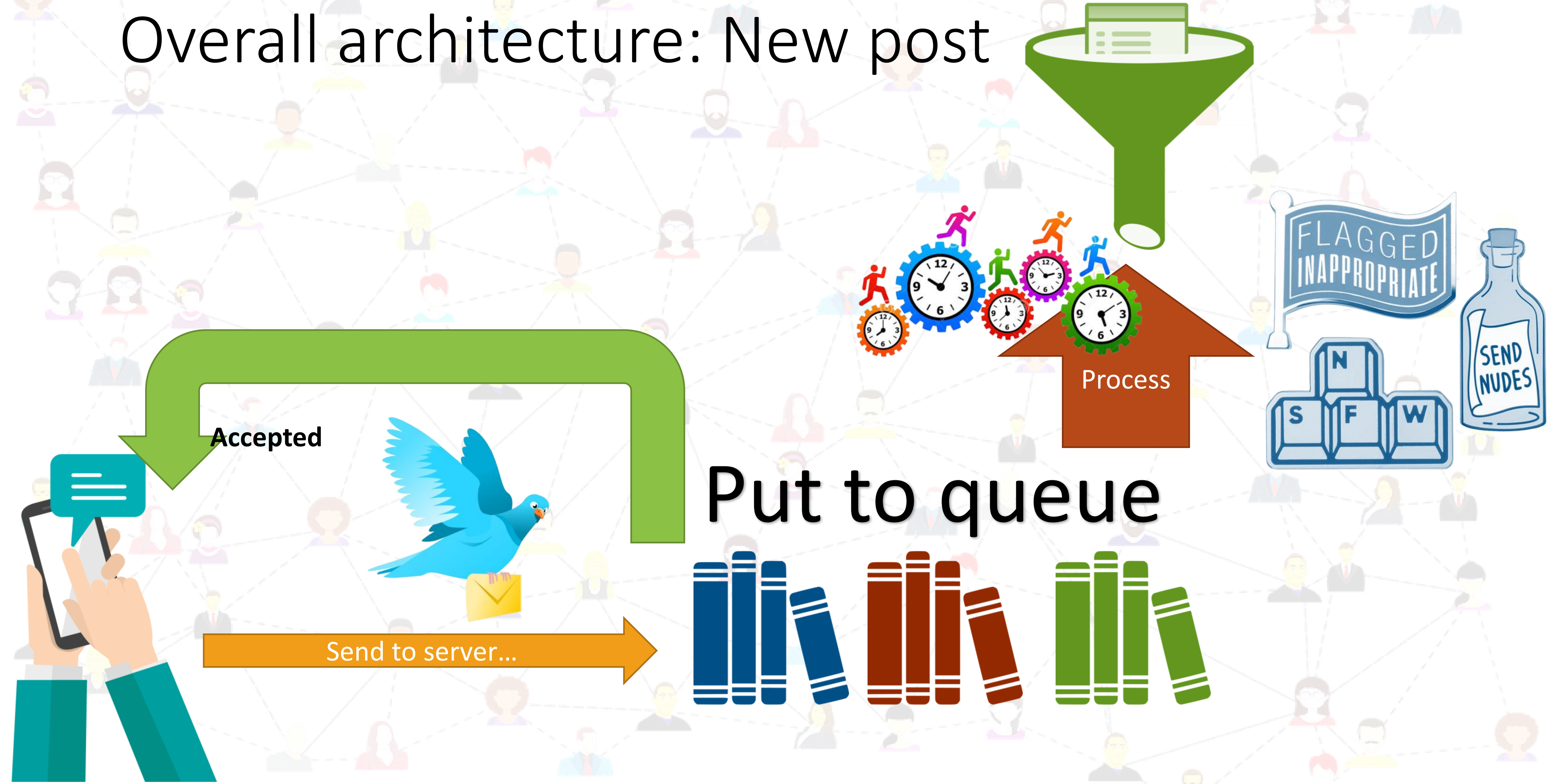
Whale has more than 500 followers

- ~4% of overall users.
- Avoid publishing to *many* timelines
- Instead, go the other way (good cache target)



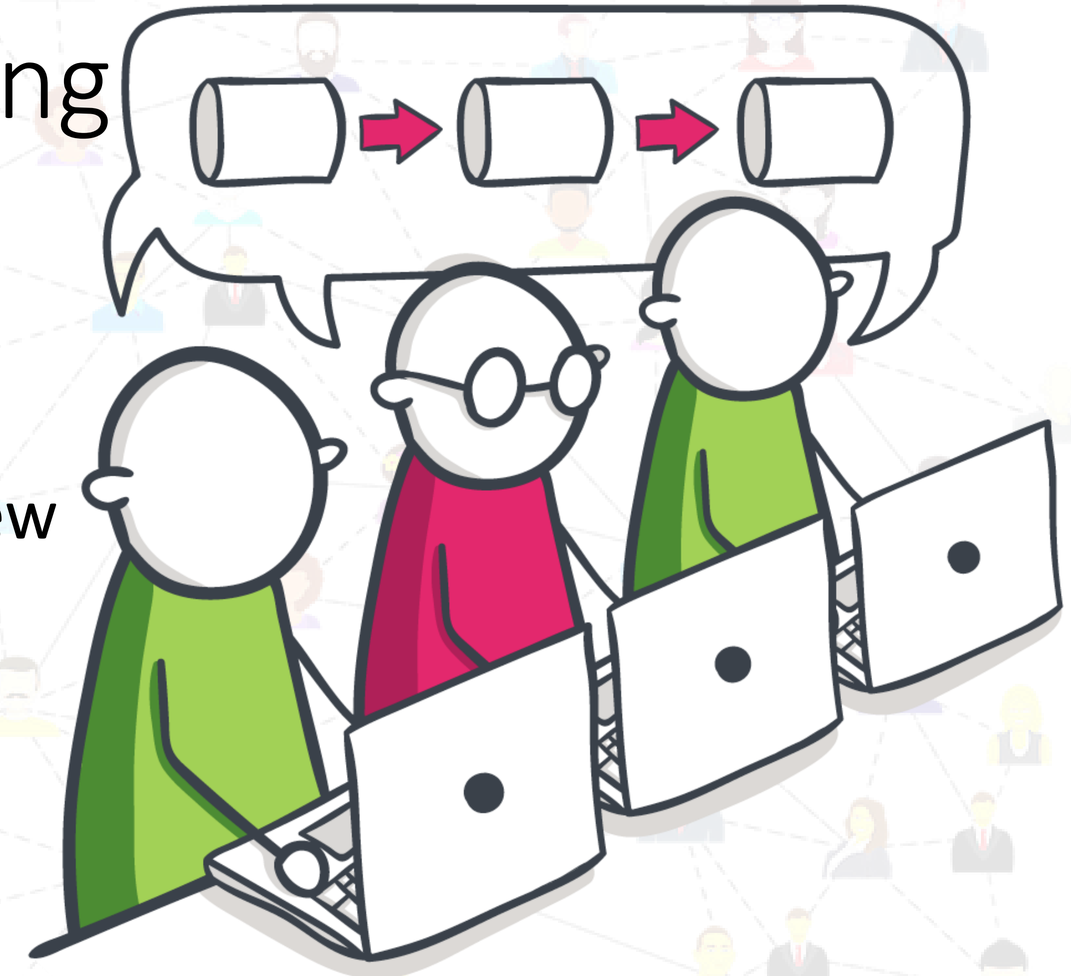
```
timeline = GET $"/timelines?id={user_id}@private"

for whale in $"/whales?user_id={user_id}":
    whale_timeline = GET $"/timelines?id={whale}@public"
    timeline.merge_with(whale_timeline)
```

Overall architecture: New post



Overall architecture: Posting

- Post to data center's Key/Value store
- Publish to timelines
- When timeline buffer is full, publish new segment



Where is the behavior?

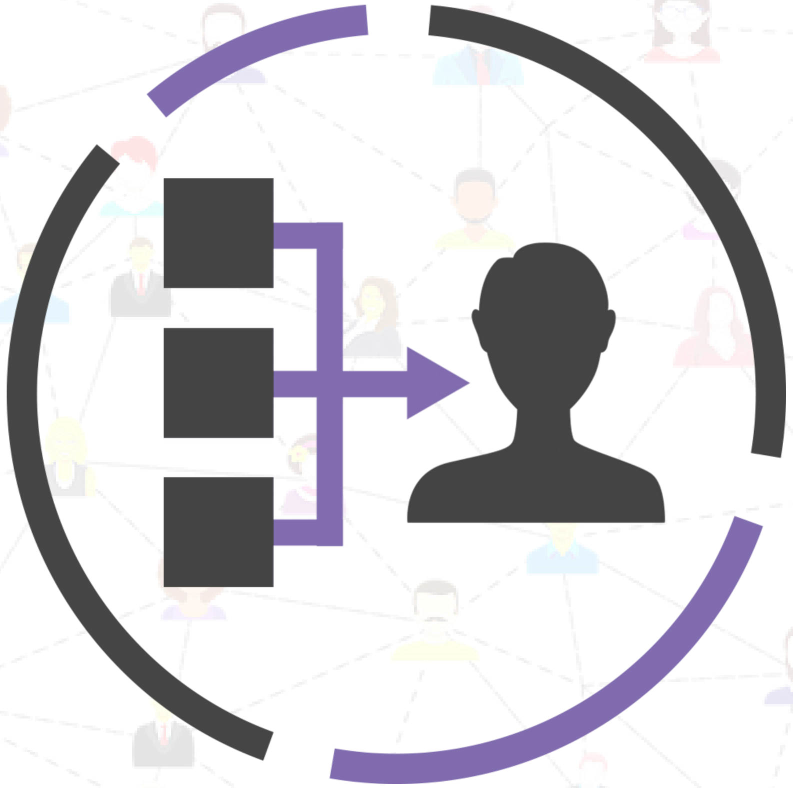
- Select the right timelines to post to
- Can do that after the fact
- Got *time* to manage this

Architecture: Simple tools, client side work

- Components:
  - CDN for media / sections
  - Key/Value
  - Queue
  - Workers
- Require a lot from the client
  - We aren’t paying for that

Client side speed

- Client does a lot of work
  - Fetch timelines
  - Fetch posts
- Additional responsibilities
  - Deduplicate posts
  - Ignore missing posts
- Work in batches, UX is meant to allow high *perceived* speed



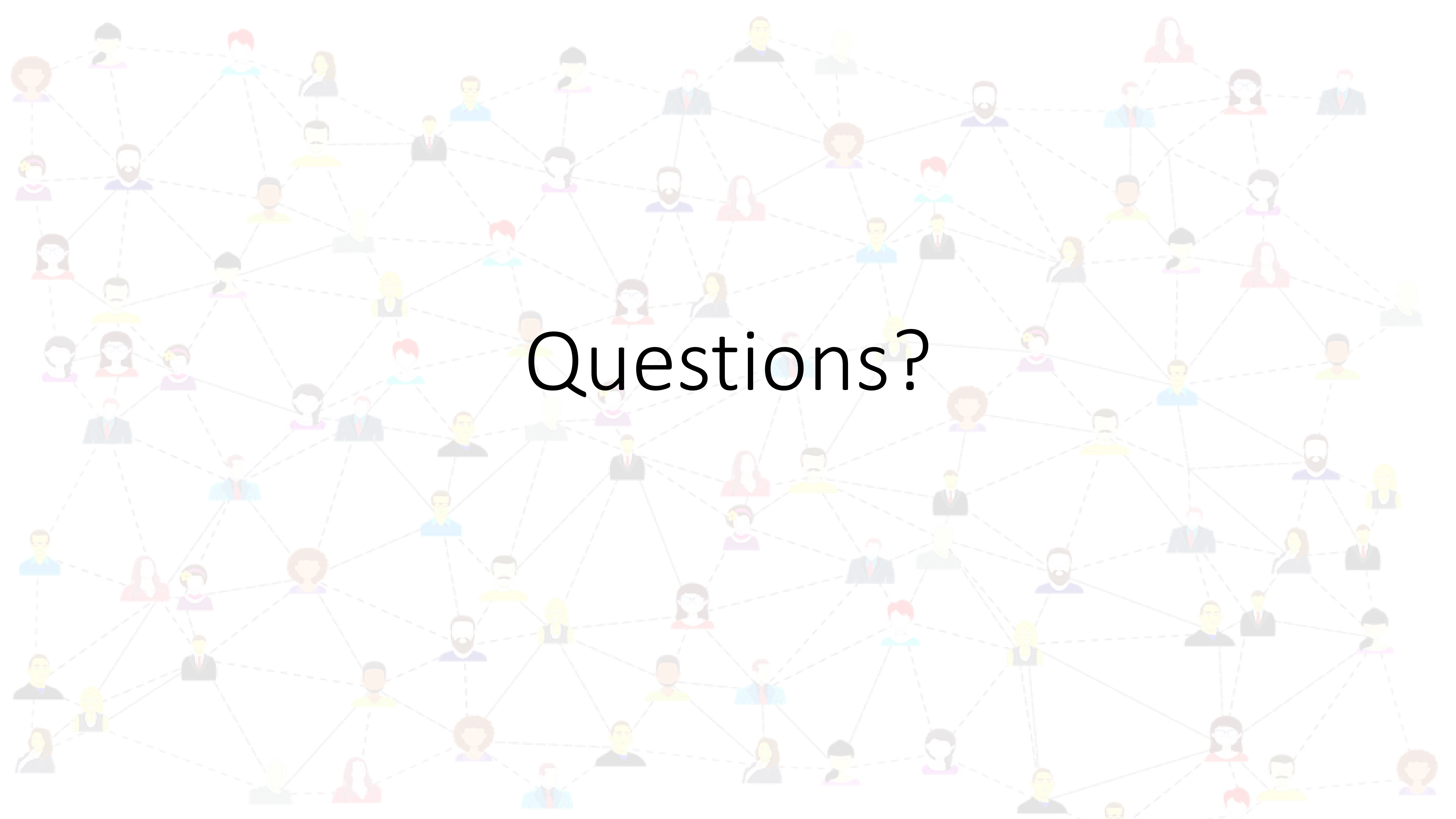
What about other things?

- Monetization – go ask a business major, this is a tech talk
- Tracking – throw into a queue, process in the back end
- Statistics – number of likes, views, etc
  - Dedicated solution (Algorithm: PN Counters)

Key concept: What do we need?

- Critical: What limits can we accept that make our life easier?

- Then set out to design a system *just for that purpose*.
- At scale, don't have agility, cost too much.



Questions?