FREE SAMPLE

The HLD Blueprint

You're reading 3 chapters from the full book. The complete edition has 47 chapters, 25 case studies and every cheat sheet.

Before you start

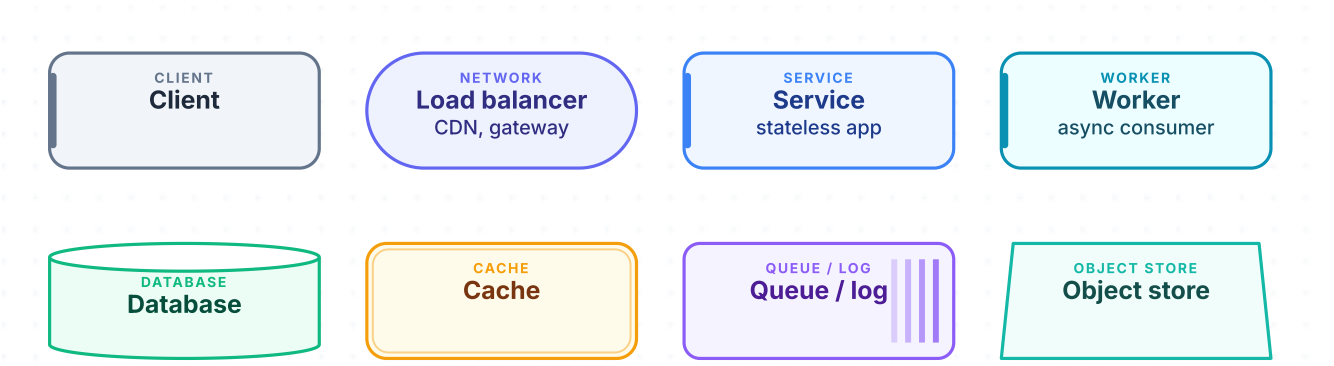
This book has one goal: when someone says *"Design WhatsApp"* or *"How would you scale this?"*, you should feel calm, not cornered. Calm comes from two things: **a method** you trust and **a mental library** of building blocks you can explain in plain words. Every chapter adds to one or the other.

How this book is different

- **One story runs through it.** Chapter 5 grows a single app from 1 user to 100 million. Later chapters zoom into each stage, so you always know *why* a component exists, not only *what* it does.
- **Diagrams carry the explanation.** Every major idea has a picture with a consistent visual language (below). Read the figure first, then the text.
- **Every chapter ends with "Check yourself".** Short questions with answers right underneath. Cover the answers and test yourself.
- **Interview phrasing is spelled out.** Green *"Say it in the interview"* boxes give you sentences you can actually say out loud.
- **The case studies follow one method.** All twenty-five use the **5S method** from Chapter 2, so by the end the method is automatic.

The visual language

Every architecture diagram uses the same shapes, so you can read a figure at a glance:



The shape legend used in every architecture figure.

Arrows: a **solid grey arrow** is a synchronous call, a **dashed arrow** is asynchronous or optional, and a **purple arrow** is an event or message flowing through a log or queue. Red marks something that fails or is a problem.

The callout boxes

© Interview Blueprints · licensed to the purchaser · do not redistribute

THE SHORT VERSION

The chapter in a few lines. Read these again before an interview.

THINK OF IT LIKE THIS

A real-world picture that makes the idea stick.

SAY IT IN THE INTERVIEW

A sentence you can say almost word for word.

COMMON TRAP

A mistake candidates (and production systems) make again and again.

PRO TIP

Something experienced engineers know that isn't in the textbooks.

How to read it

If your interview is **next week**, read Chapters 1–5, skim the *tl;dr* boxes in Parts II–IX, read the annotated mock interview in Part XI, then do six case studies (URL shortener, chat, news feed, ride-hailing, ticket booking, payments) by covering the page and designing it yourself first. If you have **a month**, read in order and do every *Check yourself* box.

A note on numbers: the figures in this book (latencies, throughput, costs) are realistic orders of magnitude, not benchmarks. In design work, being off by 2× rarely matters; being off by 1000× is what sinks a design.

© Interview Blueprints · licensed to the purchaser · do not redistribute

PART I

Think Like a Designer

What high-level design actually is, a repeatable method for any interview prompt, and the numbers you need to reason about scale.

02 The 5S Method for Any Design Prompt

CHAPTER 02 • THINK LIKE A DESIGNER

The 5S Method for Any Design Prompt

A design interview is 45 minutes of controlled ambiguity. The 5S method gives you a path through it: Scope, Size, Sketch, Stress, Summarize.

BY THE END OF THIS CHAPTER YOU CAN

- Run a full design interview with a clear timeline
- Ask the clarifying questions that change the design
- Produce a working end-to-end sketch in 10 minutes
- Pick deep dives that show seniority
- Close strongly instead of running out of time

2.1 The method on one page

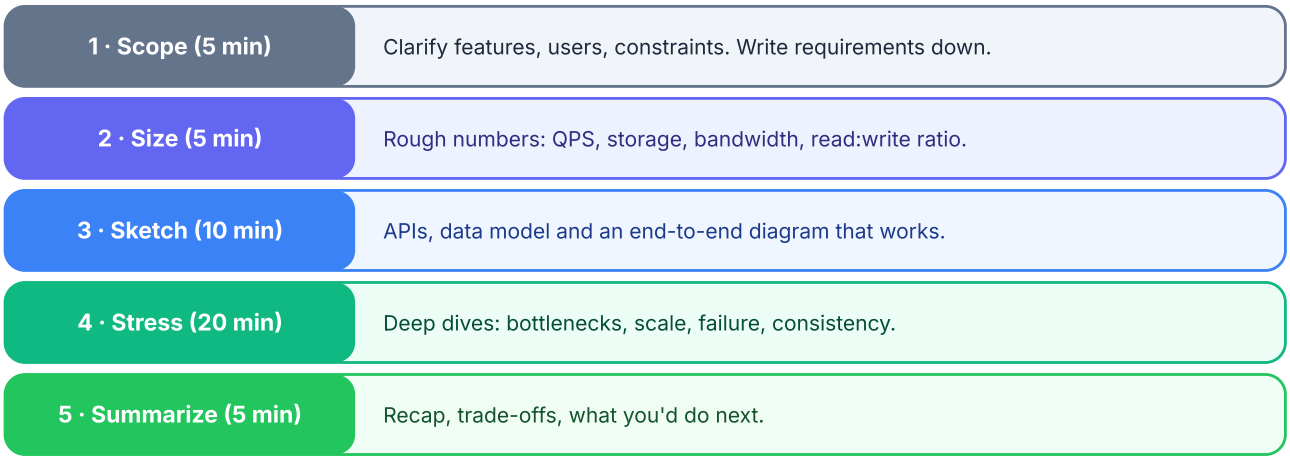


Figure 2.1 The 5S method with a suggested time budget for a 45-minute round.

The order matters. Each step feeds the next: **scope** decides what to size, **size** decides which architecture is even reasonable, the **sketch** gives you something concrete to **stress**, and the **summary** shows that you can see the whole picture.

2.2 Step 1 — Scope: shrink the problem

"Design Instagram" is a product with thousands of engineers behind it. Your job in the first five minutes is to cut it down to something you can design well in forty.

© Interview Blueprints · licensed to the purchaser · do not redistribute

The four kinds of questions

ASK ABOUT	EXAMPLE QUESTIONS	WHY IT MATTERS
Features	Which 3–4 features are core? Do we need search? DMs?	Each feature can add a whole subsystem
Users & scale	How many daily users? Global or one country? Growth?	Decides single box vs distributed
Behaviour	Read-heavy or write-heavy? Real-time? How big is a typical item?	Shapes caching, storage, protocols
Quality bar	Can a post appear 5 s late? Can we ever lose data? Latency target?	Sets consistency, durability, SLOs

Write the answers on the board as two short lists:

```
Functional (what it does)          Non-functional (how well)
- post photo + caption             - 500M DAU, global
- follow users                     - feed p99 < 300 ms
- home feed (followed users)       - eventual consistency OK for feed
- like / comment                   - uploads must never be lost
Out of scope: stories, DMs, ads, search
```

PRO TIP

Saying out loud what is **out of scope** is a strong signal. It shows you can prioritise, and it protects you from being judged on features you never meant to design.

2.3 Step 2 — Size: just enough math

You are not doing accounting. You want **orders of magnitude** that decide the architecture. Chapter 4 covers the techniques; here is the minimum:

1. **Traffic**: daily actions ÷ 100,000 s ≈ average QPS (a day is ~86,400 s; round up). Peak ≈ 2–3× average.
2. **Storage**: items per day × size × retention.
3. **Read:write ratio**: tells you whether to optimise reads (caches, replicas) or writes (logs, sharding).
4. **The one number that scares you**: for a chat app it's concurrent connections; for video it's egress bandwidth; for a URL shortener it's read QPS. Find it and say it.

SAY IT IN THE INTERVIEW

"Roughly 50,000 reads per second at peak against 500 writes: this is read-heavy by 100×, so I'll put caching and read replicas front and centre, and a single write primary is fine for a while."

© Interview Blueprints · licensed to the purchaser · do not redistribute

2.4 Step 3 — Sketch: make it work end to end

In this order:

1. **APIs** — 3–5 endpoints that cover the core features (verb, path, key params, response).
2. **Data model** — main entities, their keys, and which store each lives in.
3. **Diagram** — the simplest architecture that satisfies the functional requirements. Then walk one request through it out loud.

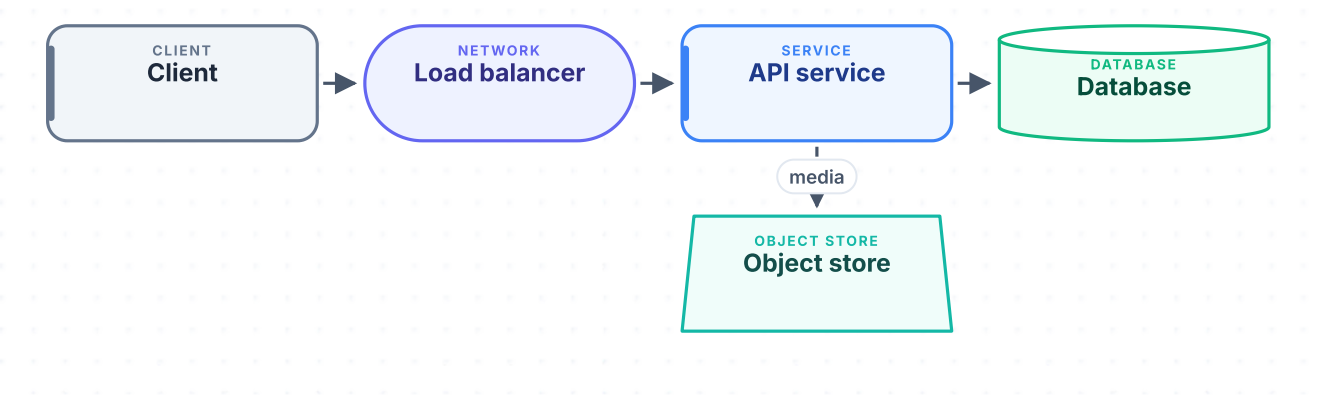


Figure 2.2 A good first sketch is boring on purpose. You'll add sophistication in Stress.

> **COMMON TRAP**
>
> Drawing the "final" architecture with 15 boxes first. You won't be able to explain it, and the interviewer can't tell which boxes you actually understand. Start simple, then evolve it **because of** a number or a requirement.

2.5 Step 4 — Stress: where seniority shows

Now you attack your own design. Pick 2–3 deep dives, ideally ones that tie back to the scary number from Step 2. Good sources of deep dives:

- **Hot paths**: the most frequent request. Can it be served from cache? From the edge?
- **Bottlenecks**: the single DB primary, a hot partition, a fan-out to millions of followers.
- **Failure**: a node dies, a region dies, a dependency is slow, a message is delivered twice.
- **Consistency**: double booking, double charging, lost updates, read-your-writes.
- **Growth**: 10× users, 100× data. What breaks first?

For each, use the loop: **problem** → **options** → **choice** → **cost**.

© Interview Blueprints · licensed to the purchaser · do not redistribute

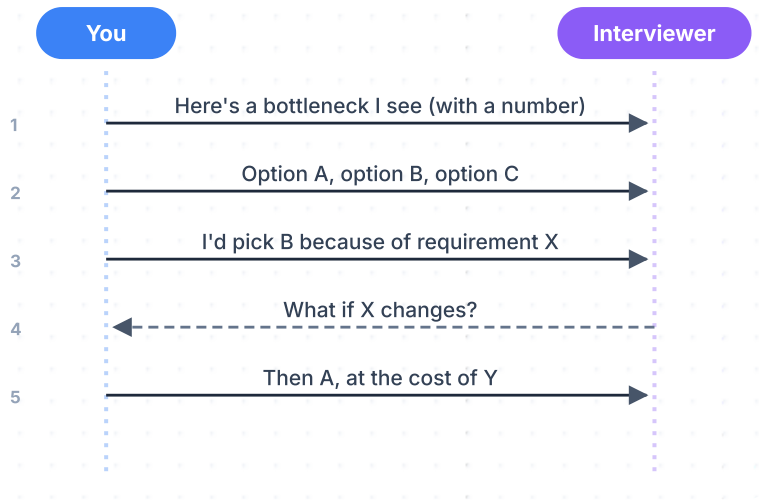


Figure 2.3 The Stress loop you repeat for every deep dive.

PRO TIP

Let the interviewer steer. Ask: *"I'd like to go deeper on either the feed fan-out or the media pipeline. Which is more interesting to you?"* It feels like collaboration and keeps you away from topics they don't care about.

2.6 Step 5 — Summarize: land the plane

Reserve the last few minutes. A clean summary turns a messy interview into a coherent story.

- **One sentence** on the overall shape: *"A read-optimised, cache-heavy service with async fan-out on write."*
- **Two or three key decisions** and their trade-offs.
- **What you'd do next** with more time: monitoring, multi-region, cost tuning, abuse prevention.

2.7 Phrasebook

SITUATION	SAY
Unsure of a requirement	"I'll assume X. Let me know if you'd like a different assumption."
Choosing between options	"Two reasonable options here; I'll pick A for this reason, and note B as a fallback."
Don't know a technology	"I haven't run that in production, but the property I need is X, and here's how I'd get it."
Running out of time	"Let me pause the deep dive and summarize, so we cover the whole picture."
Interviewer challenges you	"Good point. That breaks when... One fix is..." (never defend a broken idea)

© Interview Blueprints · licensed to the purchaser · do not redistribute

2.8 Worked example: 5S on ”design a pastebin”

Here is the method applied to a small prompt, compressed to show the shape of each step.

Scope (out loud): "Users paste text and get a short link; anyone with the link can read it. Optional expiry. No editing after creation. Pastes up to 1 MB. Reads far outnumber writes. Out of scope: accounts, syntax highlighting, search."

Size: "1M new pastes a day ≈ 12 writes/s, peak 50. Reads 10× → 120/s, peak 1,000. Average paste 10 KB → 10 GB/day, ~3.6 TB/year. That's storage-heavy but traffic-light: object storage for content, a small metadata DB, a cache and CDN for hot pastes."

Sketch:

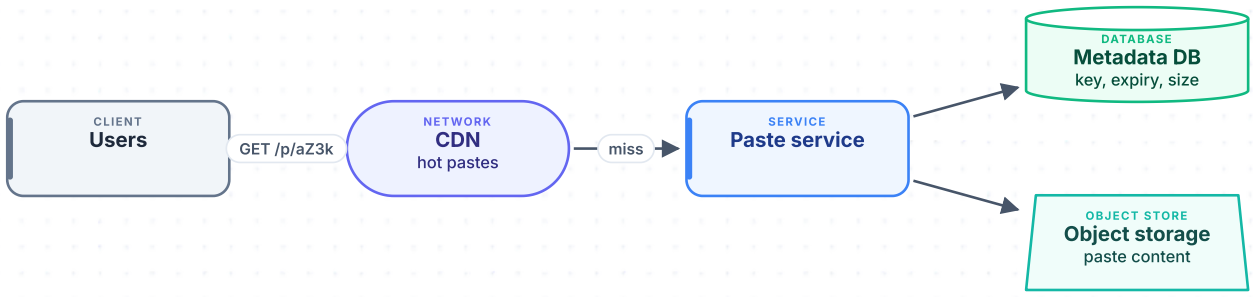


Figure 2.4 Pastebin: metadata in a database, content in object storage, hot reads from the CDN.

`POST /pastes {content, expires_in}` → `{key}`; `GET /p/{key}` → content. Keys: 8 random base62 characters (2×10^{14} possibilities), inserted with a uniqueness check.

Stress: "The interesting parts are (1) expiry: a TTL index or a daily sweeper deletes metadata, and object-storage lifecycle rules delete content; (2) hot pastes: a paste shared on social media gets 50k reads/min, so the CDN caches by key with a TTL bounded by its expiry; (3) abuse: size limits, rate limits per IP and scanning for malware or leaked secrets."

Summarize: "Small metadata DB plus object storage for content, CDN in front. Trade-off: I chose random keys over a counter so pastes can't be enumerated, at the cost of a uniqueness check on write, which is cheap at 50 writes/s."

That whole answer fits in 20 minutes of talking, leaving room for the interviewer's follow-ups, which is exactly where the remaining time should go.

© Interview Blueprints · licensed to the purchaser · do not redistribute

2.9 Adapting the method to the room

SITUATION	ADJUST
60-minute round	Same proportions; spend the extra time in Stress
35-minute round	Merge Scope and Size into 5 minutes total; keep the sketch simple
Interviewer jumps straight to a deep dive	Follow them; return to your plan afterwards ("let me tie this back to the overall design")
Junior / new-grad role	A clean working sketch and sound basics matter most; fewer deep dives expected
Senior / staff role	You're expected to drive: pick the hardest problems yourself, discuss trade-offs, operations and cost unprompted
Prompt is a product you've never used	Ask for the two or three core user actions and design around those

? CHECK YOURSELF

1. What are the five steps and their rough time budget?
   Scope 5, Size 5, Sketch 10, Stress 20, Summarize 5 minutes.
2. Why state what's out of scope?
   It shows prioritisation and prevents being judged on features you never intended to design.
3. What makes a good deep-dive topic?
   Anything tied to the scariest number or the hardest requirement: hot paths, bottlenecks, failure modes, consistency risks, growth.
4. What is the Stress loop?
   Problem → options → choice → cost (and what would change your choice).

© Interview Blueprints · licensed to the purchaser · do not redistribute

PART IV

Making It Fast

The cheapest request is the one you never make. Caches and edges, used carefully.

18 Caching and Redis

CHAPTER 18 • MAKING IT FAST

Caching and Redis

Caching is the highest-leverage tool in system design and the source of its most famous joke: there are only two hard things in computer science, cache invalidation and naming things.

BY THE END OF THIS CHAPTER YOU CAN

- Place caches at the right layer
- Choose between cache-aside, read-through, write-through and write-behind
- Pick TTLs and eviction policies deliberately
- Prevent stampedes, penetration and hot-key meltdowns
- Use Redis data structures beyond simple GET/SET

18.1 Why cache

A cache keeps the result of expensive work (a DB query, a computation, a remote call) in fast storage so repeated requests skip the work. If 90% of reads hit the cache, your database sees one tenth of the load, and p50 latency drops from milliseconds to sub-millisecond.

THINK OF IT LIKE THIS

A cache is the spice rack next to the stove. The pantry (database) has everything, but walking there for salt every time would ruin dinner. You keep the few things you use constantly within arm's reach, and occasionally check whether they've gone stale.

18.2 Where caches live

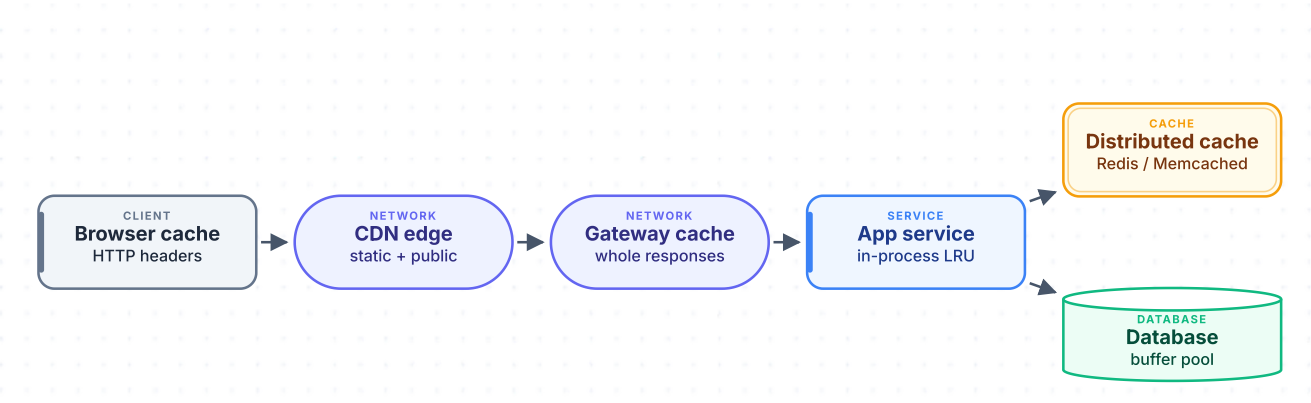


Figure 18.1 Caches exist at every layer. The closer to the user, the cheaper the hit.

© Interview Blueprints · licensed to the purchaser · do not redistribute

LAYER	HIT COST	SHARED?	GOOD FOR
Browser / app	0 network	Per user	Static assets, user's own data
CDN	~10–30 ms nearby	All users in a region	Images, video, JS, public API responses
In-process	~100 ns	Per server	Config, feature flags, tiny hot sets
Distributed (Redis)	~0.3–1 ms	All servers	Sessions, objects, counters, computed results
DB buffer pool	in DB	DB only	Automatic; why "add more RAM" helps

18.3 Caching strategies

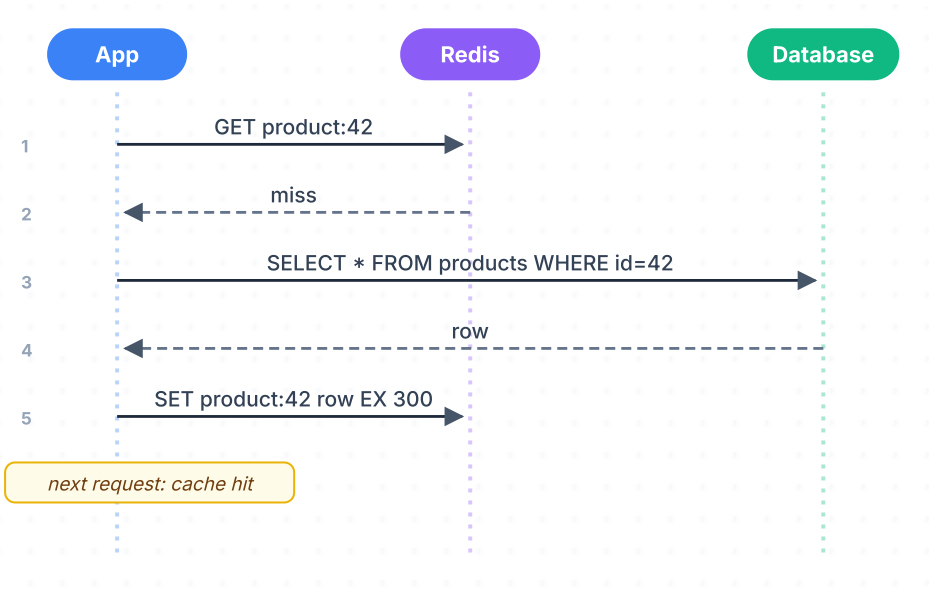


Figure 18.2 Cache-aside (lazy loading): the most common pattern.

STRATEGY	READ	WRITE	PROS	CONS
Cache-aside	App checks cache, loads on miss	App writes DB, deletes cache key	Simple, only caches what's used	First read is slow; brief staleness windows
Read-through	Cache loads from DB itself	—	Cleaner app code	Needs a cache that supports loaders
Write-through	—	Write cache and DB synchronously	Cache always fresh	Slower writes; caches unread data
Write-behind	—	Write cache; flush to DB later	Very fast writes, absorbs bursts	Data loss risk if cache dies
Refresh-ahead	Refresh popular keys before expiry	—	No miss latency for hot keys	Wasted refreshes on keys nobody reads

© Interview Blueprints · licensed to the purchaser · do not redistribute

PRO TIP

On writes, **delete** the cache key rather than updating it. Two concurrent updates can set the cache in the wrong order and leave it permanently stale; a delete simply forces the next read to reload the truth.

18.4 Invalidation and TTLs

You have three tools, usually combined:

1. **TTL**: every key expires after N seconds. Bounds staleness, even if everything else fails. Add **random jitter** (300 s ± 10%) so keys don't all expire together.
2. **Explicit invalidation**: delete keys when the source changes, from the app or from a CDC stream.
3. **Versioned keys**: `product:42:v17`. Bumping the version makes old entries unreachable instantly.

18.5 Eviction policies

When memory is full, something must go:

- **LRU** (least recently used): the default choice for most workloads.
- **LFU** (least frequently used): better when a stable set of items is popular for a long time.
- **TTL-based**: evict keys closest to expiry.
- **Random**: cheap and surprisingly fine for uniform access.

Redis exposes these as `maxmemory-policy` settings such as `allkeys-lru` and `volatile-lfu`.

18.6 When caches hurt you

FAILURE	WHAT HAPPENS	FIX
Stampede (thundering herd)	A hot key expires; 10,000 requests miss at once and hammer the DB	Request coalescing / lock per key, stale-while-revalidate, early refresh
Penetration	Requests for keys that don't exist always miss and hit the DB	Cache negative results briefly; Bloom filter of valid IDs
Avalanche	Many keys expire together, or the cache cluster restarts	TTL jitter; warm the cache gradually; protect the DB with limits
Hot key	One key gets enormous traffic and overloads one cache node	Local in-process copy, replicate the key (`key#1..#N`), read from replicas
Inconsistency	Cache and DB disagree	Delete-on-write, short TTLs, CDC-driven invalidation

© Interview Blueprints · licensed to the purchaser · do not redistribute

COMMON TRAP

Designing as if the cache is always there. When the cache cluster fails, the full load hits the database, which was sized assuming a 95% hit rate. Plan for degraded mode: rate-limit misses, shed non-critical traffic, and warm the cache before reopening the floodgates.

18.7 Redis: more than a cache

STRUCTURE	USE IT FOR
String	Cached objects, counters (`INCR`), simple locks (`SET key val NX PX 30000`)
Hash	Objects with fields (user profile)
List	Simple queues, recent-activity lists
Set	Unique visitors, tags, "who liked"
Sorted set	Leaderboards, time-ordered feeds, rate-limit windows, priority queues
HyperLogLog	Approximate unique counts in 12 KB
Stream	Lightweight append-only log with consumer groups
Geo	"Drivers within 2 km" (Chapter 41)
Pub/Sub	Fire-and-forget fan-out (no persistence)

Durability: RDB snapshots (periodic, may lose minutes) and AOF (append-only log, fsync every second loses at most ~1 s). **Availability**: replicas with Sentinel for failover. **Scale**: Redis Cluster splits the keyspace into 16,384 hash slots across primaries.

18.8 Sizing a cache

Use the access distribution, not the total data size. Most workloads are heavily skewed: a small fraction of keys gets most of the traffic.

1. Estimate the **working set**: e.g. 20% of 50M products are viewed daily → 10M items.
2. Multiply by item size **plus overhead** (Redis adds roughly 50–100 bytes per key): 10M × (2 KB + 0.1 KB) ≈ **21 GB**.
3. Add headroom for growth and fragmentation (~30%) → ~28 GB, split across nodes with replicas.
4. Check the **expected hit rate** against the DB's capacity: at 95% hits, 50k reads/s leaves 2.5k/s for the DB. If the DB can only take 2k/s, you need a higher hit rate or more DB capacity.

© Interview Blueprints · licensed to the purchaser · do not redistribute

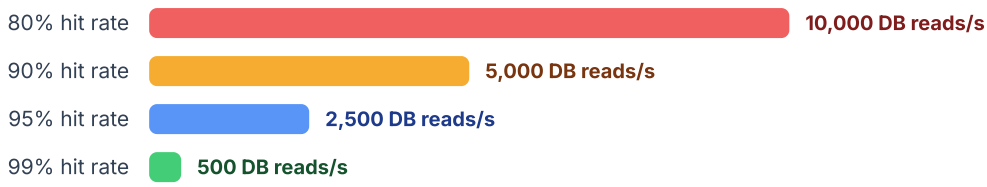


Figure 18.3 How hit rate changes the load your database sees (50,000 reads/s offered).

The jump from 95% to 99% cuts database load by **5×**. Hit rate is the cache metric that matters most.

18.9 Request coalescing in practice

When a hot key misses, only one caller should rebuild it:

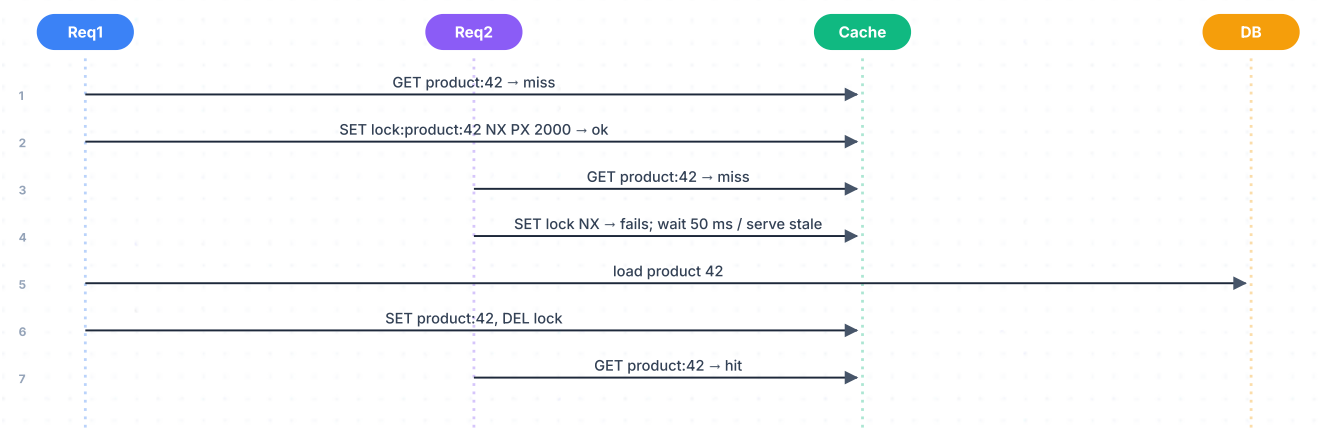


Figure 18.4 Request coalescing: one request rebuilds the value; the others wait for it or serve stale data.

In-process, the same idea is a "single-flight" map of in-progress loads per key. At the CDN, it's called request collapsing (Chapter 19).

> **SAY IT IN THE INTERVIEW**
>
> *"I'll use cache-aside with Redis for product details: 5-minute TTL with jitter, delete-on-write for invalidation, and request coalescing on misses so a popular product expiring doesn't stampede the database. At a 95% hit rate the DB sees about 2,500 reads per second instead of 50,000."*

© Interview Blueprints · licensed to the purchaser · do not redistribute

? CHECK YOURSELF

1. Why delete the cache key on write instead of updating it?

   Concurrent updates can write the cache out of order and leave stale data indefinitely; deleting forces a fresh load.

2. What is cache penetration and how do you stop it?

   Repeated requests for non-existent keys that always miss and hit the DB; cache negative results or use a Bloom filter.

3. Why add jitter to TTLs?

   To avoid many keys expiring at once and triggering an avalanche of misses.

4. Which Redis structure suits a leaderboard?

   A sorted set (ZADD, ZREVRANGE, ZRANK).

© Interview Blueprints · licensed to the purchaser · do not redistribute

PART X

Case Studies

Twenty-five full designs, each walked through with the 5S method: scope, size, sketch, stress, summarize.

01 Design a URL Shortener

CASE STUDY 01 • CASE STUDIES

Design a URL Shortener

Turn long links into short ones like sho.rt/aZ3k9Qx, and redirect billions of clicks quickly. Small problem, surprisingly rich deep dives.

BY THE END OF THIS CHAPTER YOU CAN

- Generate short codes without collisions
- Serve a read-heavy redirect path from cache and edge
- Choose 301 vs 302 based on analytics needs
- Collect click analytics without slowing redirects
- Handle custom aliases, expiry and abuse

C1.1 Scope

Functional

- Create a short URL for a long URL; optionally a custom alias and expiry.
- Redirect short URL → long URL.
- Basic click analytics per link (counts by day, country, referrer).

Non-functional

- Redirect latency p99 < 50 ms (server side); very high availability for redirects.
- Codes must be unique and not trivially guessable.
- Read-heavy: ~100 redirects per creation.

Out of scope: user accounts beyond API keys, link-in-bio pages, QR codes.

C1.2 Size

- 100M new links/month → **~40 creates/s** (peak ~200).
- Redirects 100× → **~4,000/s** average, **~40k/s** peak (viral links).
- Record ≈ 500 B → 50 GB/month → **~6 TB in 10 years**: one sharded-later relational or KV store handles it.
- 7 base62 characters = 3.5 trillion codes ≫ 12B links over 10 years.
- Click events: 4,000/s × ~200 B ≈ 70 GB/day of raw analytics.

Decision: the redirect path is everything. Optimise reads with caches and edge; writes are trivial.

© Interview Blueprints · licensed to the purchaser · do not redistribute

C1.3 Sketch: APIs and data model

```
POST /v1/links        {long_url, custom_alias?, expires_at?}  → {code, short_url}
GET  /{code}          → 302 Location: long_url
GET  /v1/links/{code}/stats?from=&to=  → aggregated counts
DELETE /v1/links/{code}
```

TABLE	KEY	FIELDS
links	`code` (PK)	long_url, owner_id, created_at, expires_at, status
click_rollups	(`code`, day)	total, by_country (map), by_referrer (map)

C1.4 Sketch: the architecture

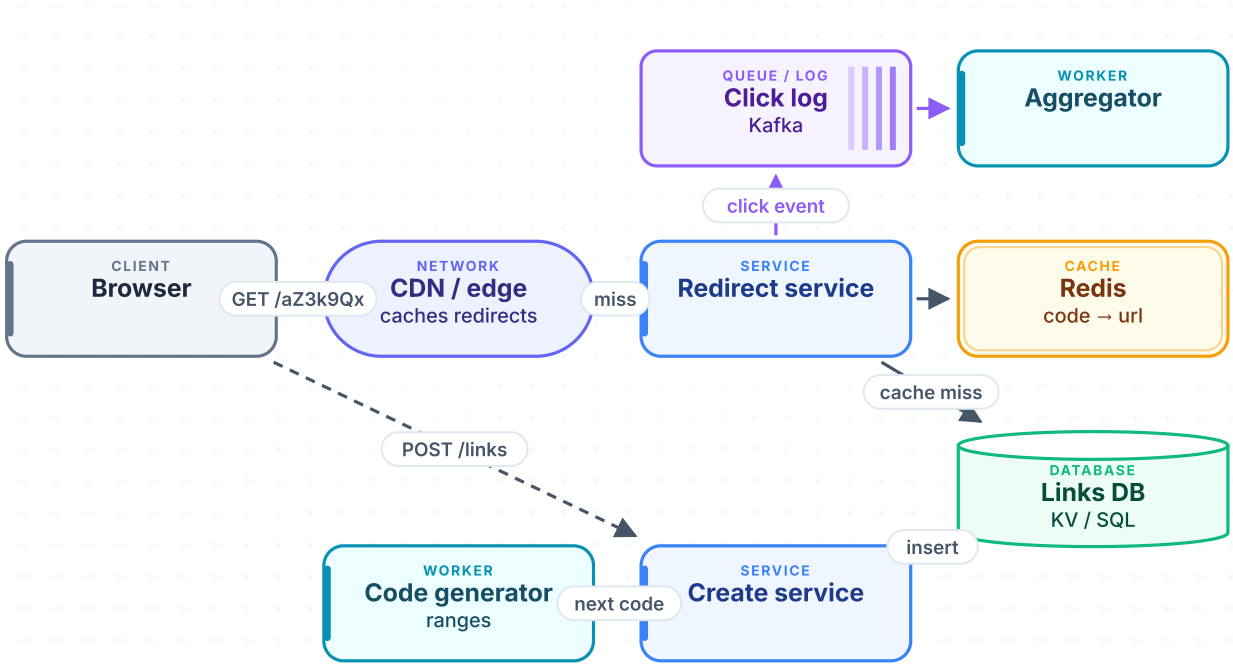


Figure C1.1 URL shortener: a fast redirect path, a separate write path, and async analytics.

The redirect path: CDN → redirect service → Redis → DB. The click is published asynchronously to Kafka; the redirect never waits for analytics.

© Interview Blueprints · licensed to the purchaser · do not redistribute

C1.5 Stress: generating codes

APPROACH	HOW	PROS	CONS
Hash + truncate	base62(MD5(url))[0:7]	Same URL → same code	Collisions need checks and retries
Random	7 random base62 chars, insert with unique constraint	Unguessable, simple	Retry on rare collision; DB check per create
Counter + base62	Global counter → base62	No collisions	Sequential = guessable; counter is a bottleneck
Range allocation	Each create-service instance leases a block of 10,000 IDs from a coordinator, then encodes a **permuted** ID	No collisions, no per-create coordination, not sequential-looking	Lost ranges on crash (harmless)

Choice: range allocation with a bijective scramble (e.g. a keyed Feistel permutation over the 42-bit ID space) before base62 encoding. Codes look random but are guaranteed unique. Custom aliases go through a uniqueness-checked insert in the same table.

C1.6 Stress: making redirects fast

- **301 vs 302.** A 301 (permanent) is cached by browsers, which then skip us entirely: great for load, terrible for analytics. Use **302** (or 307) with a short `Cache-Control: max-age` when analytics matter; offer 301 for customers who don't need click counts.
- **Edge caching:** the CDN caches the 302 response for popular codes for 60 s; viral links barely touch origin. Click events are then collected from CDN logs or an edge worker.
- **Redis:** hot set ≈ 20% of active links; even 1B links × 20% × 500 B = 100 GB, a modest Redis cluster. LRU eviction.
- **Negative caching:** cache "not found" for 60 s so bots scanning random codes don't hit the DB.

C1.7 Stress: analytics at scale

Click events go to Kafka keyed by `code`. A stream processor aggregates per (code, day, country, referrer) in 1-minute tumbling windows and upserts into `click_rollups` (or ClickHouse for flexible queries). Counts are eventually consistent by about a minute, which is fine. Raw events land in object storage for reprocessing.

C1.8 Stress: failures and abuse

- **Redis down:** fall back to DB reads with rate limits; the DB sized for ~20% of peak.

© Interview Blueprints · licensed to the purchaser · do not redistribute

- **Malicious URLs:** scan on create (Safe Browsing-style lists), re-scan periodically, and show an interstitial for flagged links.
- **Enumeration:** non-sequential codes plus rate limits per IP on 404s.
- **Expiry:** check `expires_at` on read; a background job deletes expired rows and their cache entries.

C1.9 Summarize

> **SAY IT IN THE INTERVIEW**
>
> *"A read-optimised service: codes from leased ID ranges, scrambled and base62-encoded, so they're unique with no per-request coordination. Redirects are 302s served from CDN and Redis, with clicks published asynchronously to Kafka and rolled up per minute. The main trade-off is 302 over 301: more traffic to us in exchange for accurate analytics."*

C1.10 Follow-ups the interviewer may ask

- *How do you support custom aliases without colliding with generated codes?* Keep one `links` table keyed by code with a unique constraint. Generated codes use a fixed length (7) and alphabet; aliases may be any length within limits, and a collision with an existing code simply fails the insert with a friendly "taken" message.
- *How would you make it multi-region?* Redirects are read-only, so replicate the links table asynchronously to every region and serve redirects locally. Creates go to a home region (or each region gets its own ID ranges), and a just-created link is served from the creating region until replication catches up.
- *How do you delete a link that's cached in CDNs and browsers?* Purge it from the CDN by URL, delete it from Redis, and mark it deleted in the DB. Browsers that cached a 301 can't be reached, which is one more reason to default to 302 with short cache lifetimes.
- *What if analytics consumers fall behind for an hour?* Redirects are unaffected because clicks are published asynchronously. Kafka retains the events; consumers catch up, and rollups are eventually correct.

© Interview Blueprints · licensed to the purchaser · do not redistribute

? CHECK YOURSELF

1. Why is 302 usually preferred over 301 here?

   Browsers cache 301s permanently, so later clicks never reach the service and analytics are lost.

2. How does range allocation avoid a global counter bottleneck?

   Each instance leases a large block of IDs and assigns them locally, contacting the coordinator only once per block.

3. How do you stop random-code scanning from hurting the DB?

   Negative caching of misses plus per-IP rate limits on 404s.

4. Roughly how much storage do 12B links need?

   About 6 TB at ~500 bytes each, before replication.

© Interview Blueprints · licensed to the purchaser · do not redistribute