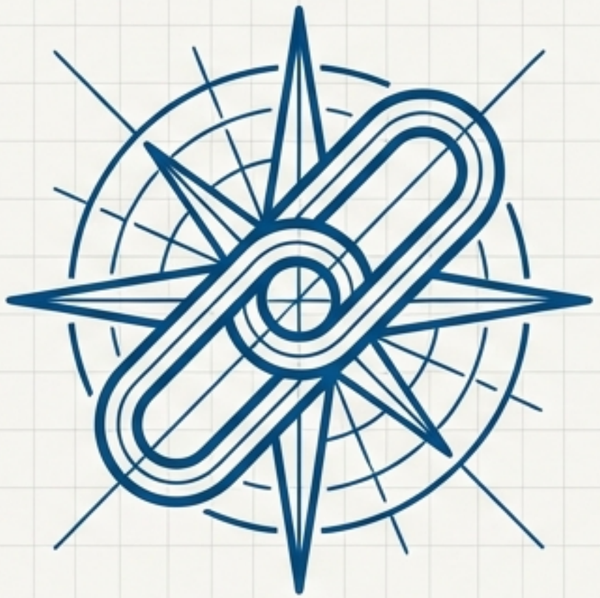
The Architect’s Journey: Designing a URL Shortener from the Ground Up



A step-by-step guide to building a scalable, high-performance service.

Why do we turn long, unwieldy links...

Domain

Path

Session ID

https://www.example.com/products/category-x/item-12345?sessionid=abc-123-xyz-789&source=email&utm_campaign=q4_promo&utm_content=banner_ad

Query Parameters

Source

Campaign

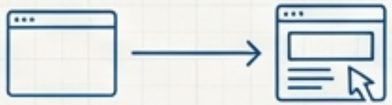
Content

...into short, memorble ones?

Memorable Alias

brand.ly/new-promo

Custom Domain

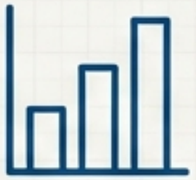


Clean & Shareable

A shorter link is a smarter link.

More Memorable & Branded

Polished links like `brand.ly/feature` are easier to remember and extend your brand's reach.



Trackable Performance

Integrated analytics track click counts and sources, providing valuable engagement data.

Increased Click-Through

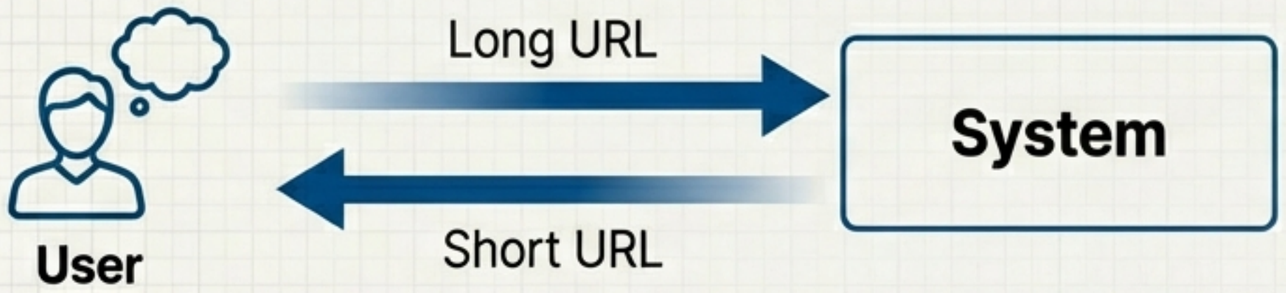
Clean, trustworthy links are more visually appealing and receive more clicks than long, complex URLs.

Character-Efficient

Essential for character-limited platforms, freeing up space for your message.

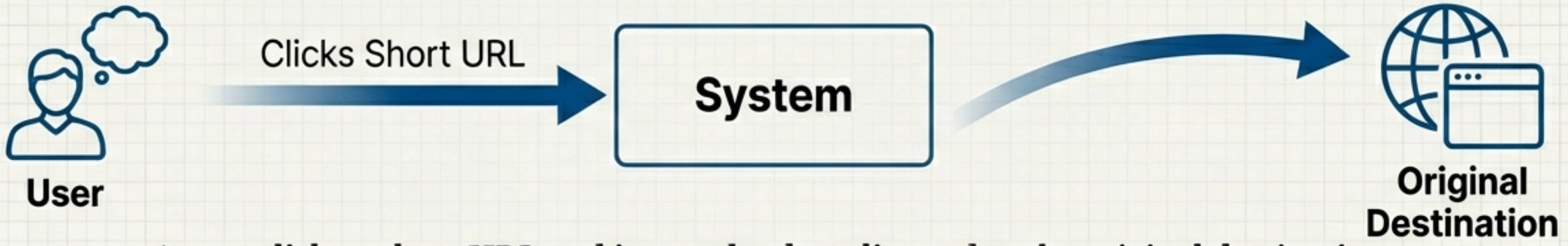
What are the system's core functions?

1. Shorten:



A user provides an original URL and receives a shortened version.

2. Redirect:



A user clicks a short URL and is seamlessly redirected to the original destination.

Beyond function: What defines success?

High Availability:
99.9% uptime for redirects. Our service must always be on.

High Throughput:
Capable of handling a massive volume of requests.

Low Latency:
Redirects must be near-instantaneous to ensure a good user experience.

How big is our user base?

We will start with an assumption of **1 Million Monthly Active Users (MAU)**.
This single figure will drive all our estimations for traffic and storage.

How many new links will we generate?

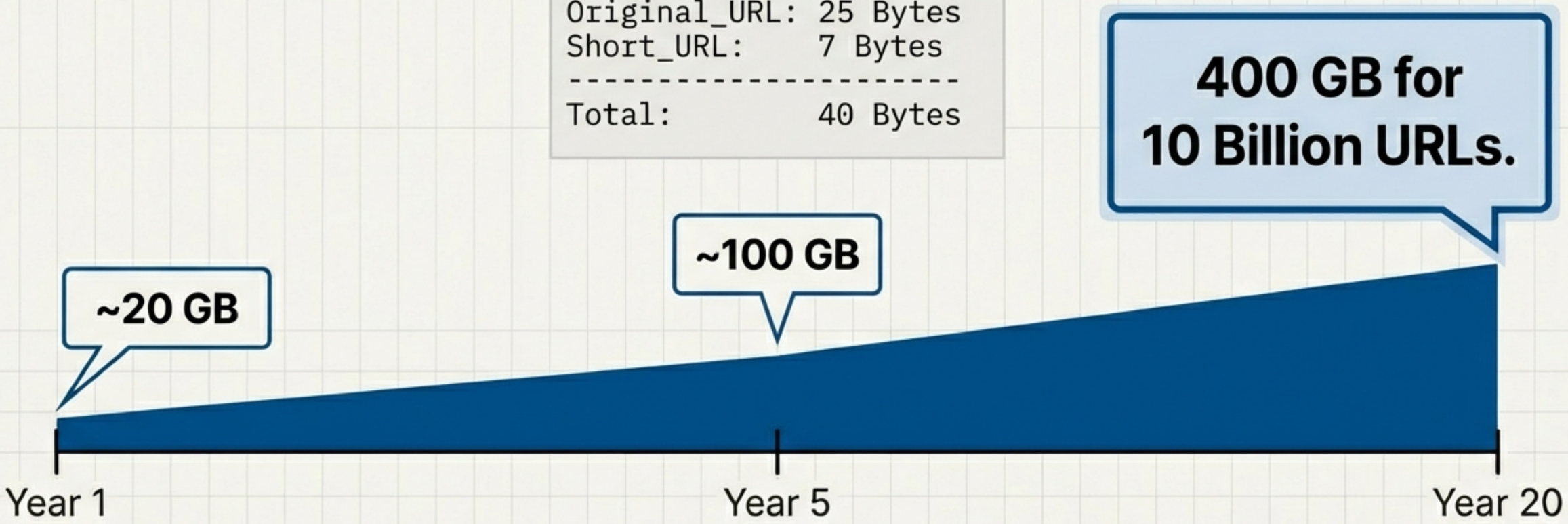
1M MAU → 50% of users → create 2 URLs/day

1 Million new URLs created every single day.

This translates to ~**500 Million** new URLs per year.

How much storage will we need for 20 years?

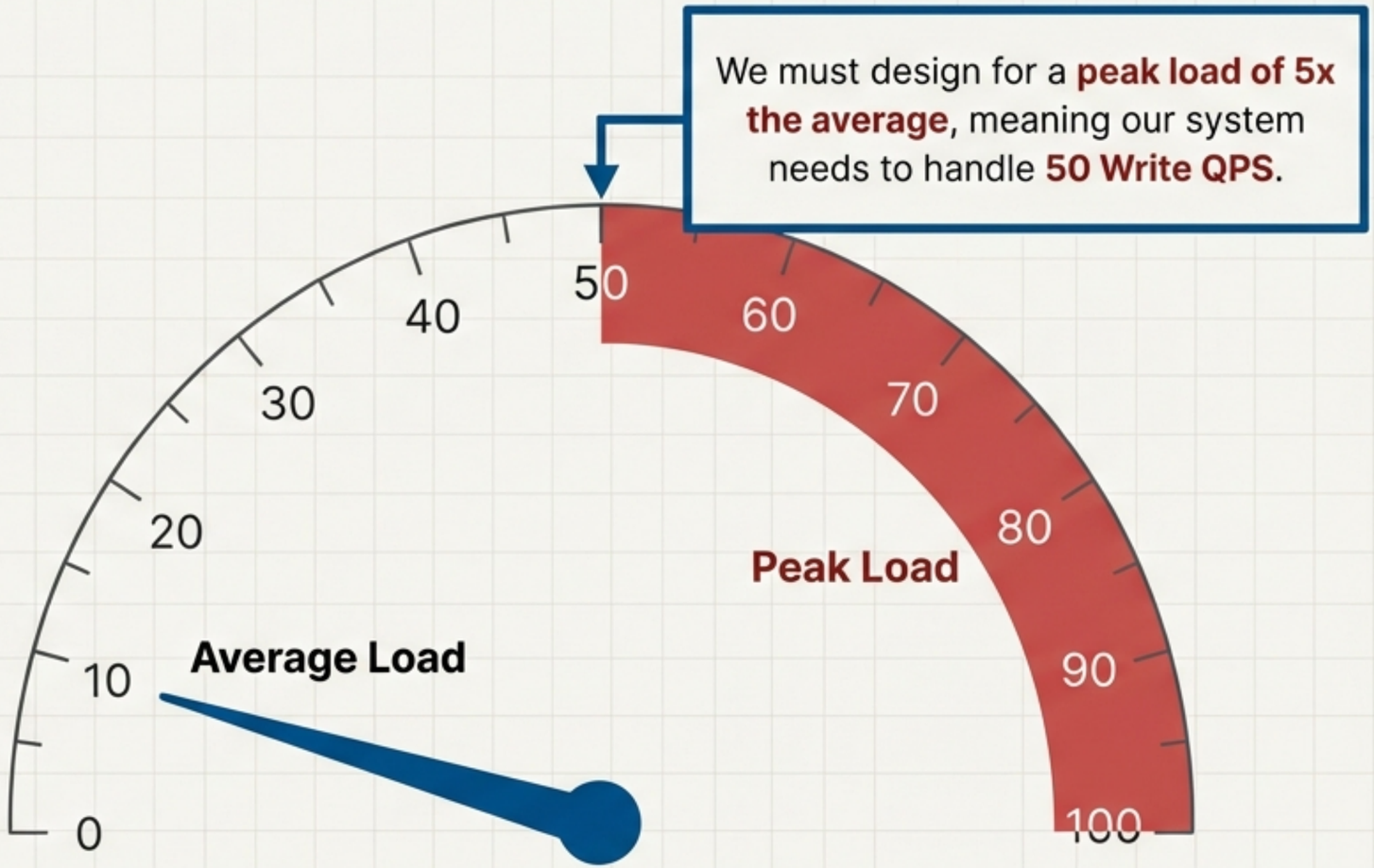
```
Data Schema (per link)
----------------------
PK:            8 Bytes
Original_URL: 25 Bytes
Short_URL:     7 Bytes
----------------------
Total:         40 Bytes
```



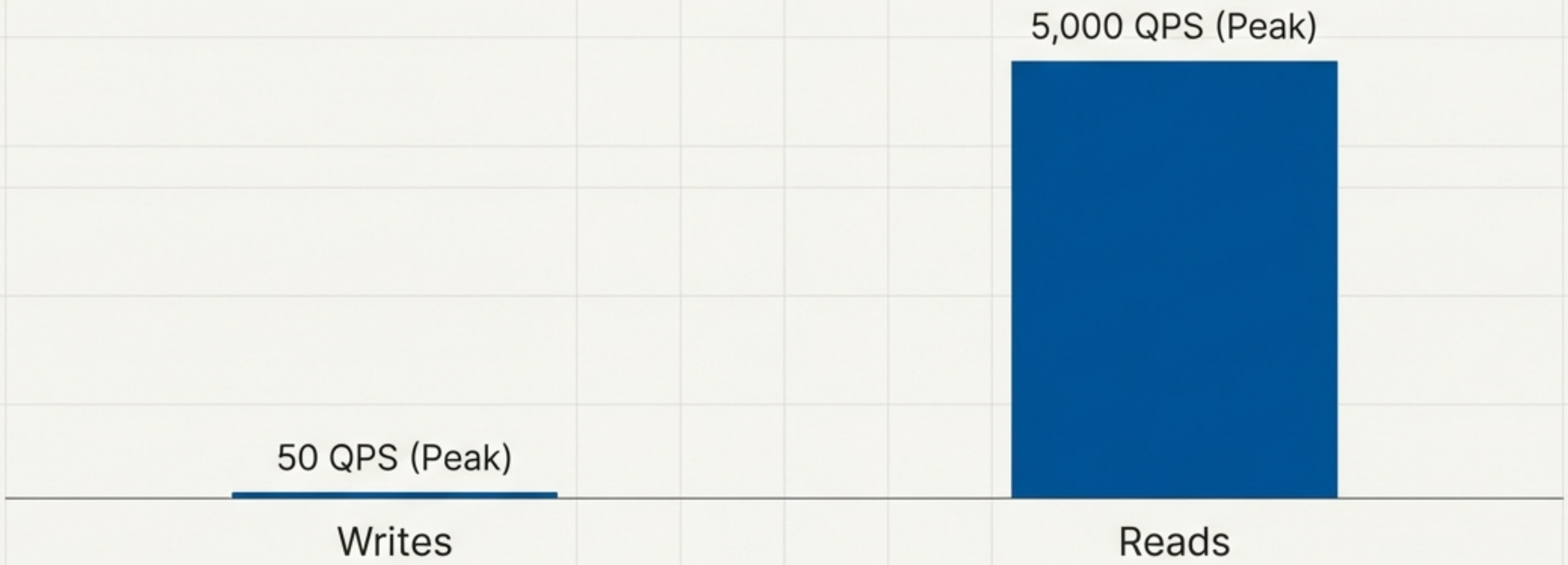
How fast must our ‘write’ operations be?

1 Million URLs /
~100,000 seconds per day

~10 Write QPS (Average)



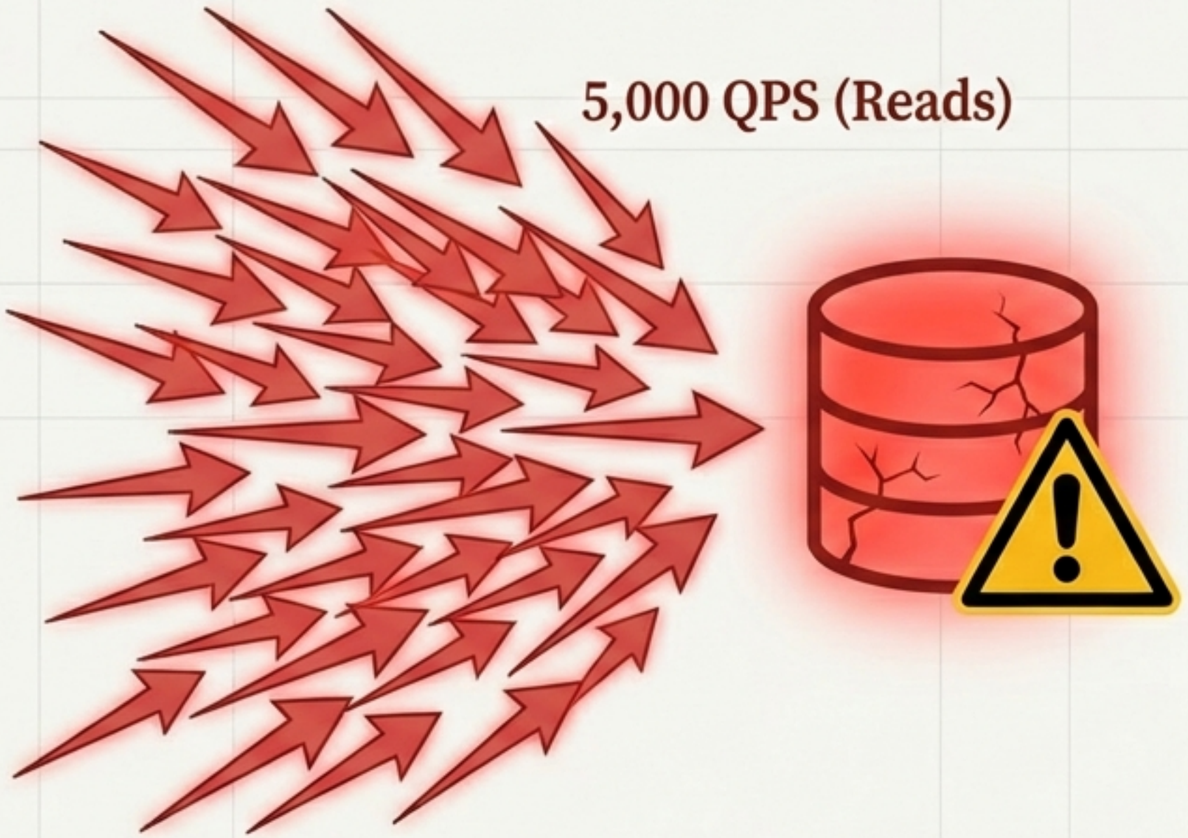
The real challenge isn’t writes. It’s reads.



A **100:1** Read-to-Write Ratio.

For every link created, we anticipate 100 redirect requests.

What happens if every redirect hits our database?

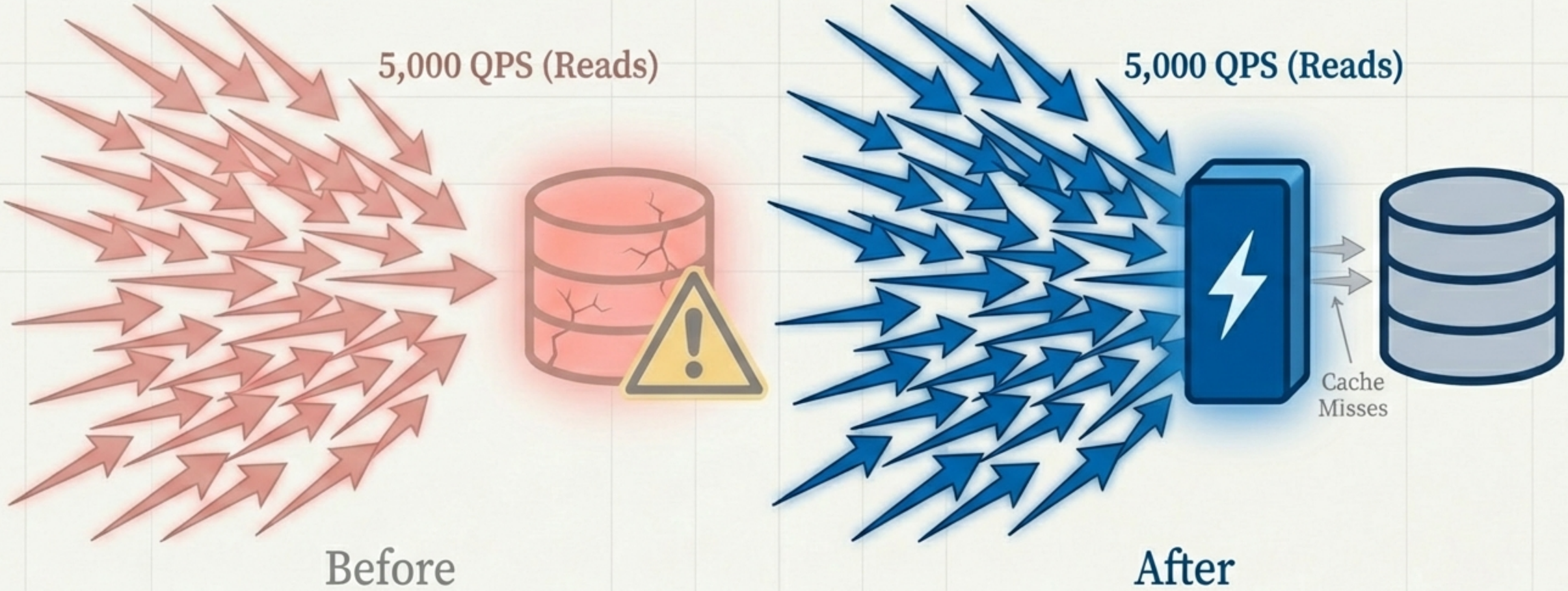


This approach leads to system failure:

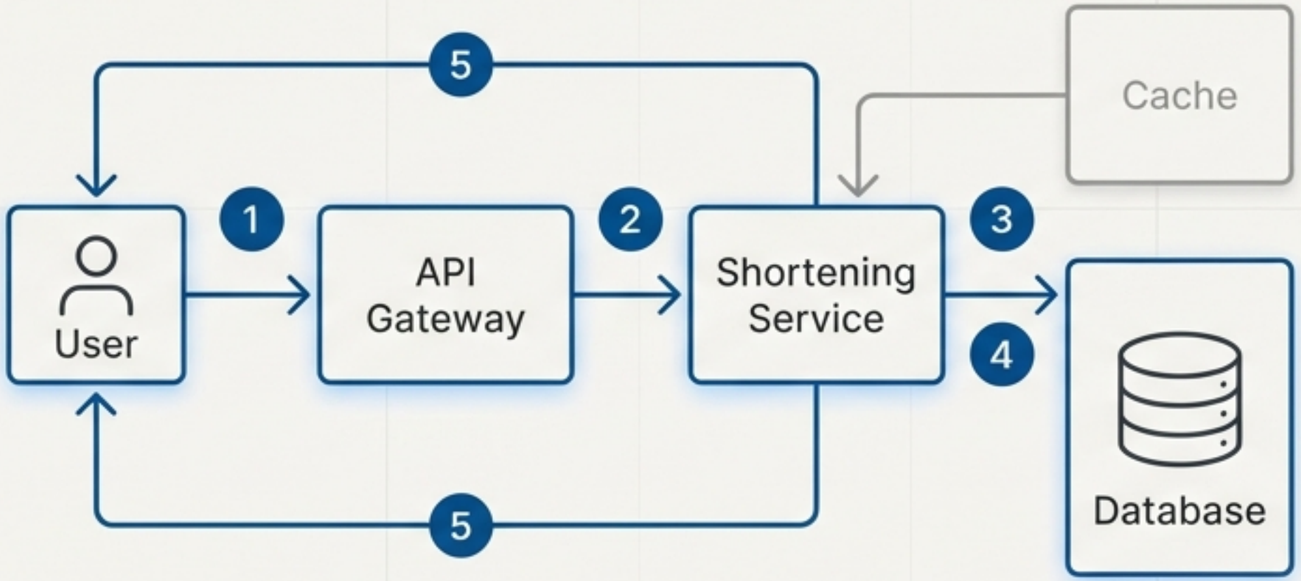
- **High Latency:** Database lookups are slow under load.
- **Exhausted Connections:** The database runs out of available connections.
- **SLA Breach:** We cannot guarantee 99.9% availability.
- **Catastrophic Failure:** The database becomes the single point of failure for the entire service.

The solution is mandatory, not optional.

Redirects *must* be served from an in-memory cache.

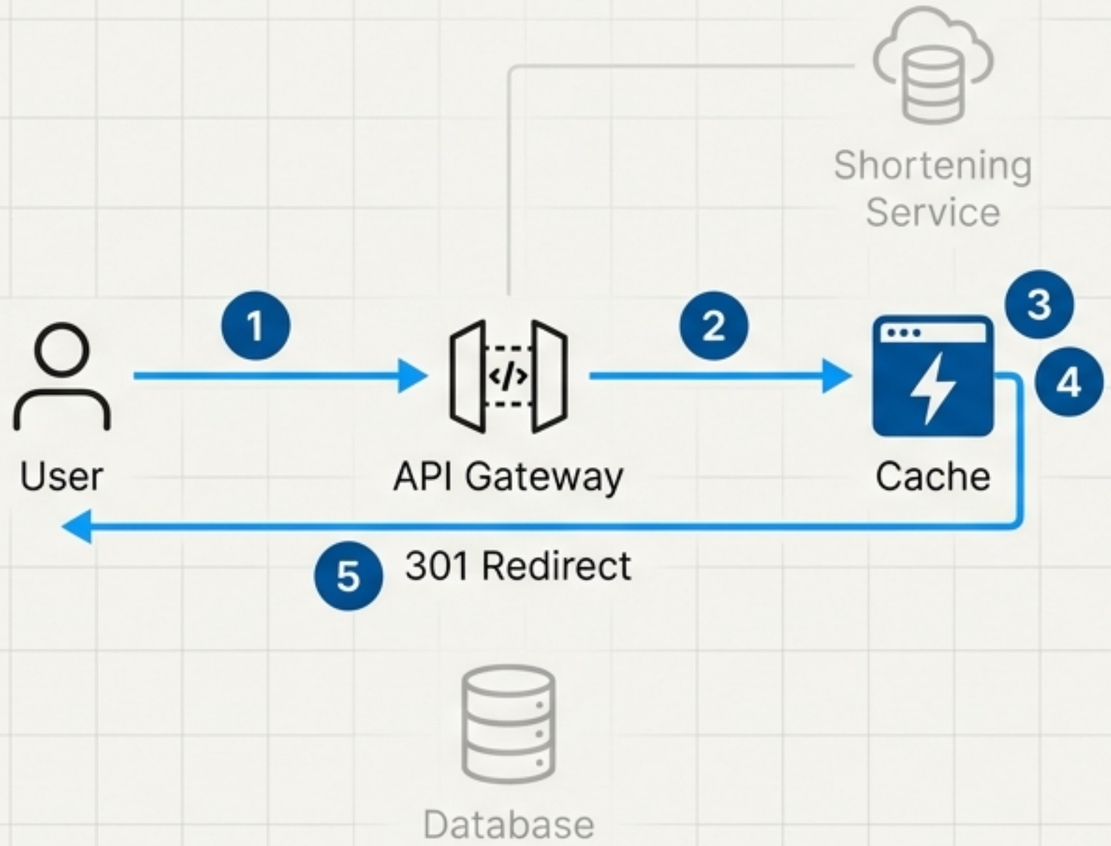


The Final Blueprint (Part 1): The Write Path



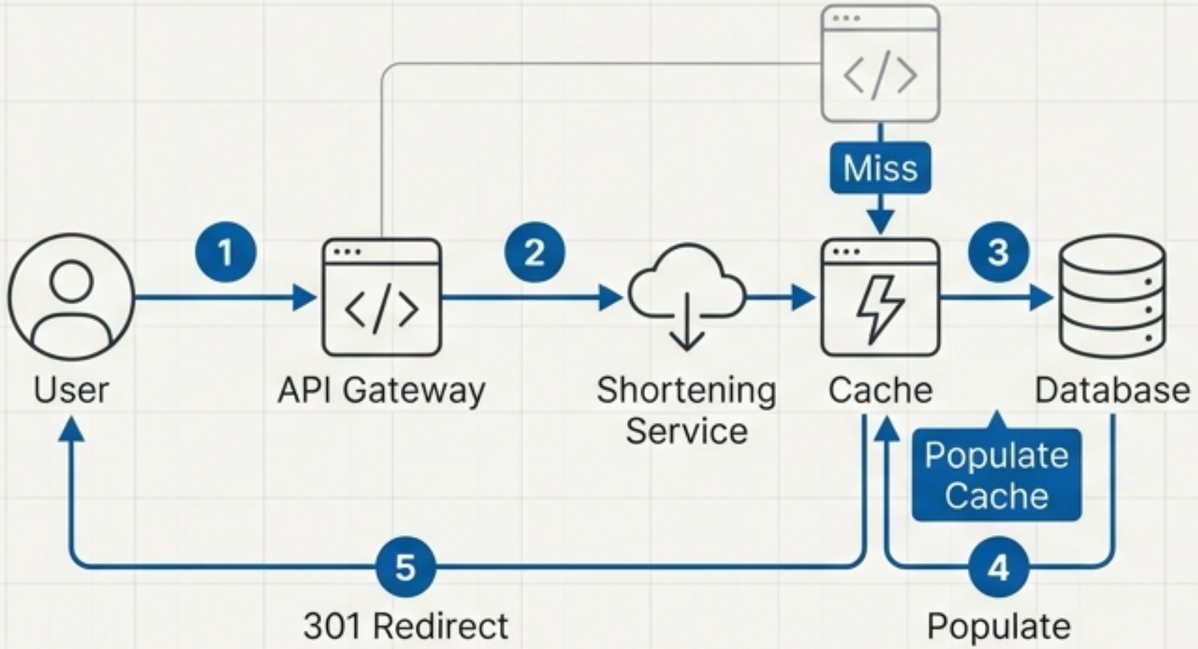
1. A user sends a `POST` request with the original URL.
2. The API Gateway routes the request to the Shortening Service.
3. The service generates a unique short code.
4. The (short_code, original_url) pair is written to the primary database.
5. The new short URL is returned to the user.

The Final Blueprint (Part 2): The Read Path (Cache Hit)



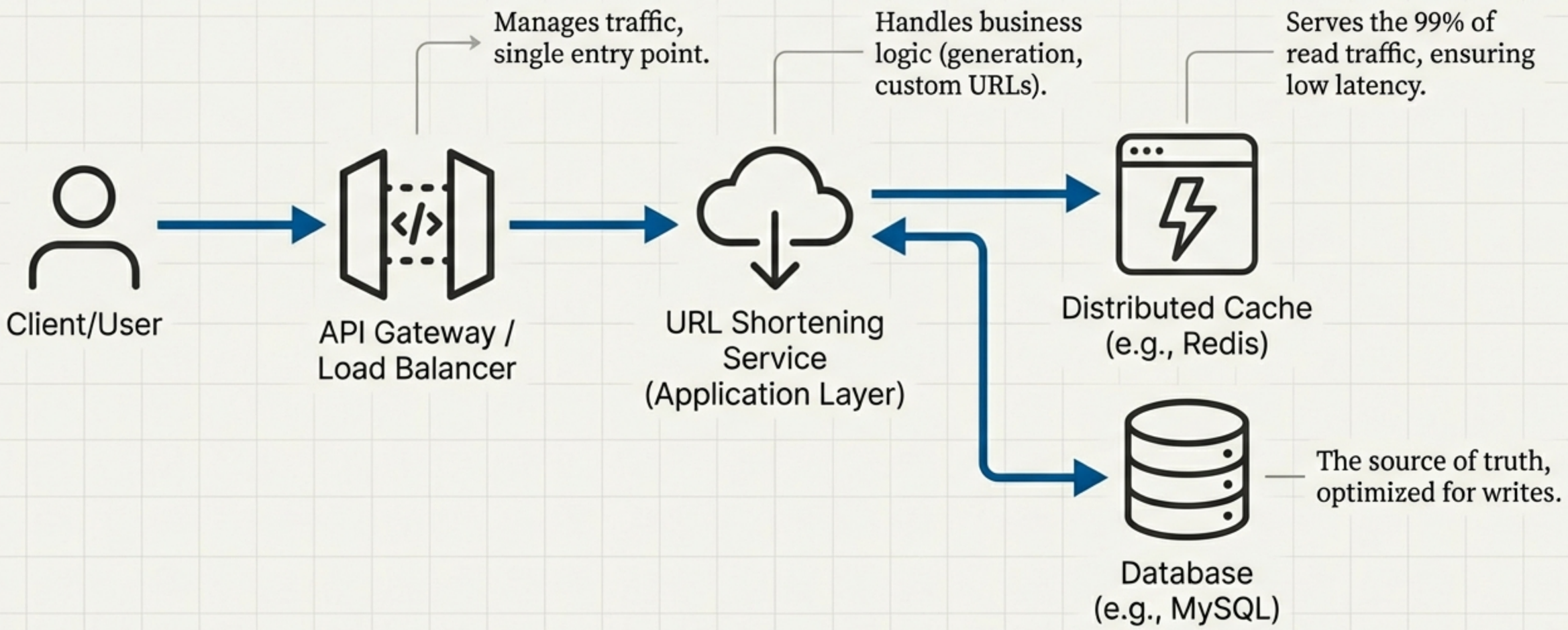
1. A user clicks a short link (**GET` request).
2. The API Gateway routes the request.
3. The system checks the Cache for the short code.
4. **Cache Hit!** The original URL is found instantly in memory.
5. A 301 redirect is returned to the user. **This is the fastest possible path.**

The Final Blueprint (Part 3): The Read Path (Cache Miss)

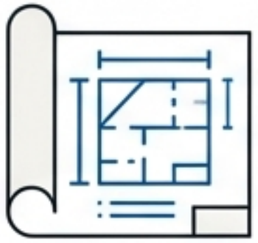


1. The system checks the Cache, but the code is not found (**Cache Miss**).
2. The system queries the primary Database for the short code.
3. The database returns the original URL.
4. **Crucial Step**: The (short_code, original_url) pair is written back to the Cache for future requests.
5. A 301 redirect is returned to the user.

Our Final Architecture: Built for Scale and Speed



The Architect’s Process: From Problem to Solution



Define Requirements

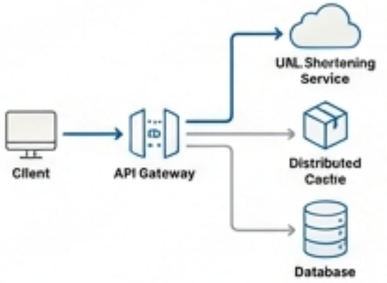
Establish clear functional and non-functional goals.

Estimate the Scale

Use data (MAU) to calculate QPS and storage.

Identify the Bottleneck

Uncover the critical performance driver (the 100:1 read/write ratio).



Design the Solution

Build an architecture that directly solves the identified bottleneck.

A robust system isn't just built; it's justified.