URL Shortener like TinyURL

TinyURL is a URL shortening service that converts a long web link into a short, easy-to-share link.

Example:

Long URL:
https://www.example.com/products/category/summer-sale/shoes?id=12345

TinyURL:
https://tinyurl.com/abcd12

Functional Requirements (WHAT the User should be able to do)

1. Shorten a long URL
   User provides a long URL
   System returns a short, unique URL
2. Redirect to original URL
   When a user visits the short URL
   System should redirect to the original long URL
3. Custom alias (optional but common)
   User can choose a custom short URL
   tinyurl.com/aniket
4. Link expiration (optional)
   Short URLs may expire after:

   Fixed duration
   One-time use
   Manual deletion

Non-Functional Requirements

1. Low latency redirects
   Redirection should be fast
   Target latency: ~200 ms or lower
   Reads are much more frequent than writes
2. Scalability
   Support 100M DAU
   Store and serve ~1 billion URLs
   Handle very high read traffic
3. Uniqueness guarantee
   Each short URL must uniquely map to one long URL
4. High availability
   Eventual consistency is acceptable

Core Entities

1. User
2. URL Mapping
   long_url
   short_code

APIs

1. Create Short URL API

Endpoint POST /api/v1/urls

Request Body
```
{
  "long_url": "https://example.com/long/path",
  "custom_alias": "aniket",          // optional
  "expires_at": "2026-01-01T00:00:00Z" // optional
}
```

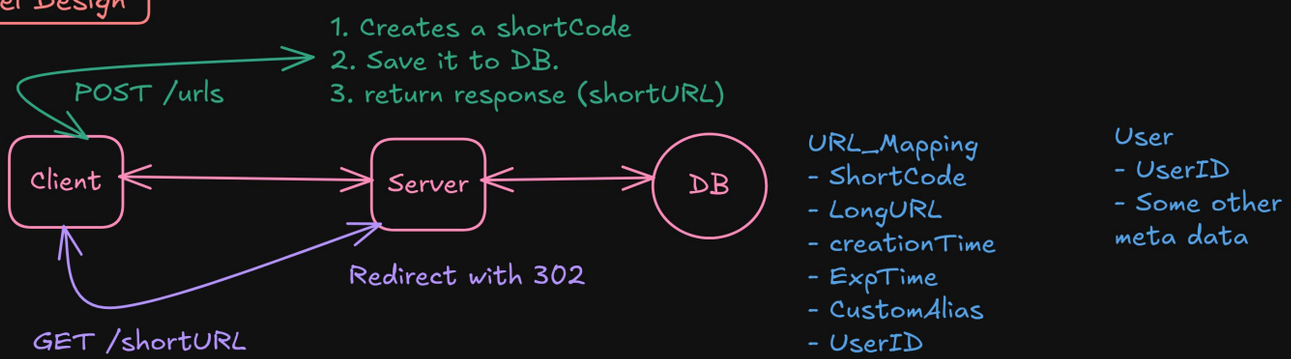
Response
```
{
  "short_url": "https://short.tiny/abc123",
  "expires_at": "2026-01-01T00:00:00Z"
}
```

2. Redirect API

Endpoint GET /{short_url}

Response HTTP 302 Redirect
Location: https://example.com/very/long/path

High Level Design



301 – Moved Permanently

-> Browser may cache the redirect
-> After first request, browser may not hit your server again

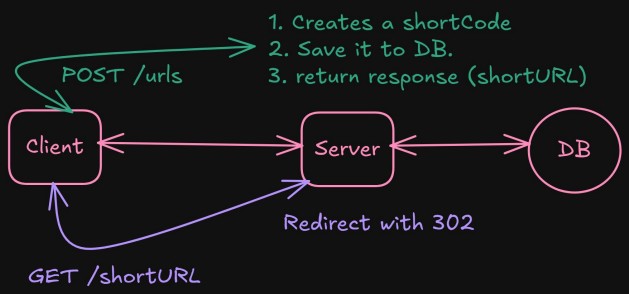
-> Analytics events may be missed (future requests bypass server)
-> Logging may be missed (no server hit → no logs)

302 – Found (Temporary Redirect)

-> Browser does not cache permanently
-> Every request hits your server

-> Accurate analytics (every click recorded)

-> Reliable logging (IP, timestamp, user-agent captured)



URL_Mapping
- ShortCode (PK)
- LongURL
- creationTime
- ExpTime
- CustomAlias
- UserID

User
- UserID
- Some other meta data

Custom alias

For custom aliases, we use the alias directly as the short code and rely on the database primary key for uniqueness.

If the alias already exists, we can either reject the request with a conflict or
fall back to generating a system-generated short URL.

Expired URLs

We store an optional expiry timestamp with each short URL.
During redirect, we compare the current server time with expires_at;

if the link has expired, we return 410, otherwise we redirect.

Background jobs can clean up expired entries asynchronously.

Deep Dives

How can we make URL redirection fast?

1. DB Indexing on short_code

shortCode VARCHAR PRIMARY KEY

What this gives us

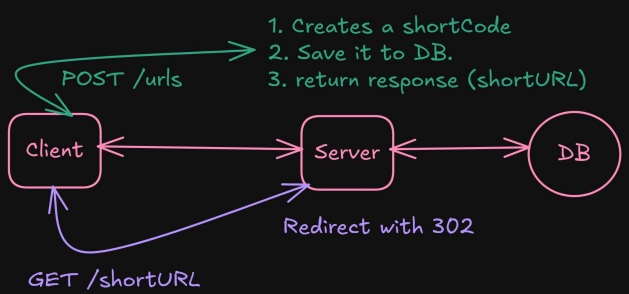
-> Uniqueness guarantee
-> Automatic index on primary key
-> Lookup time ≈ O(log N) (B-tree)
-> Reliable and strongly consistent

Why this works initially
-> Correctness is guaranteed
-> Simple design
-> Enough for low traffic

At scale, redirects are:

Read-heavy (99%)
Extremely high QPS
Latency-sensitive

So, DB Indexing Alone Is NOT Enough



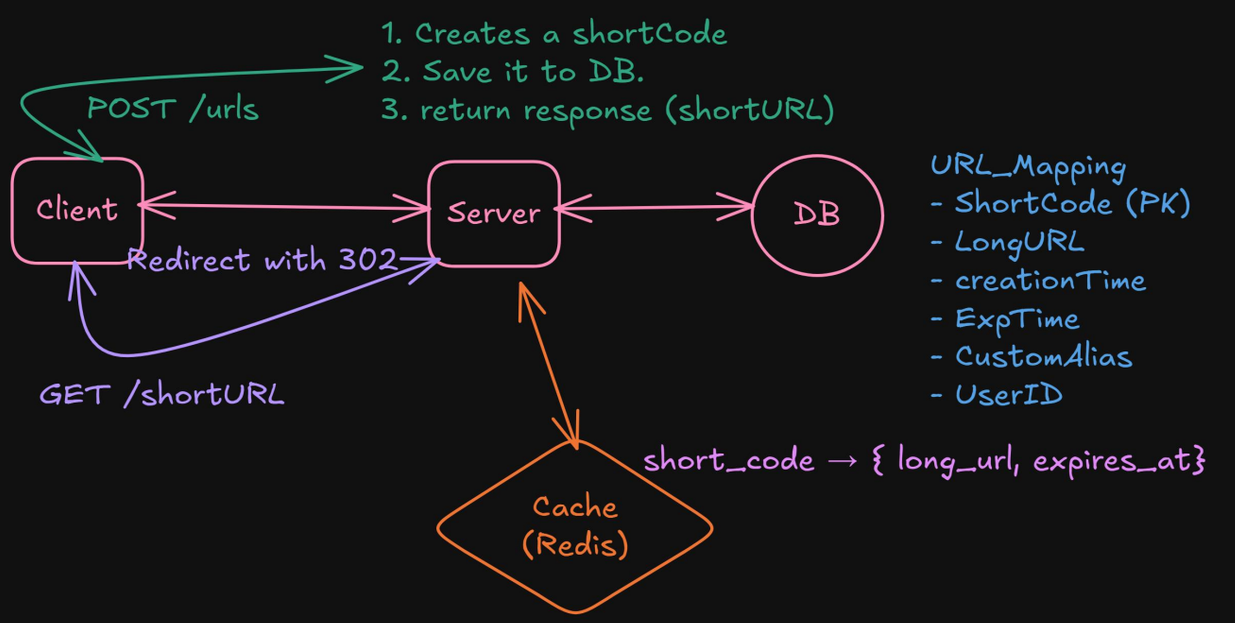
URL_Mapping
- ShortCode (PK)
- LongURL
- creationTime
- ExpTime
- CustomAlias
- UserID

User
- UserID
- Some other meta data

Introducing Redis Cache

-> In-memory storage
-> Sub-millisecond access
-> Perfect for key-value lookups

Cache Key




```
GET /{short_code}

if redis.get(short_code):
    return redirect

else:
    record = db.get(short_code)
    redis.set(short_code, record, TTL)
    return redirect
```

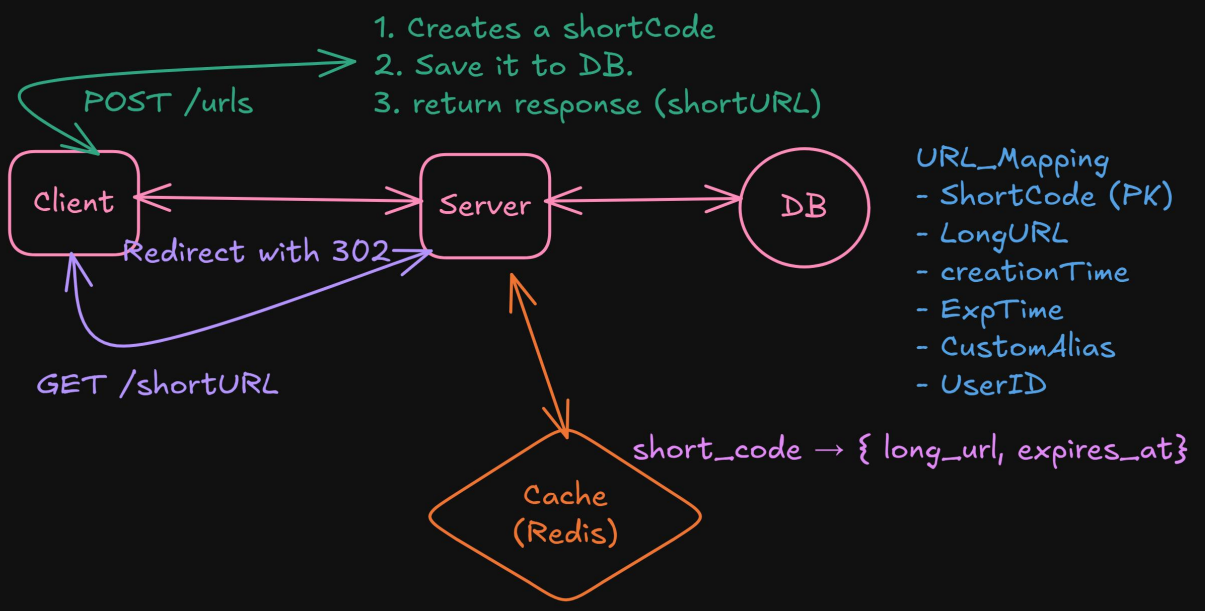
CDN / Edge Redirects

How it works

-> App sets redirect headers
-> CDN caches redirect response
-> Future requests handled at edge

Trade-off

-> Harder analytics



How do we ensure uniqueness of short URLs ?

1. Random Number Generation + Base62

When we choose a 6-character short code, we typically use Base62 encoding:

[0–9, A–Z, a–z] → 62 characters

6 characters → 62^6 → ~56 billion
7 characters → 62^7 → ~3.5 trillion

-> Generate a random number
-> Encode it in Base62

Pros

Simple to implement
Non-sequential (hard to guess)

Cons

Collision risk exists

How collisions are handled

Try inserting into DB

If short_code already exists → regenerate and retry

-> Works well with low collision probability + retries

2. Hashing the Long URL

-> Take the long URL as input

-> Generate an MD5 hash of the long URL
->Output is a hexadecimal string

-> Convert the hex hash into a number

-> Encode that number using Base62

-> Produces a URL-safe string (0–9, A–Z, a–z)

-> Take the first 6 characters as the short code

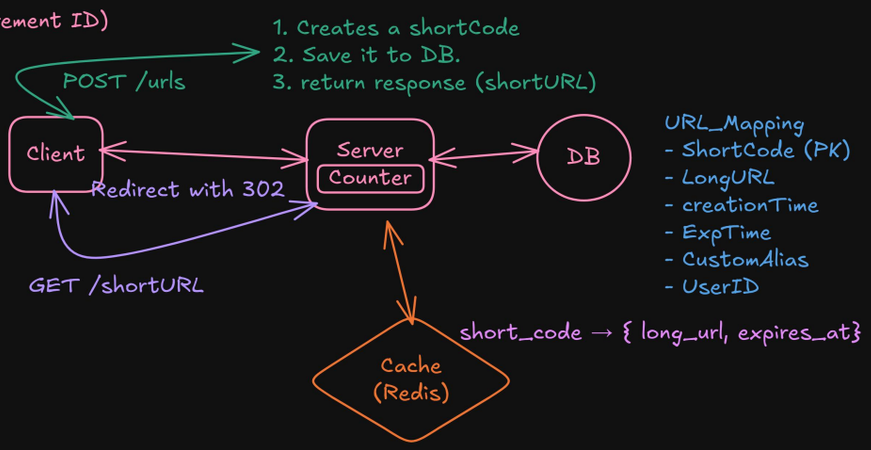
-> Check for collision in the database

-> If exists → add salt / retry

-> If not → store and return short URL

3. Counter-Based Approach

-> Maintain a global counter (auto-increment ID)
-> Convert counter value to Base62



Pros

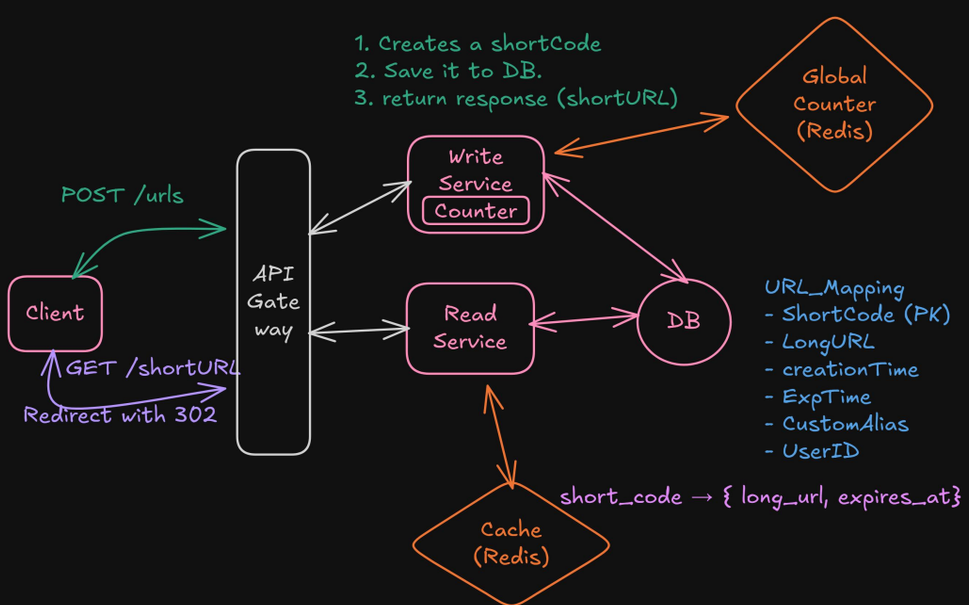
-> Guaranteed uniqueness
-> No collision checks needed
-> Very fast

Cons

-> Predictable
-> Easy to enumerate URLs
-> Potential security concerns

Mitigation

-> Add rate limiting
-> Don't shorten private URLs
-> Combine with permissions



Scalability

Support 100M DAU
Handle very high read traffic
Store and serve ~1 billion URLs

We separate reads and writes, scale them independently, and use Redis for speed + coordination.

Why?

URL shorteners are read-heavy
Redirect traffic ≫ URL creation traffic

-> API Gateway routes the traffic to correct service

-> We horizontally scale read and write services.

-> Global Counter using Redis (for Unique IDs)

-> Atomic operations (INCR)
-> Very fast
-> Works across multiple write servers

Write flow :
-> Write service calls Redis:
-> INCR global_counter

-> Redis returns a unique number
-> Convert it to Base62 → short_code
-> Store mapping in DB

Counter Batching (Range Allocation)

Instead of fetching 1 ID at a time, each write server:

-> Fetches a range of IDs
-> Uses them locally
-> Goes back to Redis only when exhausted

Example: batch size = 1,000
global_counter = 10,000,000

INCRBY global_counter 1000 --> 10,001,000

Range allocated:
10,000,001 → 10,001,000

Server uses local counter
Converts each ID → Base62
No Redis call per request

When batch is exhausted
-> Server repeats INCRBY
-> Gets a new range

What if server crashes mid-batch?
-> Some IDs are lost
-> No duplicates occur
-> Gaps in sequence are acceptable

-> URL shorteners care about uniqueness, not continuity

Typical choice for batch size: 1K–10K

Database

Estimating row size

Each URL mapping record typically contains:
short_code → ~6–8 bytes
long_url → ~120–150 bytes (URLs can be long)
created_at → ~8 bytes
expires_at → ~8 bytes
Flags & metadata (is_active, is_custom, alias, user_id, etc.) → ~100 bytes
Index + row overhead → ~100–150 bytes

To stay conservative, we can estimate: ~700 bytes per row

1B rows × 700 bytes ≈ 700 GB

This size is very reasonable for modern SSD-backed database systems.

Database choice

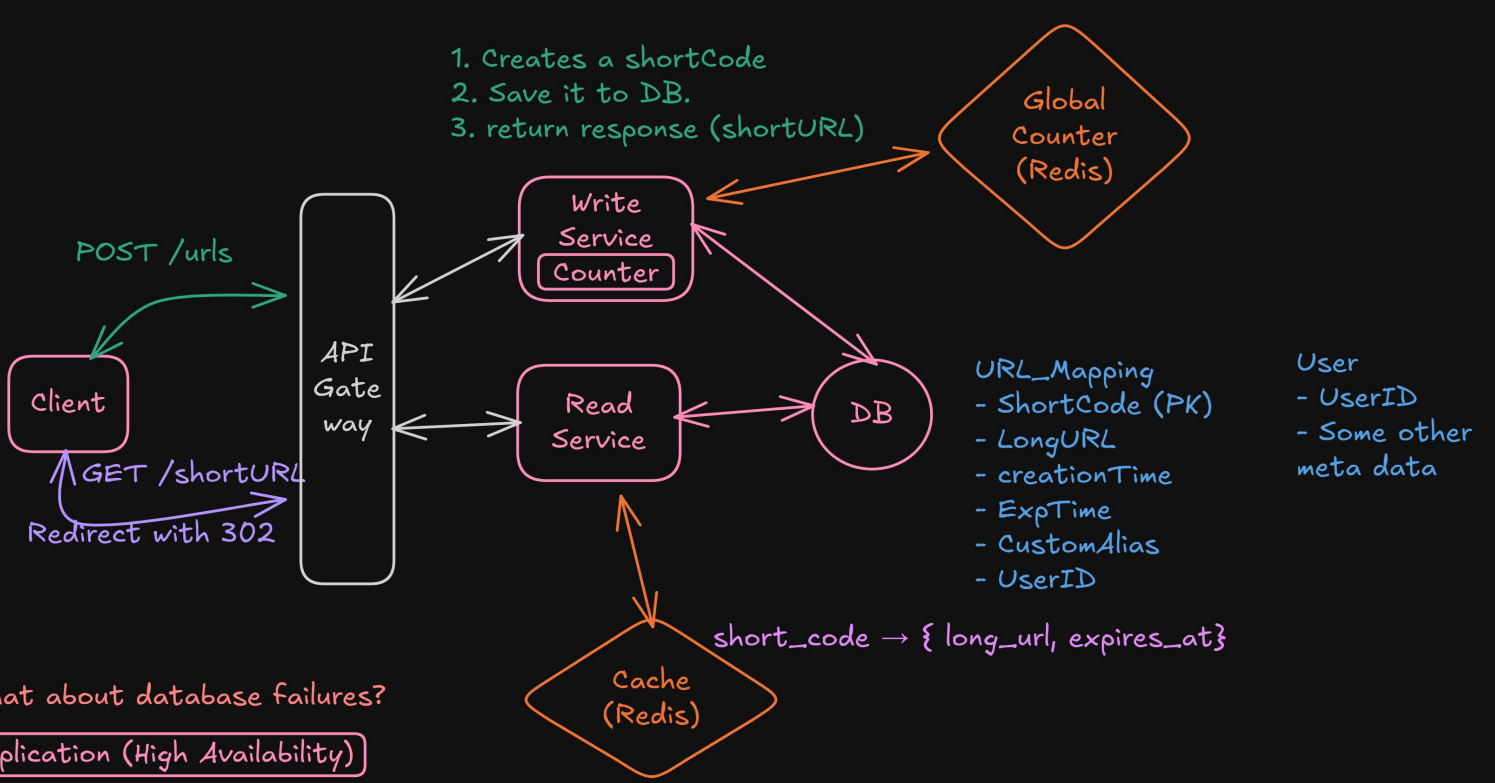
Many databases work here:

Postgres / MySQL (relational)
DynamoDB (managed NoSQL)
Cassandra

Because:

Simple key-value style access
Low write throughput
Cache absorbs reads

-> In interviews, choose what you're comfortable with.



What about database failures?

Replication (High Availability)

Maintain one primary database
One or more replicas
If primary fails → promote a replica

Periodic snapshots (daily / hourly)
Stored in separate storage (e.g., object storage)

Redis counter : why we need HA + AOF + WAIT

The one thing we must absolutely prevent is:
-> Re-issuing the same counter range after a failure

High availability

-> Redis supports replication with a primary–replica setup.
-> If the primary fails, a replica can be promoted automatically (via Sentinel or managed Redis).
-> This ensures that temporary node failures do not permanently take the counter service offline.

Durability (AOF)

-> Since Redis is in-memory, persistence is required to survive crashes and restarts.
-> Enabling Append Only File (AOF) ensures that every counter increment is recorded on disk.
-> When Redis restarts, it replays these operations and restores the counter to a value that never goes backward.

-> This guarantees that already-allocated ID ranges are not reused, even after a crash.

Replica safety (WAIT)

-> Replication in Redis is asynchronous. Without extra care,
a failover could promote a replica that is slightly behind, causing the counter to rewind.

-> Using the WAIT command after allocating a block ensures that
at least one replica has acknowledged the increment before it is considered committed.
This prevents stale replicas from handing out duplicate ranges during failover.

Why this works well with Counter batching

By allocating IDs in blocks (for example, 1,000 at a time), Redis is removed from the hot path:

Redis is contacted infrequently
App servers generate IDs locally
Short Redis outages do not immediately affect URL creation

Even if Redis is down temporarily, app servers can continue
using their local ranges.
The worst-case outcome is skipped IDs, never duplicates.

Result:
Redis + block allocation + HA + AOF + WAIT gives:

Strong correctness guarantees
High availability in practice
Very short URLs (6-7 Base62 characters)

Snowflake : why it's different (and its trade-offs)

Snowflake takes a fundamentally different approach:
instead of making a shared counter highly available, it eliminates the shared counter entirely.

How Snowflake works

Each application server generates IDs locally using:
-> timestamp + worker_id + sequence
There is no Redis, no database, and no runtime coordination.

Why Snowflake scales so well

No centralized dependency
No serialized writes
No failover impact
Capacity scales linearly with the number of app servers

Snowflake trade-offs

Snowflake is not free of cost:

Longer URLs:
A 64-bit Snowflake ID encoded in Base62 is typically 10–11 characters, compared to 5–6 with counters.

Worker ID management:
Each server must be assigned a unique worker ID (usually via deployment configuration or orchestration).

Snowflake trades shortness and simplicity for coordination-free scalability.

Redis vs Snowflake Conclusion

A naive Redis counter may not scale well, but with block allocation/counter batching
Redis becomes a low-frequency allocator and works reliably even at high DAU
while giving very short URLs.
Snowflake removes the allocator entirely and scales better architecturally,
but produces longer URLs.

The choice depends on whether URL length or coordination-free scaling is more important.

Why ZooKeeper is not a good fit

Using Apache ZooKeeper, it is technically possible to generate IDs
using sequential znodes. ZooKeeper provides extremely strong consistency guarantees.

However, ZooKeeper is the wrong abstraction for this problem.

Why ZooKeeper is a poor choice
-> ZooKeeper is designed for coordination, not data-path operations
-> All writes go through a single leader and require quorum agreement
-> Latency is significantly higher than Redis or local ID generation
-> Operational complexity is high (JVM tuning, disk guarantees, quorum management)

For a URL shortener:

ID generation is user-facing
Latency-sensitive
Expected to scale smoothly

ZooKeeper prioritizes correctness over throughput and simplicity,
which makes it overkill and inefficient for this use case.

Why UUID is usually not a good fit

UUID solves uniqueness and availability,
but fails the core product goal of a URL shortener: short, clean URLs.

UUID is great at what it was designed for:
Generated locally
Require no coordination
Have negligible collision probability
Extremely reliable at large scale

From a systems perspective, UUIDs are actually better than Redis counters.

The real problem: URL length

A standard UUID (v4) looks like this:
550e8400-e29b-41d4-a716-446655440000

That's:

128 bits
36 characters (with dashes)
~32 hex characters without dashes

Even if you:
Convert it to Base62
Remove dashes
Compress aggressively
You still end up with:

~20–22 Base62 characters

For a URL shortener, that's simply too long.