System Design 101

Let’s build it.

One box at a time.

Hacker Devils

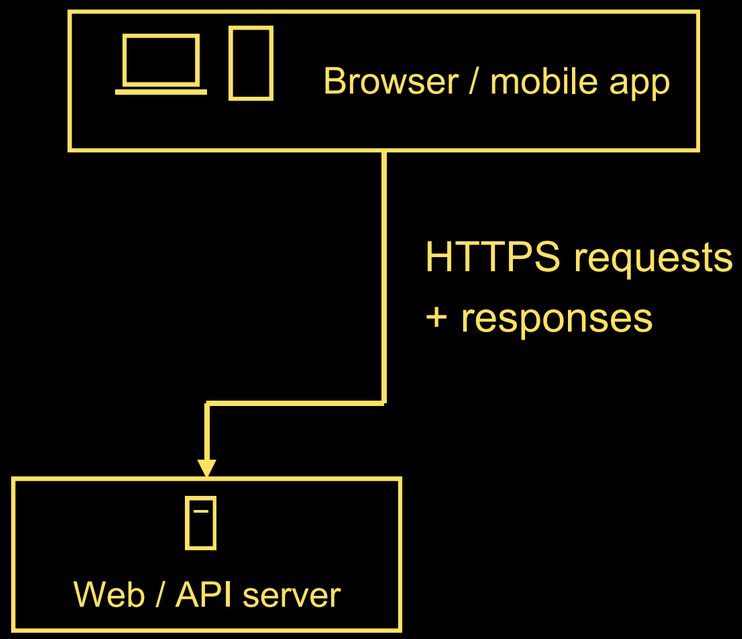


Your first server after
the app gets popular

You open an app. What actually happens?

The app asks.
The server does the work.
The app gets a response.

An API is the agreed
format for that conversation.



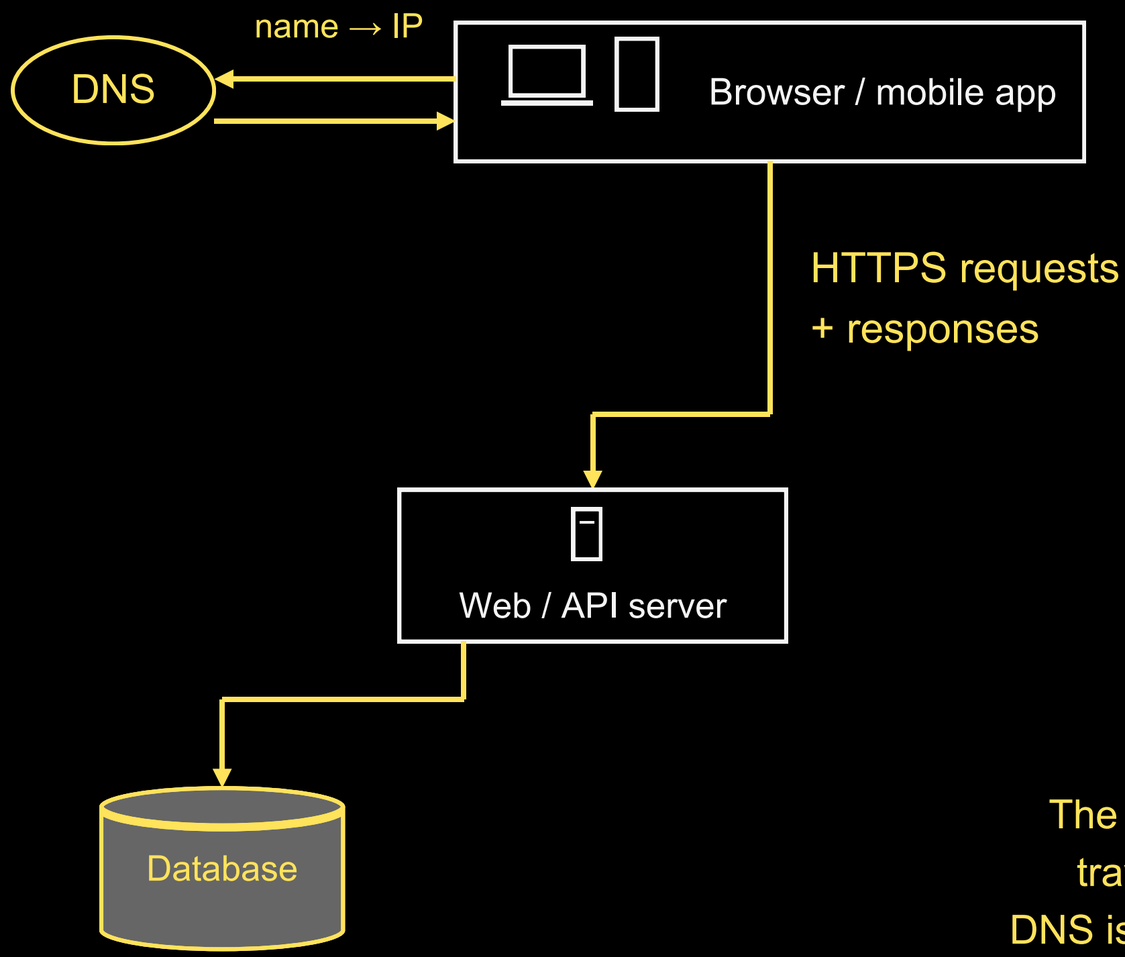
GET /feed
→ a list of posts

First, find the server. Then fetch the data.

DNS finds the IP address.

Packets travel through
Wi-Fi, your ISP and routers.

The server queries the DB
and returns a response.



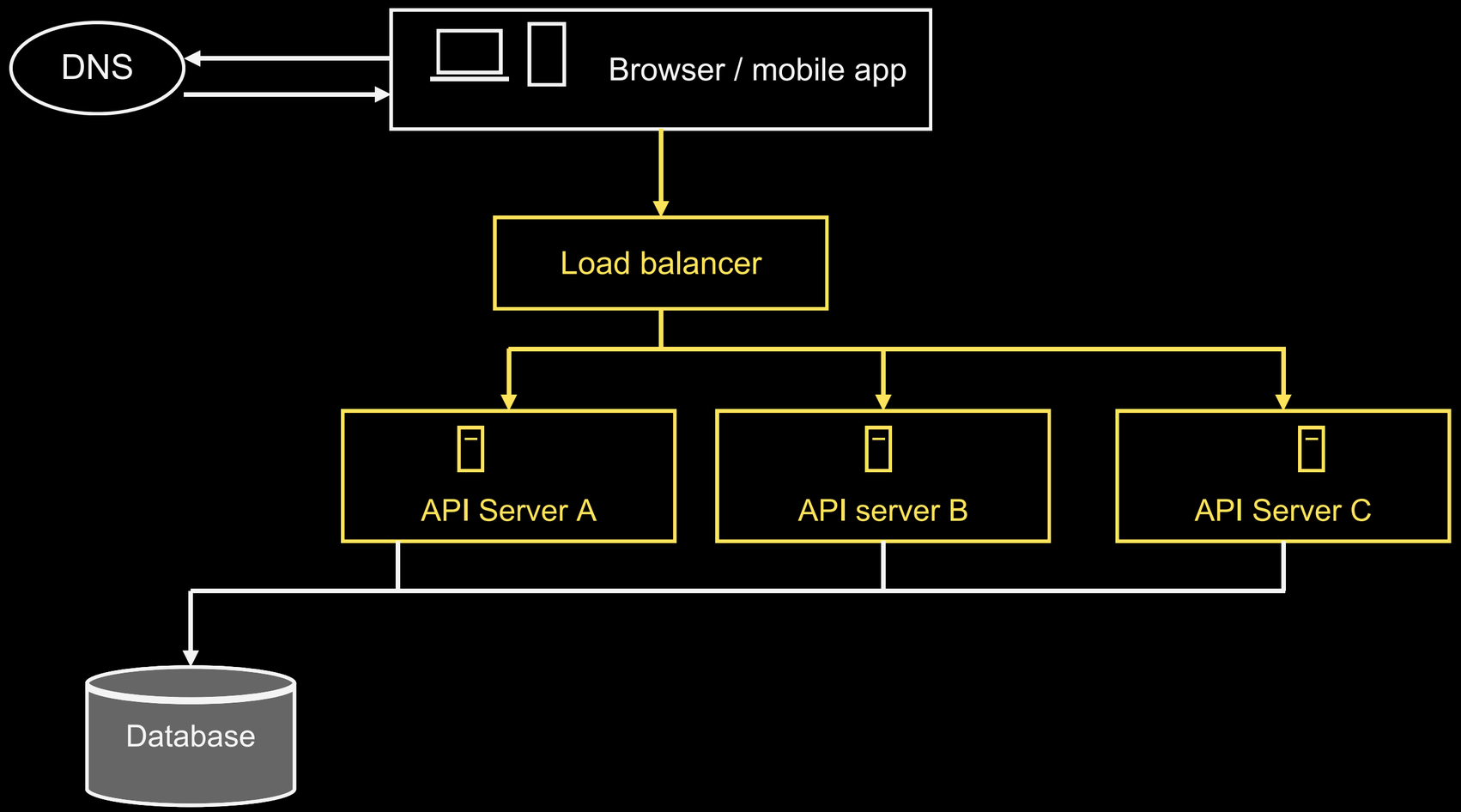
The request does NOT
travel through DNS.
DNS is the address lookup.

More users arrive. One machine is struggling.

Scale up: bigger machine.
More CPU and RAM.

Scale out: more servers.
A load balancer spreads
requests across them.

Same API. More capacity.

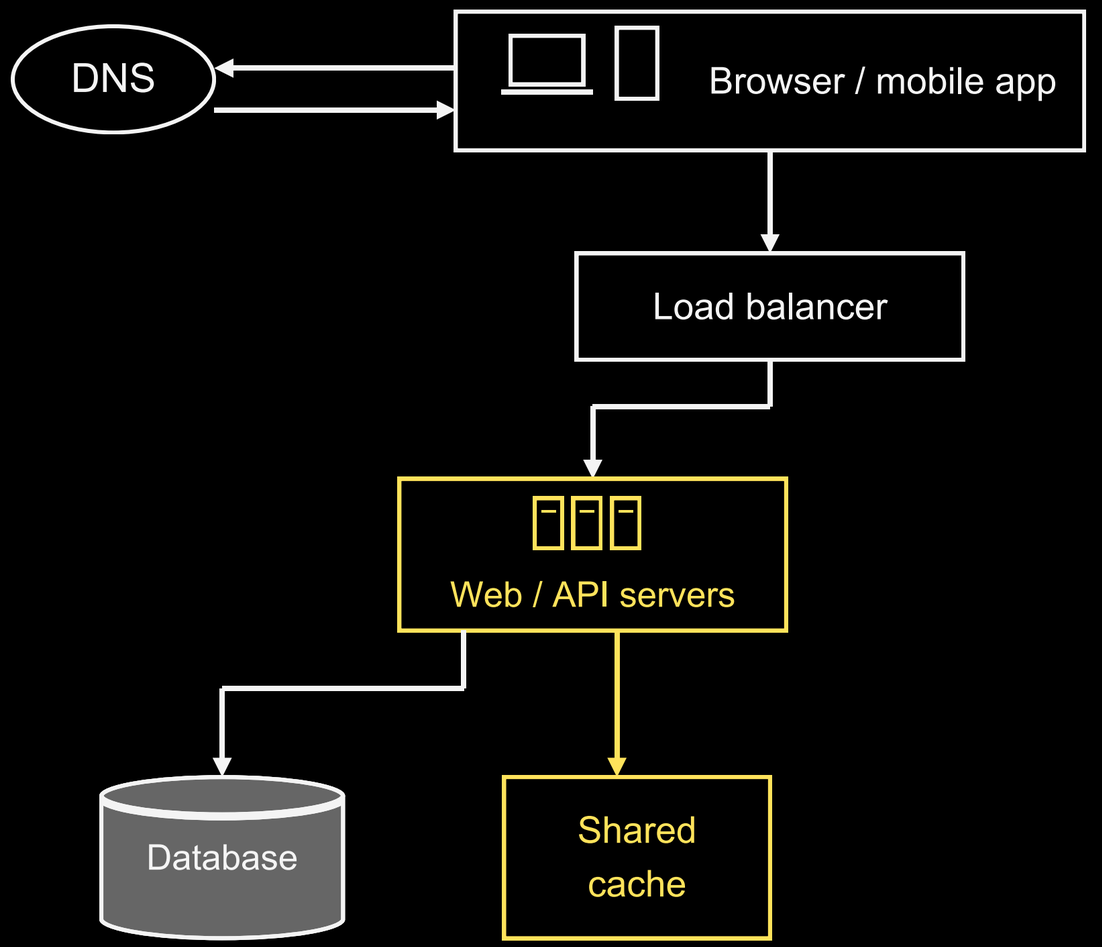


Any server should be able to handle you.

Stateless web tier:
keep sessions outside
one particular server.

Shared cache = fast copies
of frequently used data.

Cache miss? Read the DB.
Then refill the cache.

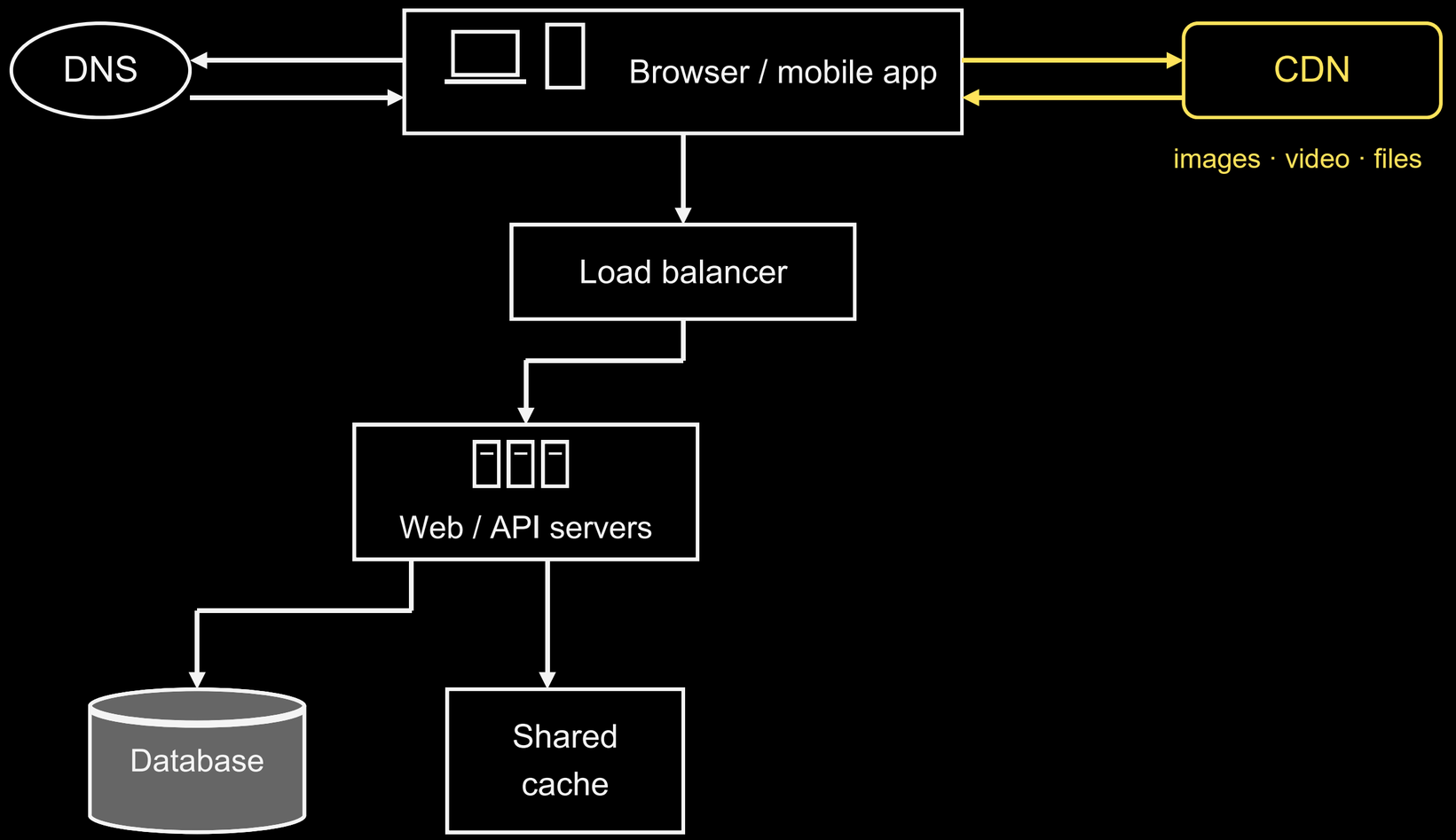


Do images need to cross the world every time?

Add nearby copies of images, video and files.

That network of edge caches is a CDN.

The app still calls our API for its personalized data.

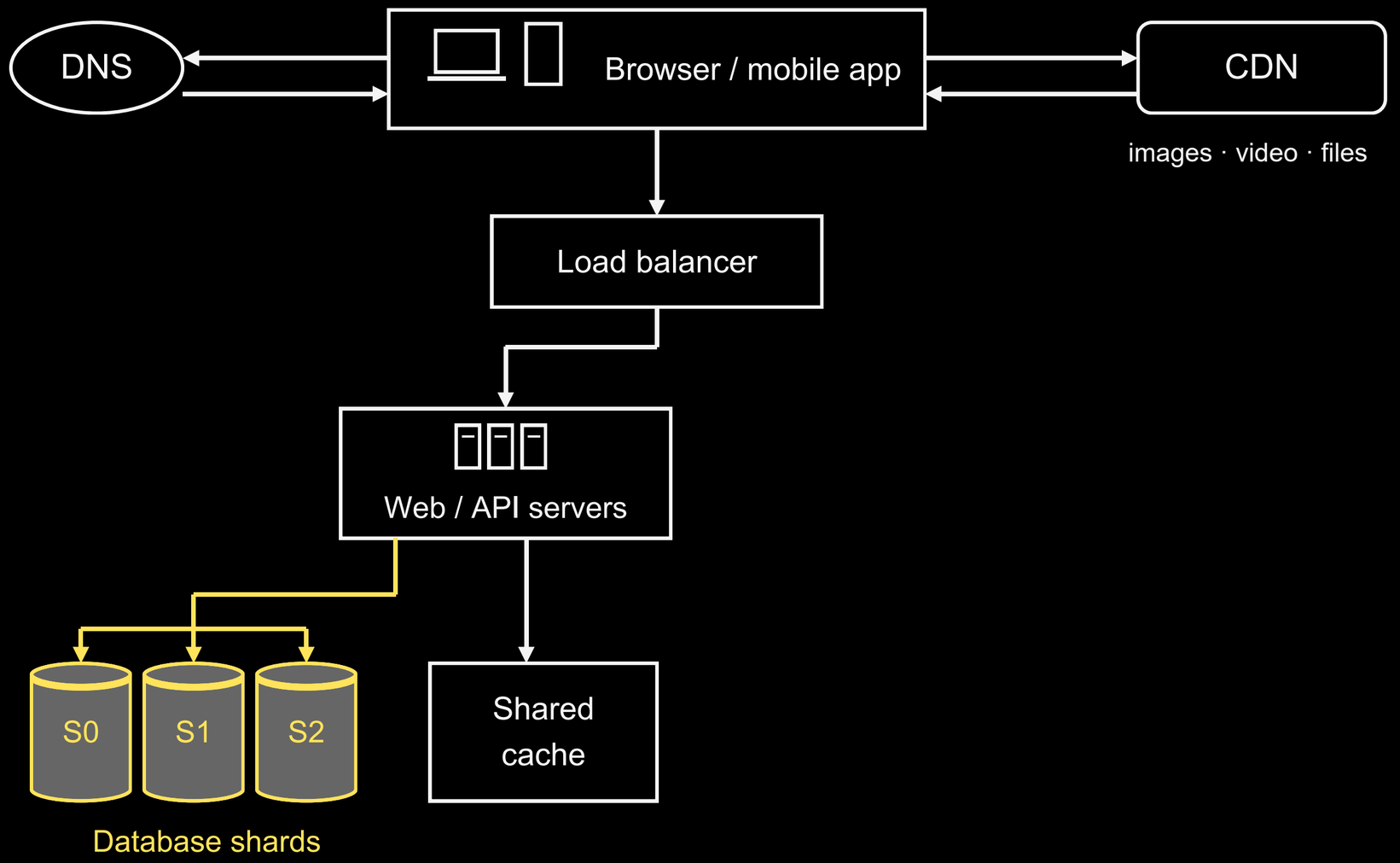


Now the database is the bottleneck.

We can scale the DB up.
Or split its rows across multiple machines.

That is sharding.

Example: user_id % 3 decides the shard.

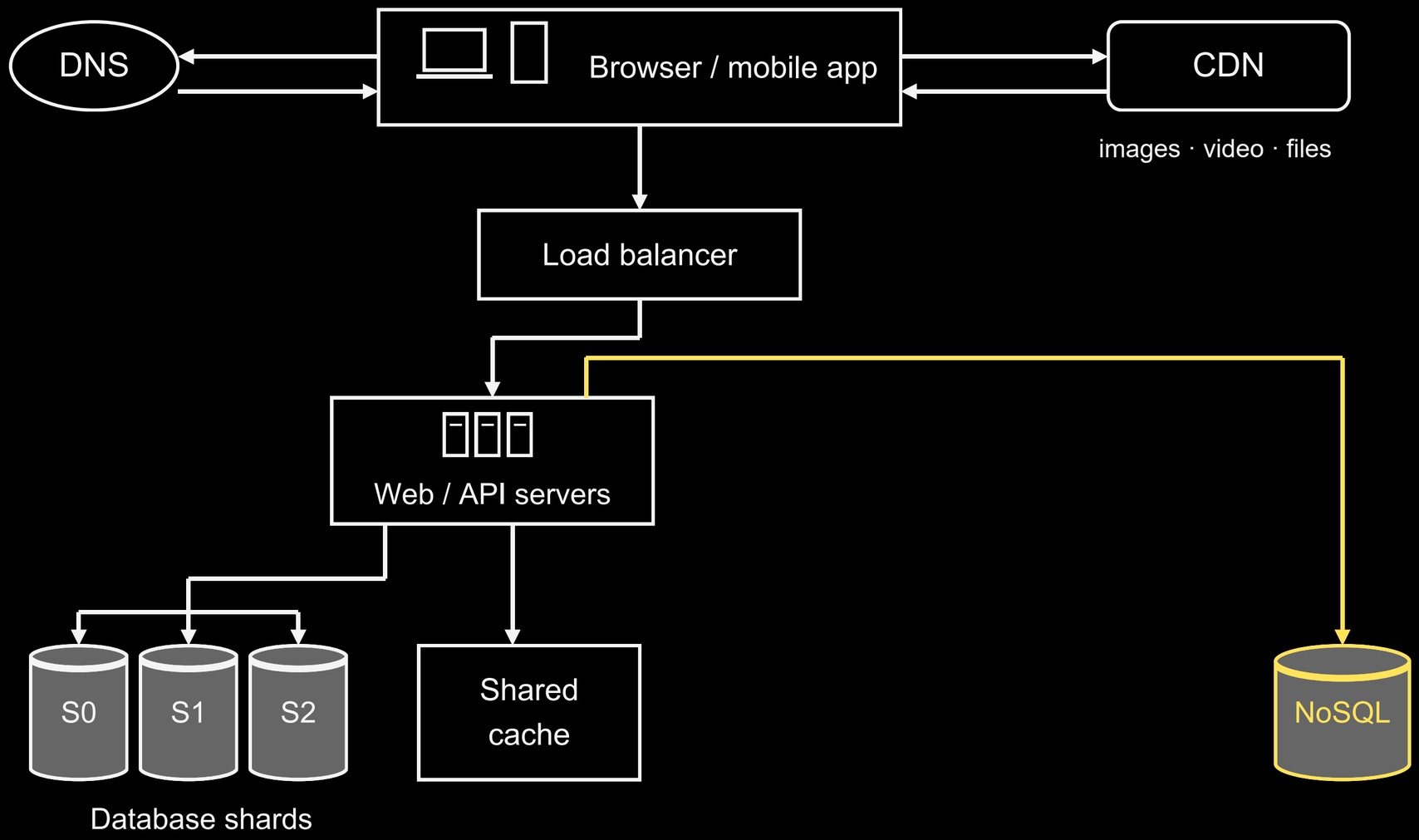


Replication copies data. Sharding splits it.

Different data may need a different store.

NoSQL offers models like key-value, document and graph storage.

Use it when the access pattern fits. Not by default.

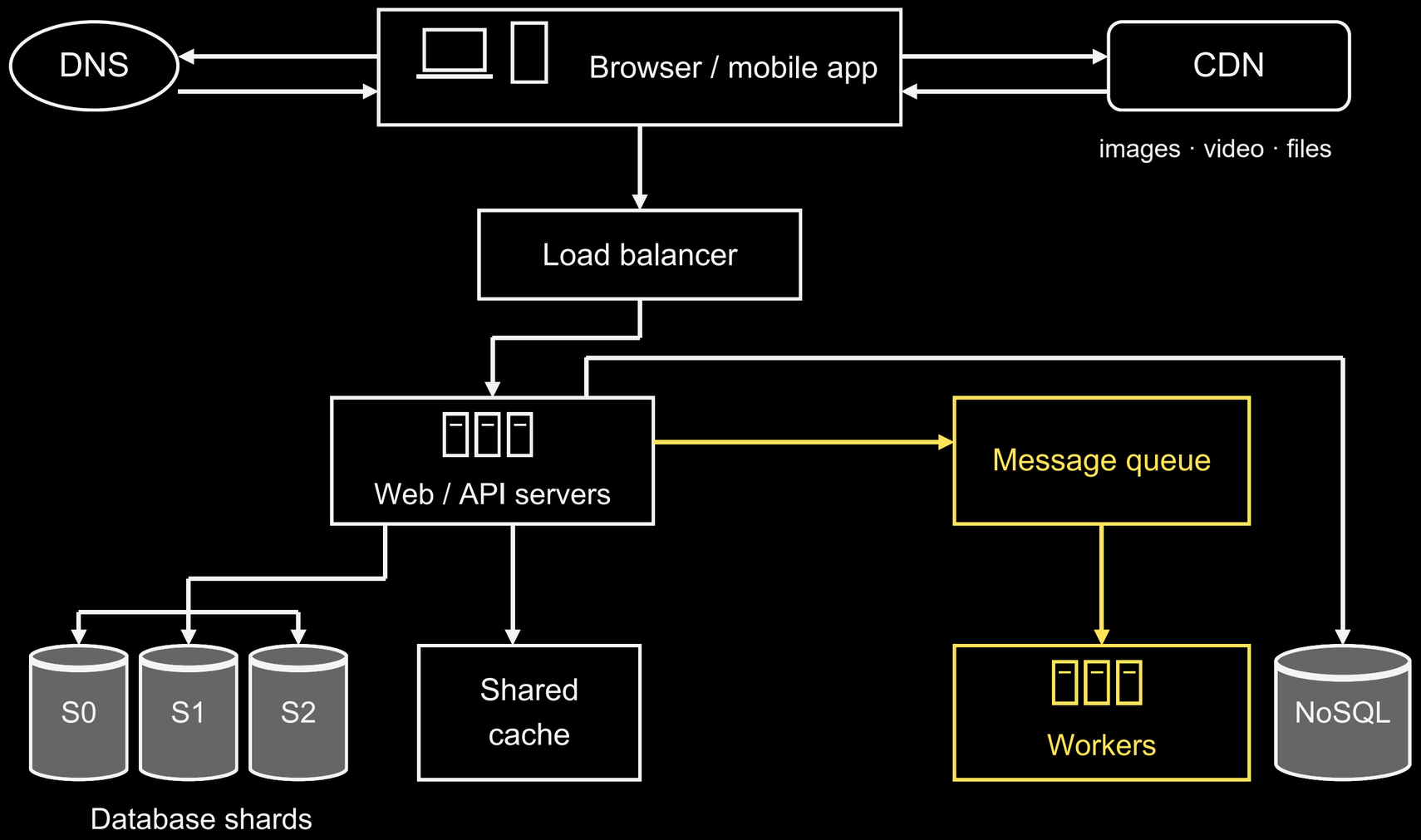


Some work can happen after the response.

Web server → queue
→ background workers.

Examples: resize a photo
or send a notification.

Servers can call services
directly, or send messages
for work to happen later.

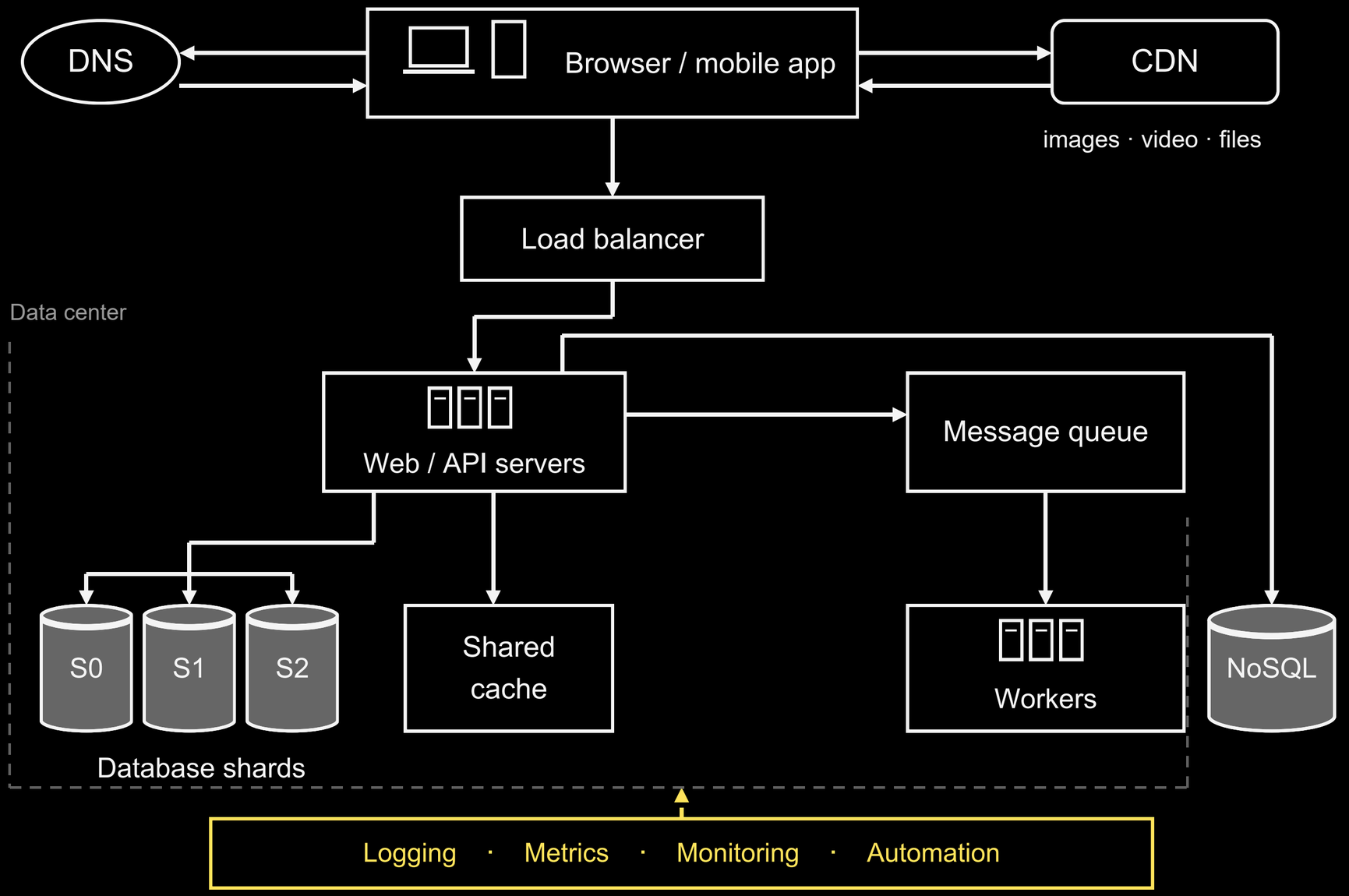


That is the complete picture.

One app request.
A system of connected parts.

Add logs and metrics
so we can see failures.

Start small. Add a block
when a real bottleneck
justifies it.



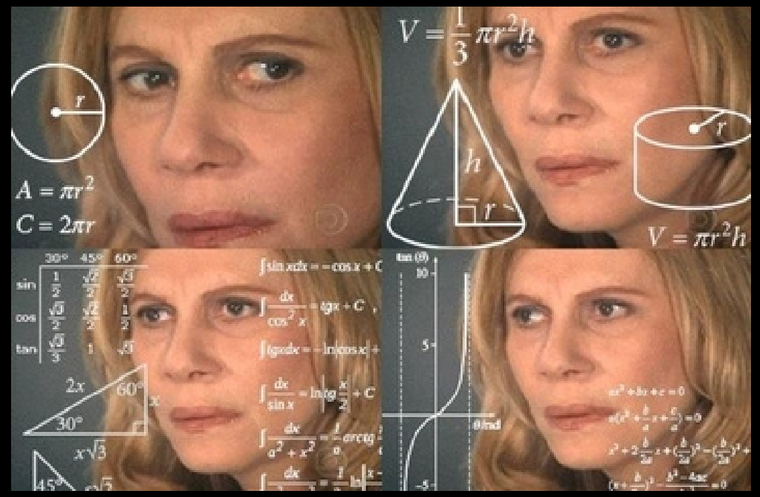
The interview, in three steps.

Let’s watch them on “social media feed publishing”

1. Understand the problem
   and set the scope.

2. Propose a high-level design
   and get buy-in.

3. Deep dive into the parts
   that matter most.



When you draw 12 different databases
before asking one question

Step 1: Understand the problem first.

Candidate asks. Interviewer helps set the scope.

What can users do?
Post, follow and read a feed.

Ranked or chronological?
Newest posts first.

What can wait?
A few seconds of feed delay.

What matters most?
Fast reads; do not lose posts.

What is out of scope?
Ads, ranking and recommendations.

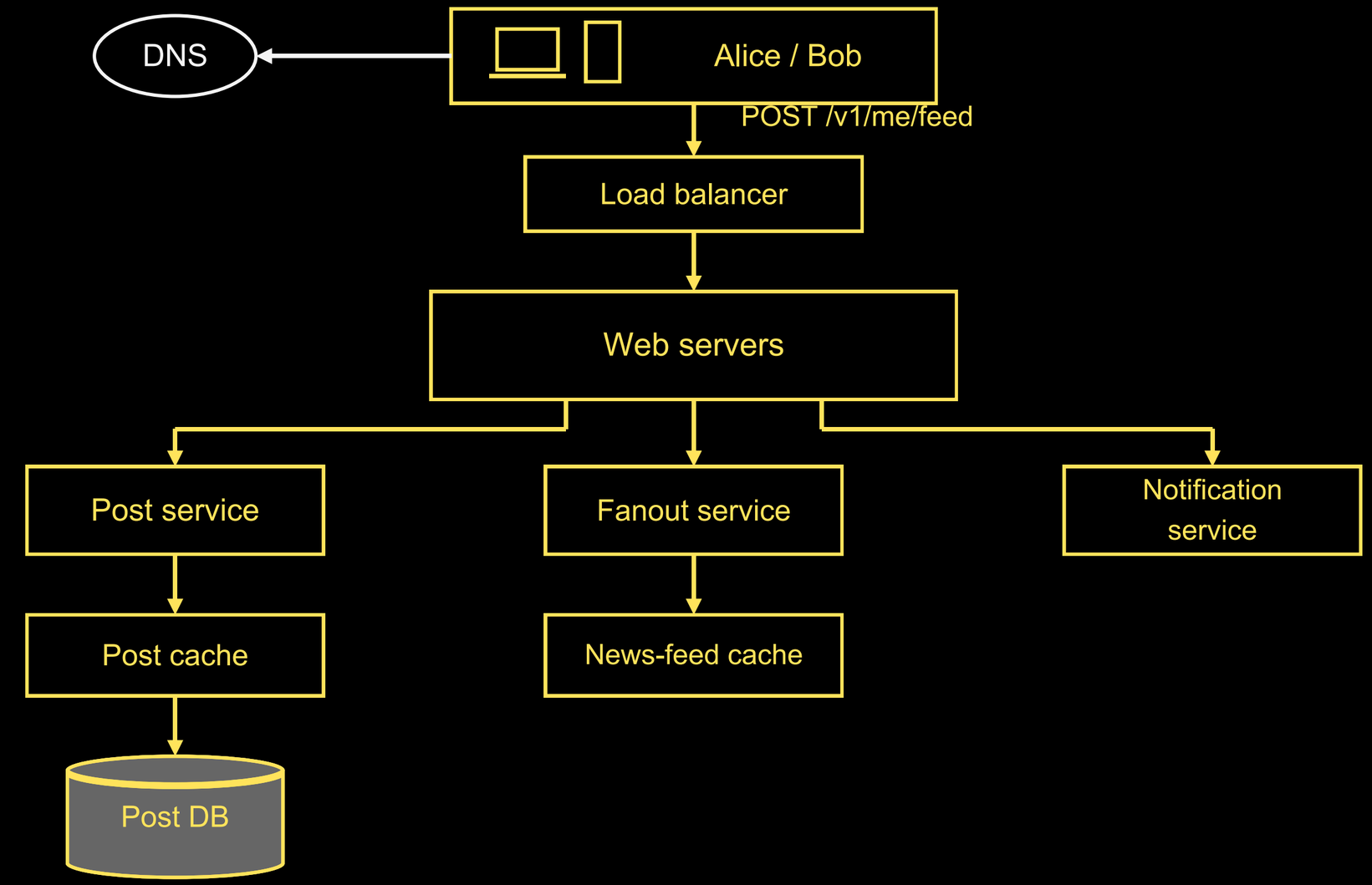
Which paths do we design?
Publish a post + retrieve the feed.

Step 2: Sketch the blueprint. Get buy-in.

Two flows to design.

Flow 1: publishing.
Save the post, push it
to friends' feeds,
notify them.

Draw the boxes, then ask:
"Does this look right?"

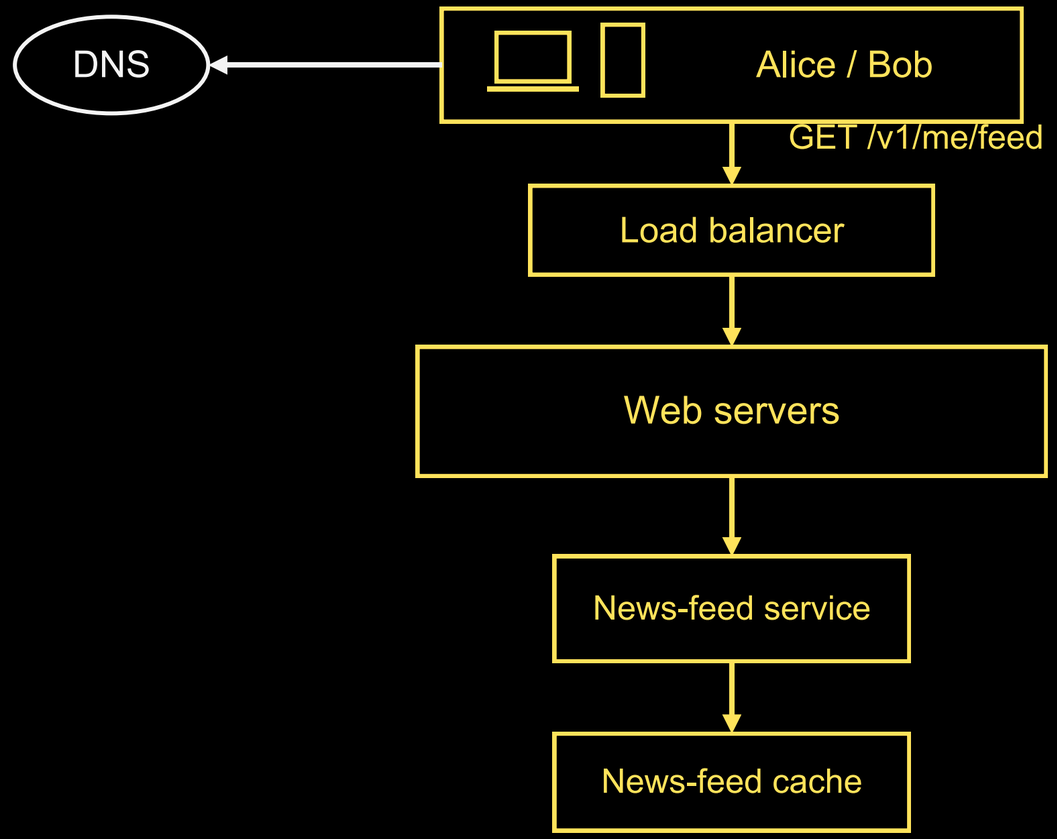


Step 2: The second flow, building the feed.

Flow 2: Bob reads.

The news-feed service gets Bob’s list from the news-feed cache.

Interviewer: “Looks good. Let’s go deeper.”



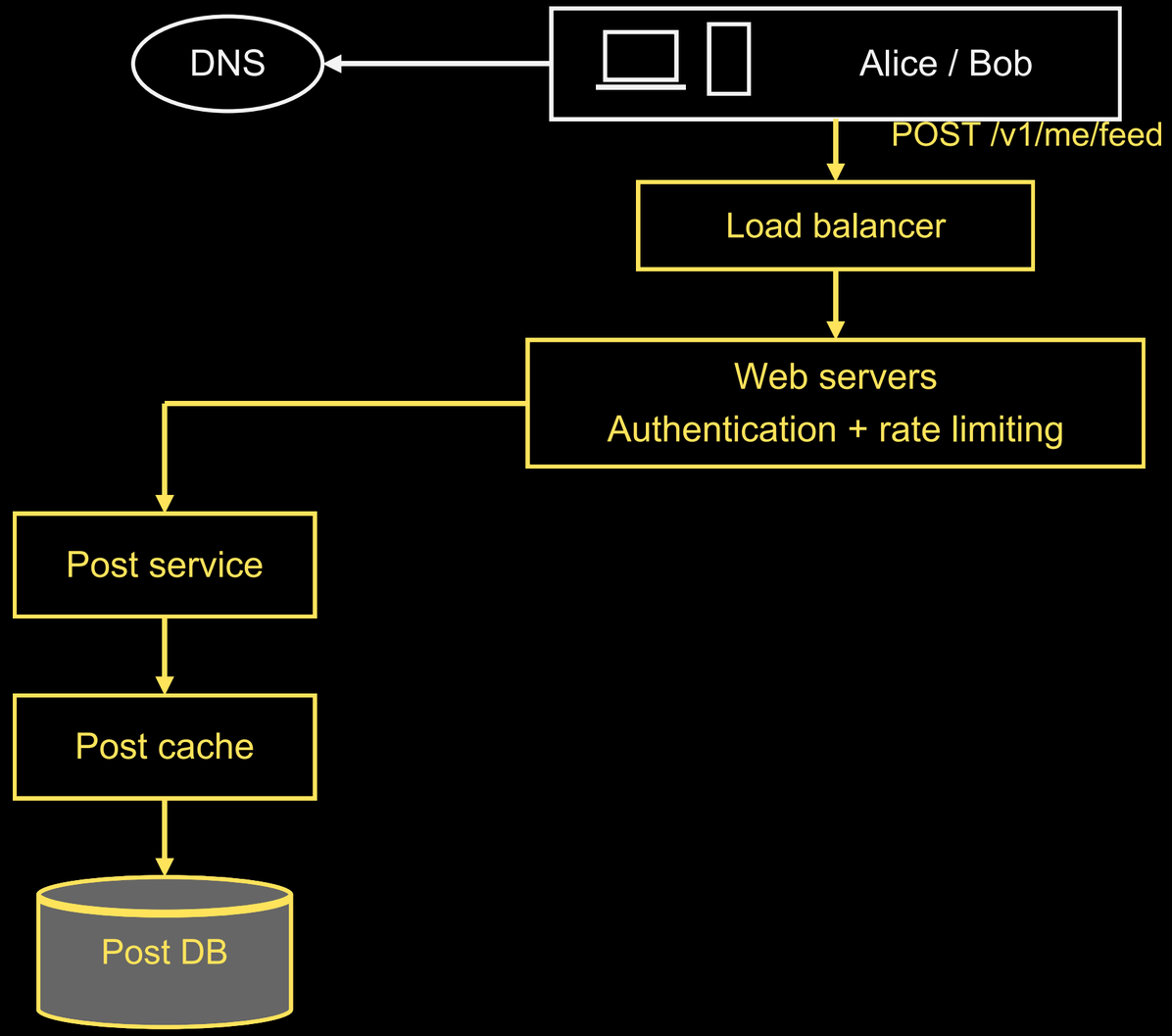
Step 3: Deep dive into publishing a post.

Alice publishes post #42.

Check identity and limits.

Save durably to the DB before saying “success”.

Then update the cache for faster reads.

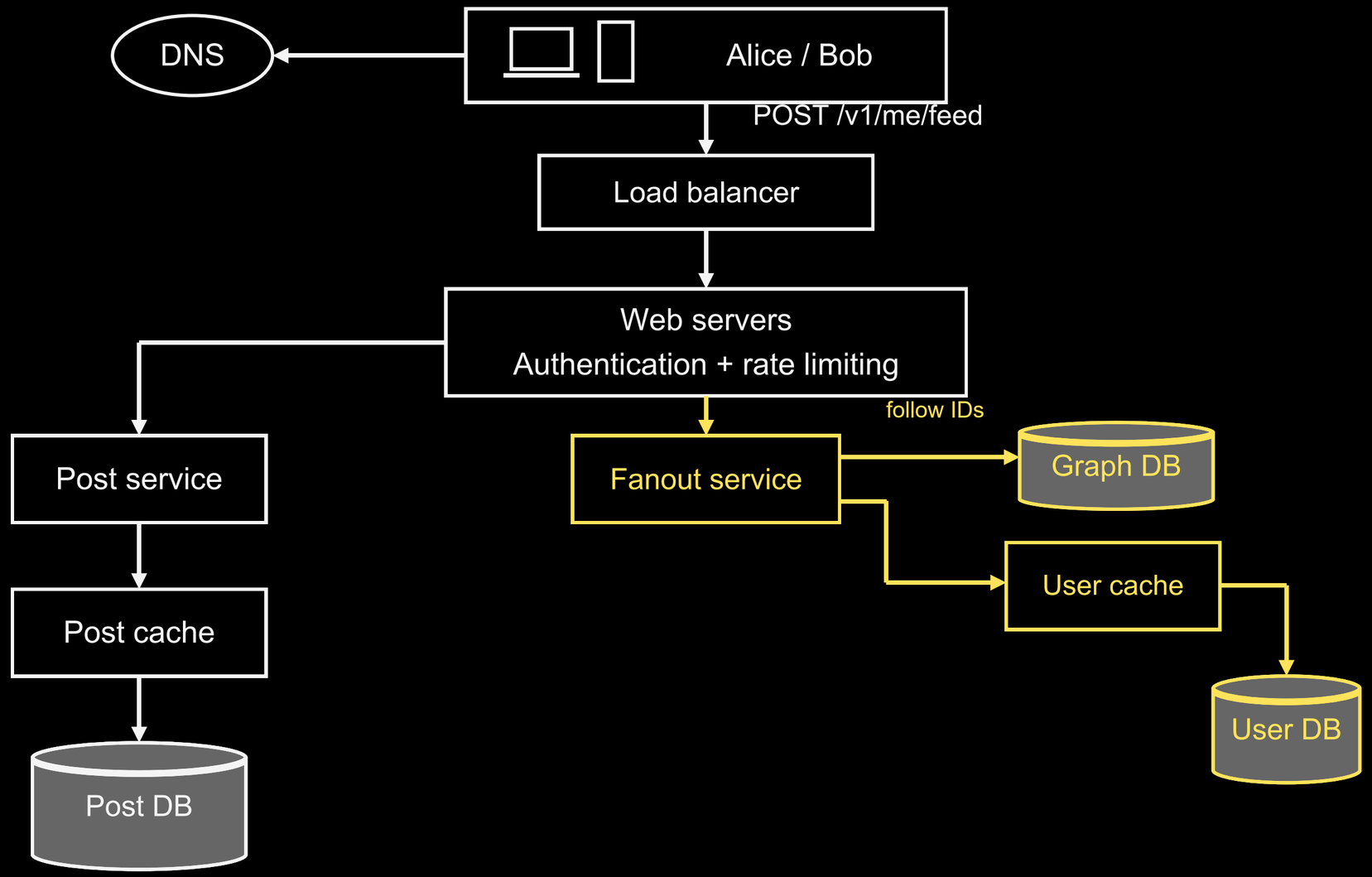


“How do Alice’s followers hear about it?”

Fanout = deliver one post to many followers.

Graph DB: follower IDs.

User cache / DB: filter recipients, such as muted or blocked users.

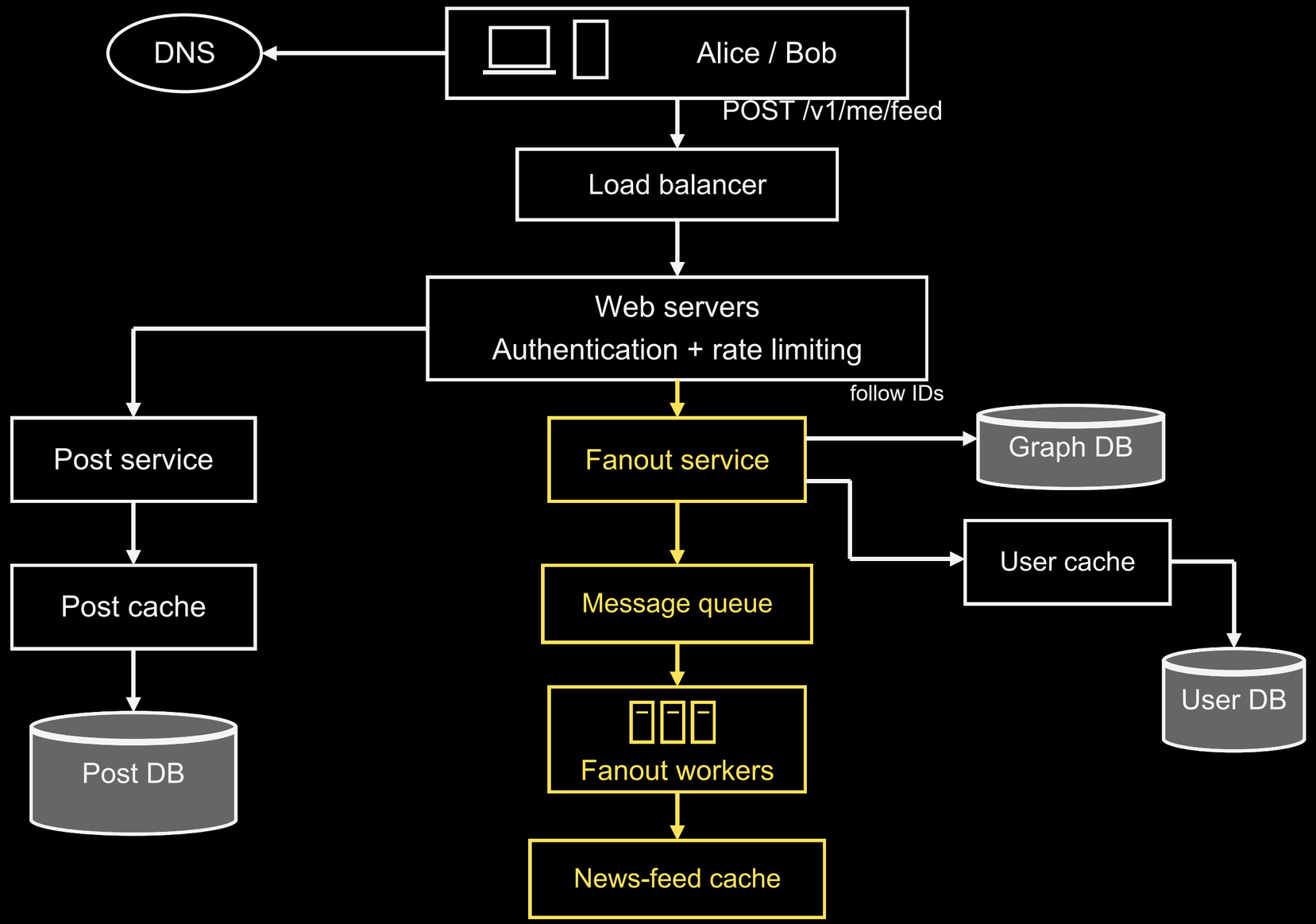


“Do we make Alice wait for every follower?”

Save the post.
Enqueue delivery work.
Then Alice can move on.

Workers add post ID #42
to followers’ feed caches.

Bob’s cached feed:
[42, 38, 35, …]

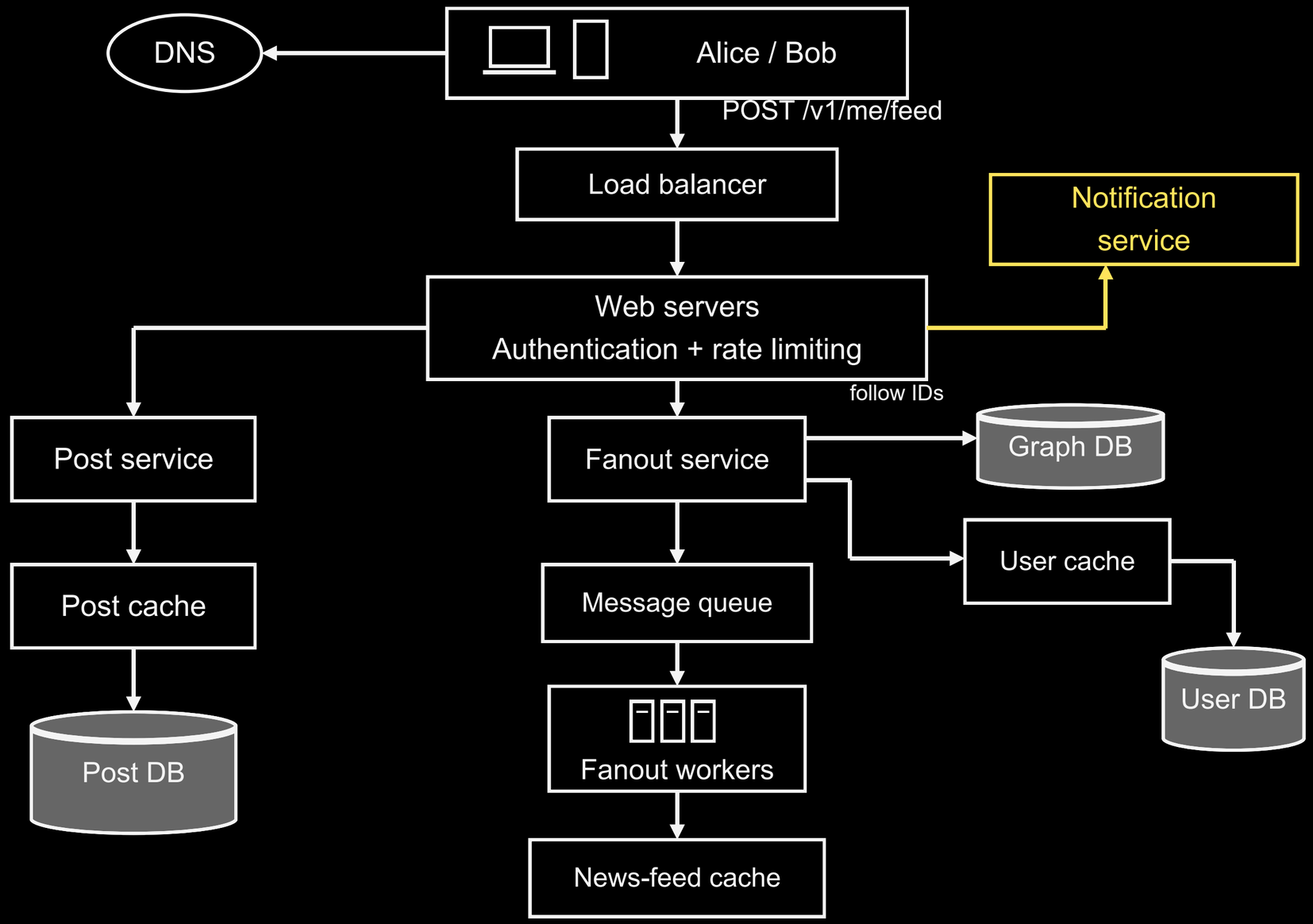


“And notifications?”

The notification service sends the alert asynchronously.

It is separate from building the feed.

A slow alert should not block the saved post.



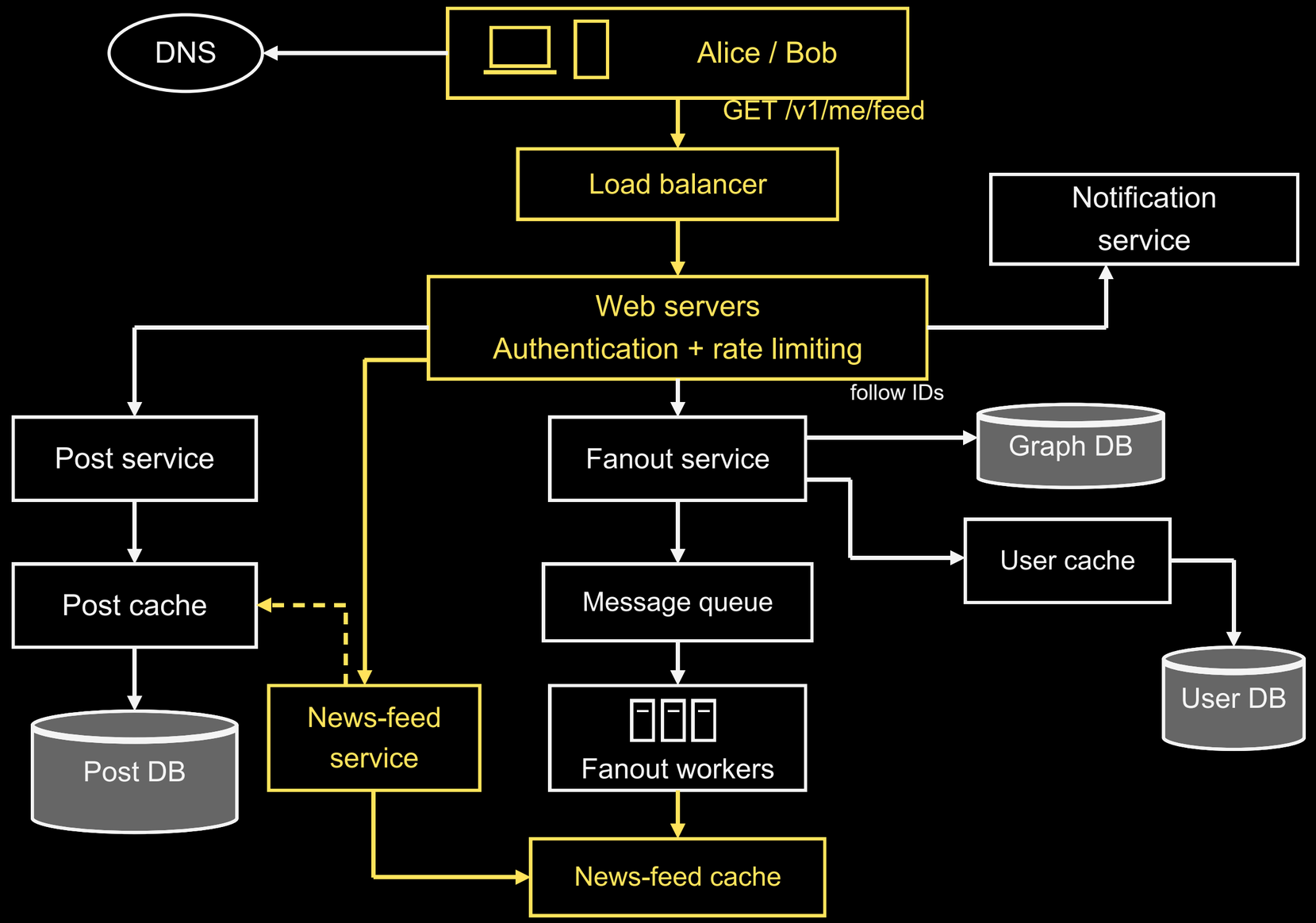
“Now Bob opens his feed.”

GET /v1/me/feed

News-feed service reads Bob’s prebuilt list.

It fills in post details and returns a page.

Same front door. Different path inside.

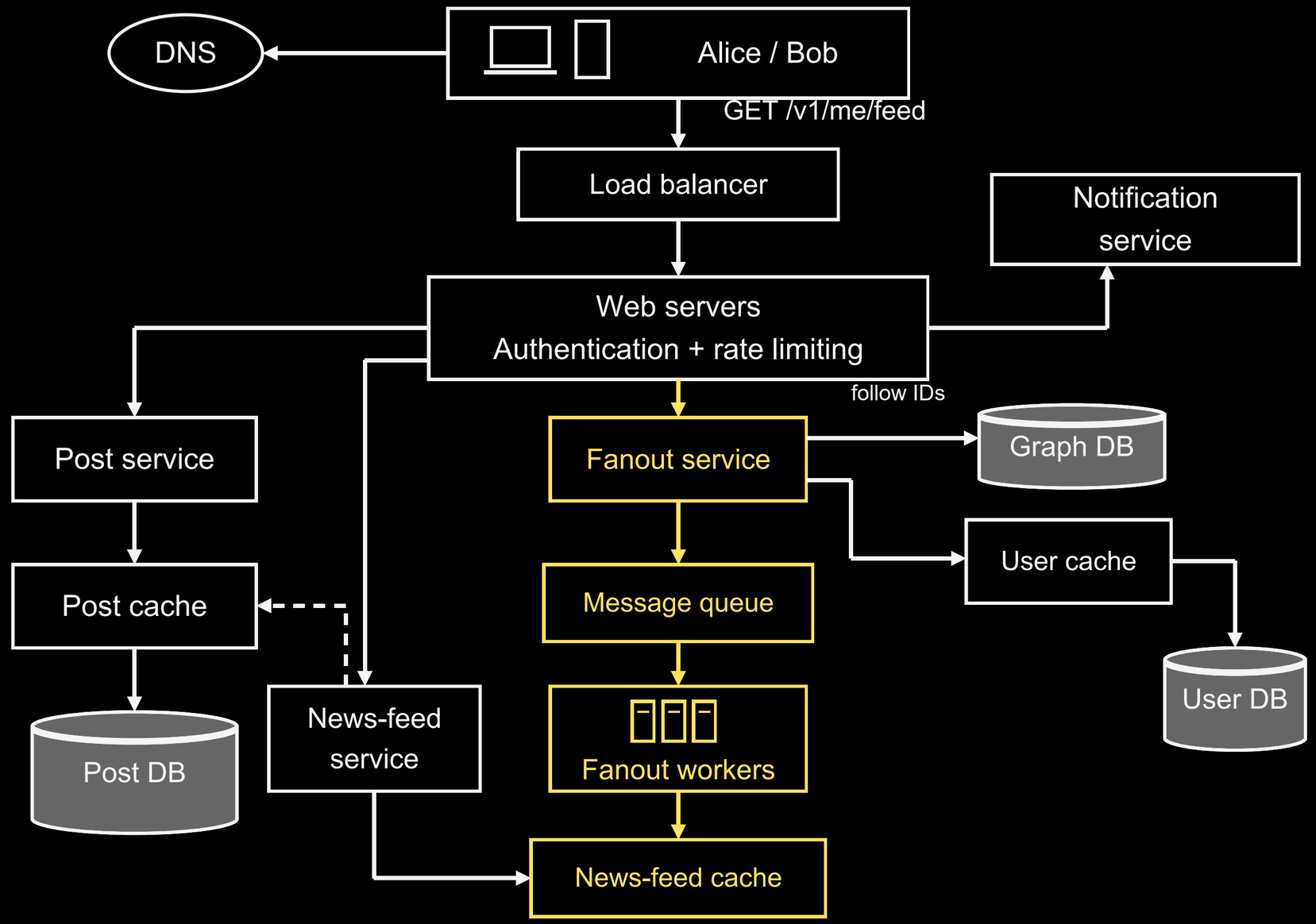


“What happens when Alice has 10 million followers?”

Copying into every feed would be expensive.

A possible hybrid: push ordinary posts; fetch celebrity posts when the feed is read.

Faster writes, more work on the read path.



Your turn: design a rate limiter.

One API server.

Some clients send way too many requests.

Use the same three steps:

1. What would you ask?
2. Sketch the high-level design.
3. Pick one part to deep dive.



Work in groups. 6 minutes.

Step 1: Ask before you draw.

Candidate asks. Interviewer answers.

Client-side or server-side?
Server-side API limiter.

Throttle by IP or user ID?
Flexible rules for both.

How big is the system?
A large number of requests.

Distributed environment?
Not yet. One server for now.

Separate service or app code?
Your call.

Tell users they are throttled?
Yes.

Step 1: Agree on what “done” means.

Accurately limit excessive requests.

Low latency: do not slow down responses.

Use as little memory as possible.

Show clear errors to throttled users.

If the limiter breaks, the API keeps working.

Step 2: Where does the rate limiter go?

Client-side? Easy to fake,
and we may not control it.

Server-side: put middleware
in front of the API servers.

Too many requests?
They never reach the API.

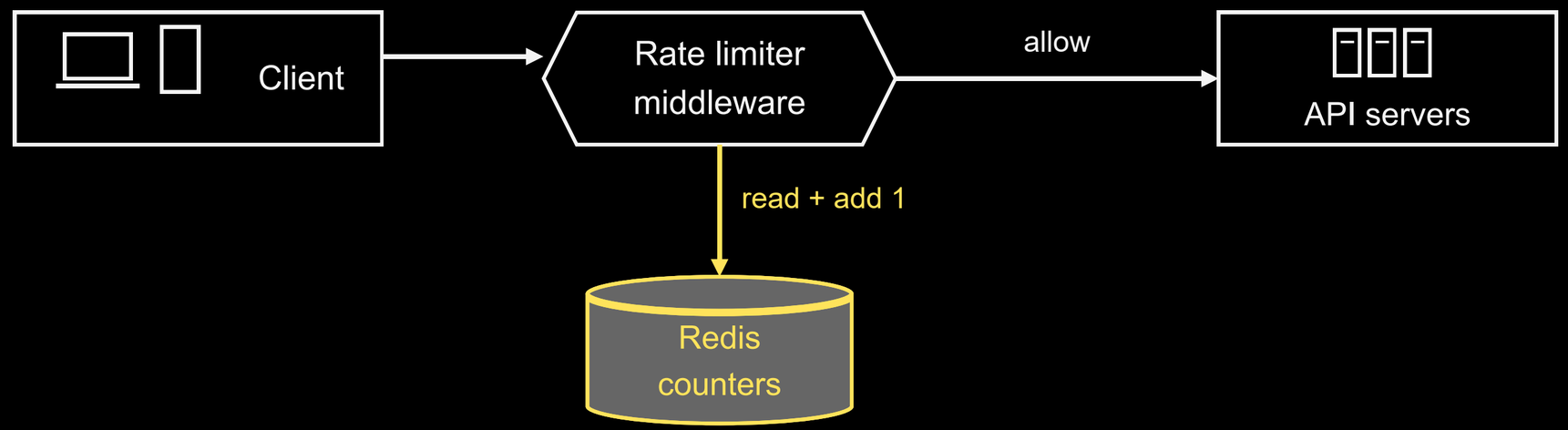


Step 2: How does it count?

Pick an algorithm: token bucket, leaking bucket, fixed window, sliding window.

Keep it simple: count requests per user per minute in Redis.

Interviewer: “Looks good.”

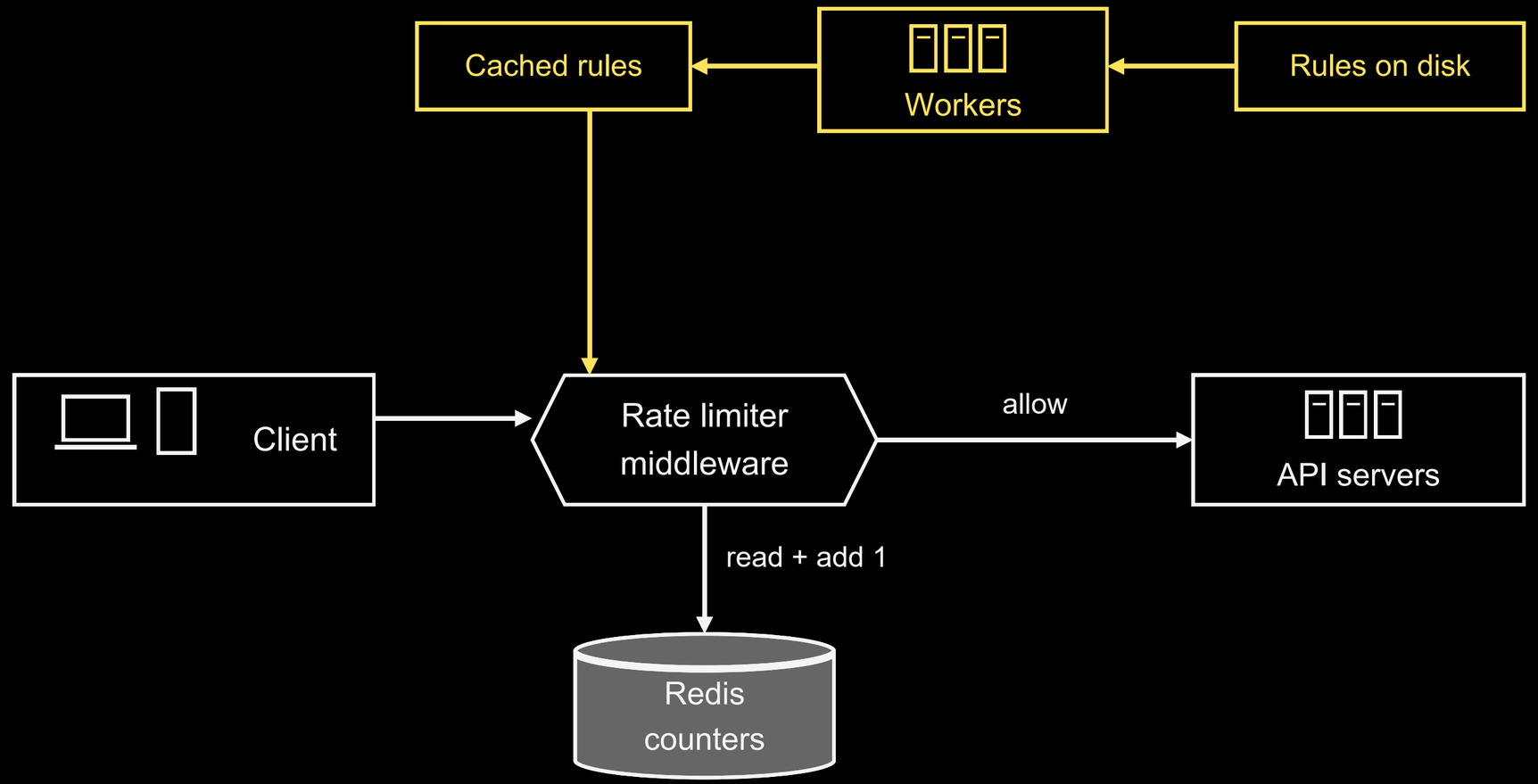


Step 3: Where do the rules live?

Rules sit in config files:
login → 5 per minute.

Workers pull them from
disk into a cache.

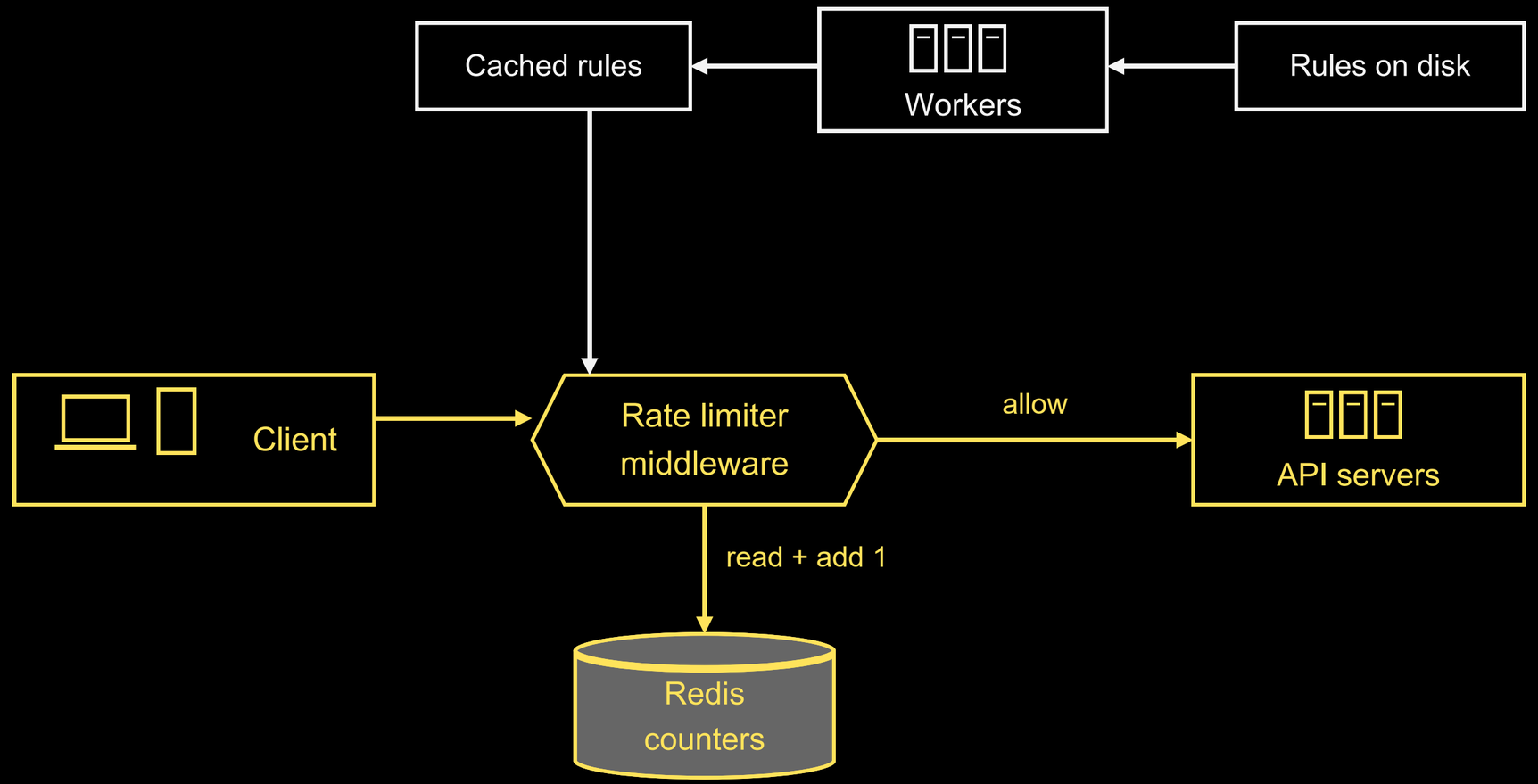
The middleware reads
rules from the cache.



Step 3: Under the limit? Let it through.

Read this user's count
from Redis.

Below the limit:
forward to the API
and add 1 to the count.

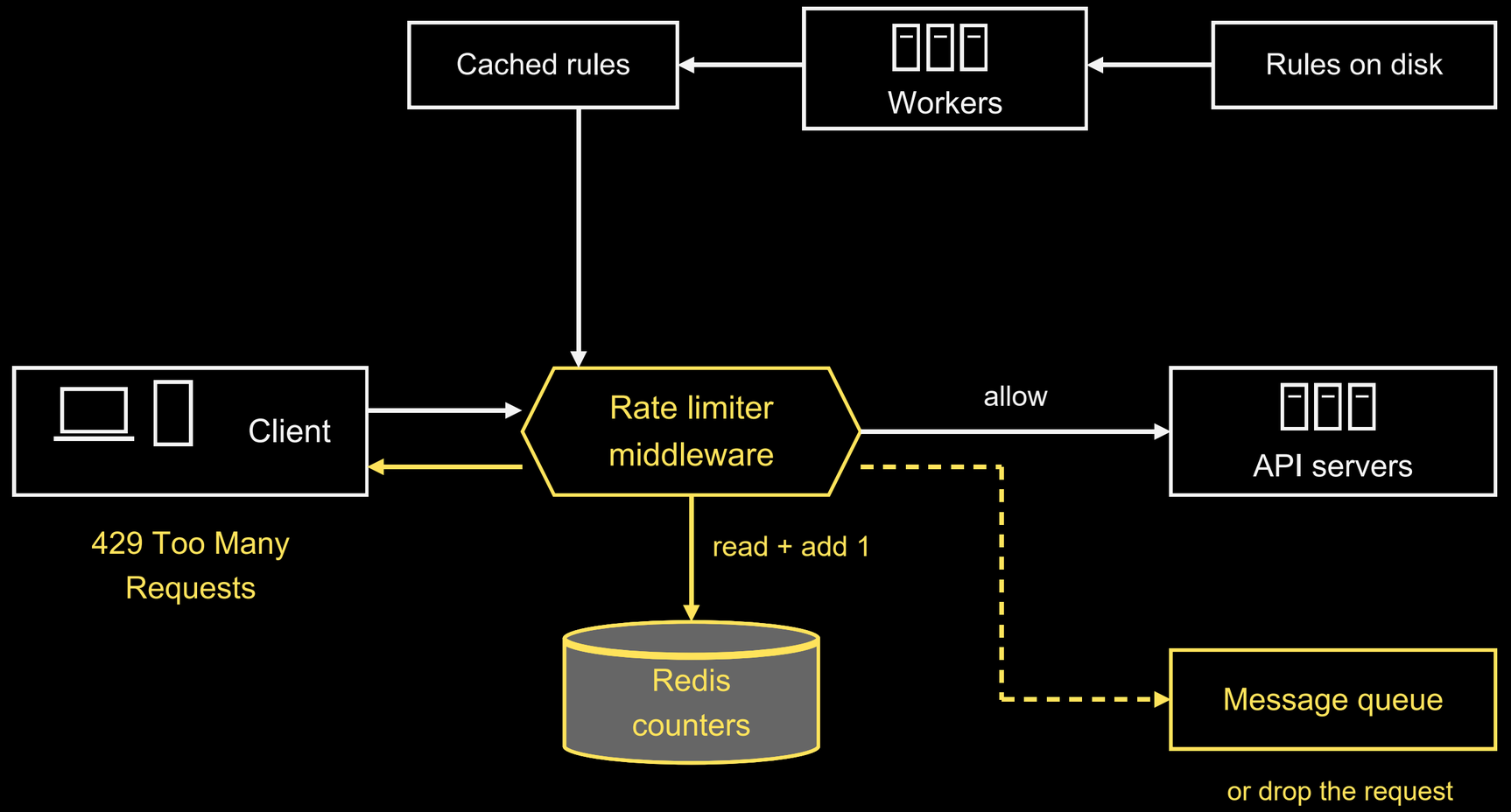


Step 3: Over the limit? Return 429.

Return HTTP 429:
Too Many Requests.

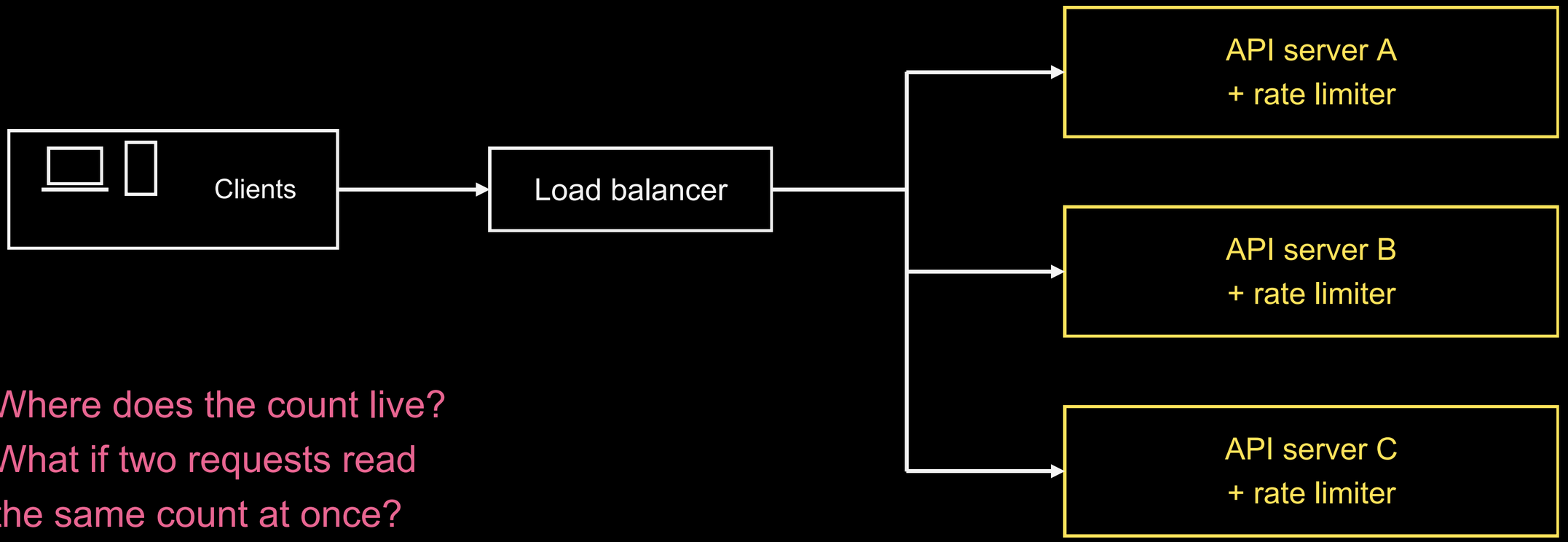
Headers tell the client:
Remaining, Limit,
Retry-After.

Then drop the request,
or queue it for later.



Now make it distributed.

One server was easy. Now the rate limiter runs on three.



Where does the count live?
What if two requests read
the same count at once?

How would you design it?