Experiment 1

Student Name: Sohit Kumar	UID: 23BCS12492
Branch: BE-CSE	Section: Group: KRG-3A
Semester: 6	Date of performance: 08-01-2026
Subject name: System Design	Subject code: 23CSH-314

AIM:

To design, implement, and evaluate a simple URL redirection system that converts long URLs into short URLs, and to measure system performance (latency and throughput) under varying request loads.

OBJECTIVE:

1. To understand the working of a URL shortening service.
2. To design REST APIs for URL creation and redirection.
3. To identify suitable database and server choices.
4. To analyze different short URL generation techniques.
5. To understand scalability challenges and their solutions.

PROBLEM STATEMENT:

Design a URL Shortener system that converts long URLs into short URLs and redirects users from short URLs to original long URLs with low latency, high availability, and scalability.

1. Requirements:-

A. Functional Requirements:-

1. Generate a short URL from a long URL.
2. Support custom URLs (optional).
3. Support URL expiration (default and custom).
4. Redirect short URL to the original long URL. 5. User registration and login using REST APIs.

B. **Non-Functional Requirements**:-

1. Low Latency: URL creation and redirection < 20ms
2. Scalability:
   - ➢ 100M daily active users
   - ➢ 1B shortened URLs
3. Availability: System should be available 24×7.
4. Uniqueness: Each short URL must be unique.
5. CAP Theorem:
   - ➢ Availability > Consistency

     Eventual Consistency model.

System Entities

1. User
2. Long URL
3. Short URL

2. API Design:-

1. Create Short URL Endpoint: POST /v1/url

Request Body:

```
{
  "longURL": "string",
  "customURL": "string (optional)",
  "expirationDate": "date"
}
```

Response:

```
{
  "shortURL": "https://short.ly/2bi"
```

}

2. Redirect Short URL

Endpoint: GET /v1/url/{shortURL}

Action: Redirects to original long URL.

3. User APIs

- POST /v1/register
- POST /v1/login

3. Database Design:-

Table T1 – User Metadata

Column	Description
user_id	Primary Key
name	User Name
email	Unique
password	Encrypted
created_at	Timestamp

Table T2 – URL Table

Column	Description
id	Primary Key
shortURL	Unique
longURL	Original URL
customURL	Optional
expirationDate	Expiry Time
user_id	Foreign Key

4. High Level Design (HLD) Components:-

1. Client (Browser / App)
2. Server (Business Logic)
3. Database (Storage)

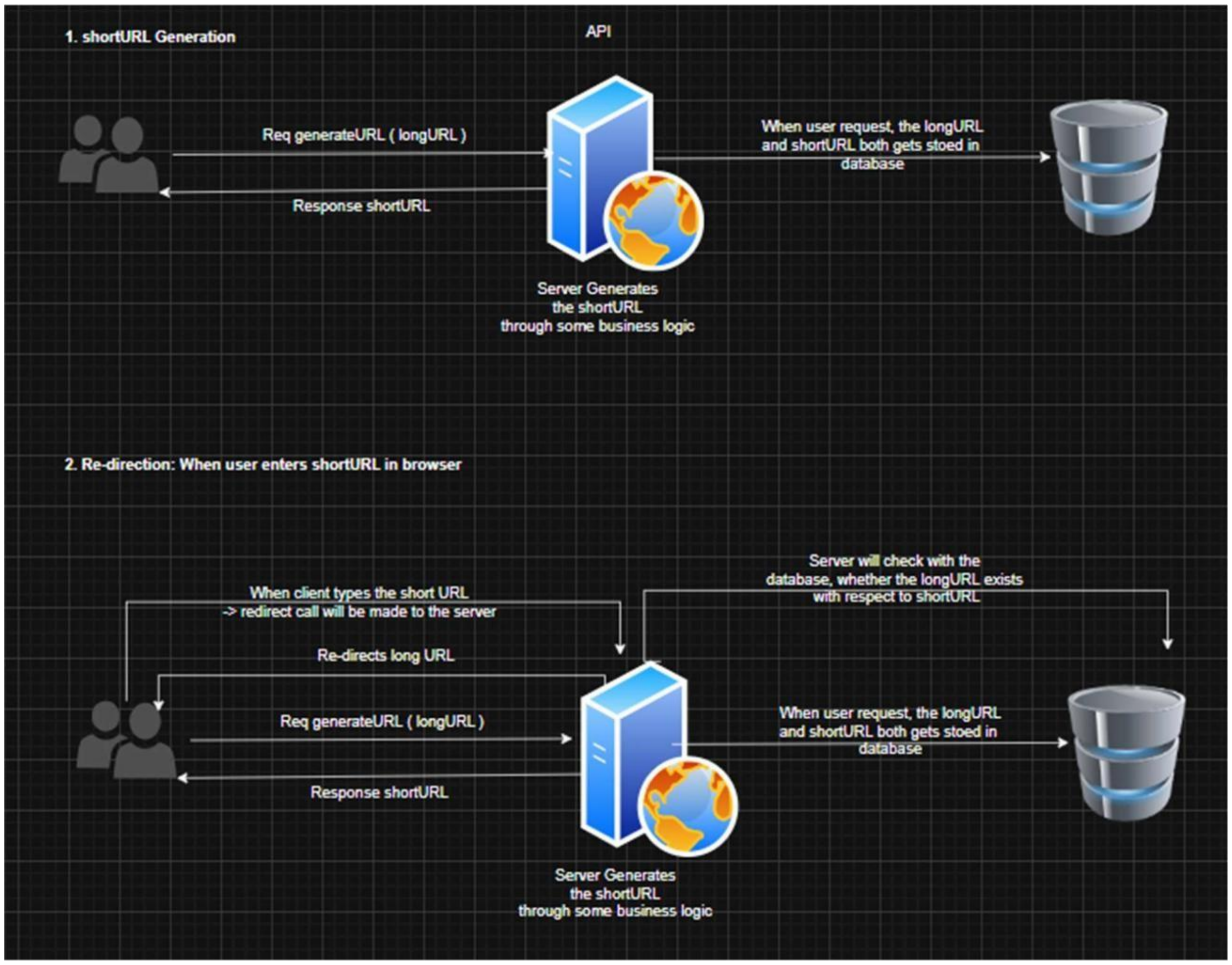
Flow:
Client → Server → Database → Server → Client

1. Workflow

1. Short URL Generation 1. Client sends long URL request.
2. Server generates short URL.
3. Server stores long URL + short URL in database.
4. Server responds with short URL.

2. Redirection

1. Client enters short URL.
2. Server validates short URL from database.
3. Server redirects to original long URL.


1. shortURL Generation
API
Req generateURL (longURL)
Response shortURL
When user request, the longURL and shortURL both gets stoed in database
Server Generates the shortURL through some business logic
2. Re-direction: When user enters shortURL in browser
When client types the short URL -> redirect call will be made to the server
Server will check with the database, whether the longURL exists with respect to shortURL
Re-directs long URL
Req generateURL (longURL)
Response shortURL
When user request, the longURL and shortURL both gets stoed in database
Server Generates the shortURL through some business logic

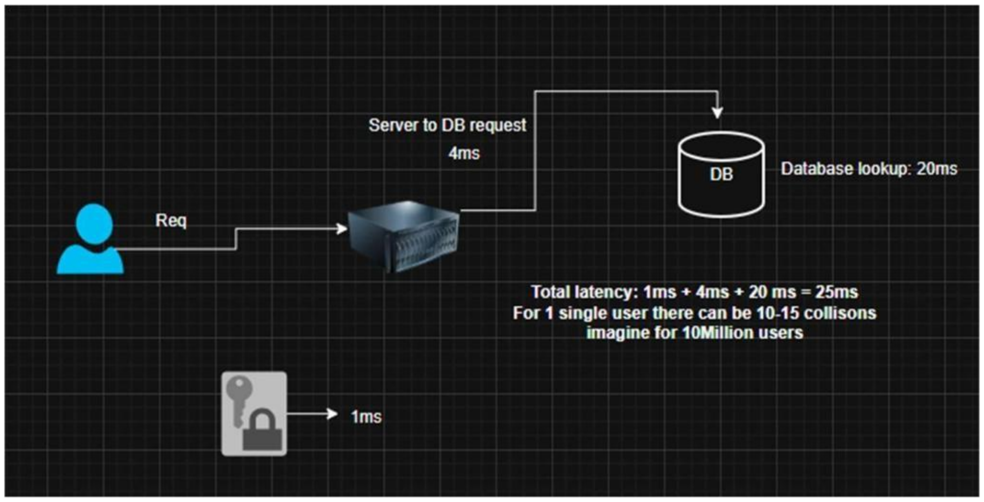
5. Low Level Design (LLD):-

Approach 01: Hashing (Encryption)

Method: shortURL = encrypt(longURL)
Algorithms: MD5, SHA-1, Base64

Problems:

1. Long hash length (not short).
2. High collision probability.
3. Multiple DB lookups.
4. High latency at scale.
   Not suitable for large-scale systems


Server to DB request
4ms
DB
Database lookup: 20ms
Req
Total latency: 1ms + 4ms + 20 ms = 25ms
For 1 single user there can be 10-15 collisons
imagine for 10Million users
1ms

<u>Approach 02:</u> Counter-Based Approach (Recommended)

Logic:

1. Generate a global counter value.
2. Convert counter to Base62.
3. Use Base62 value as short URL.

Example:

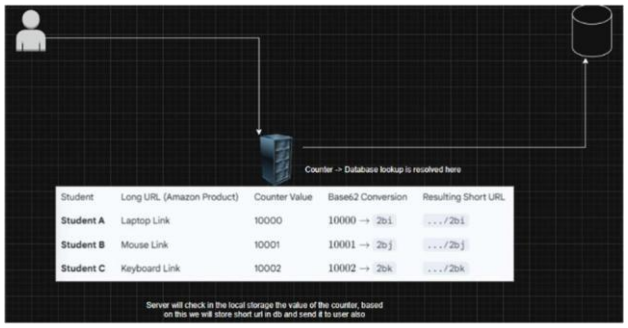
Counter = 10000

Base62 = 2bi

Short URL = short.ly/2bi

Advantages

- No collision
- Fast generation
- Predictable length



Student	Long URL (Amazon Product)	Counter Value	Base62 Conversion	Resulting Short URL
Student A	Laptop Link	10000	10000 → 2bi	.../2bi
Student B	Mouse Link	10001	10001 → 2bj	.../2bj
Student C	Keyboard Link	10002	10002 → 2bk	.../2bk

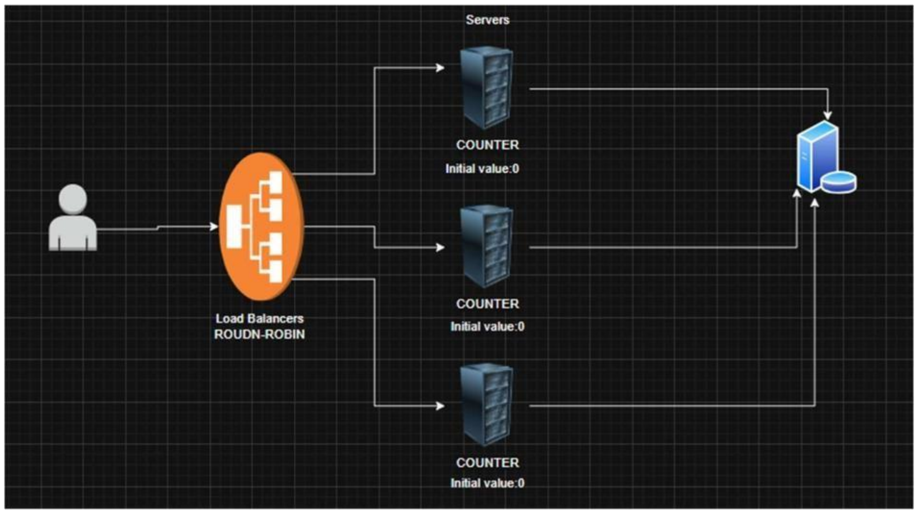
Server will check in the local storage the value of the counter, based on this we will store short url in db and send it to user also

Scaling Challenges & Solutions

Problem 1: Monolithic Server

- Cannot handle 100M users

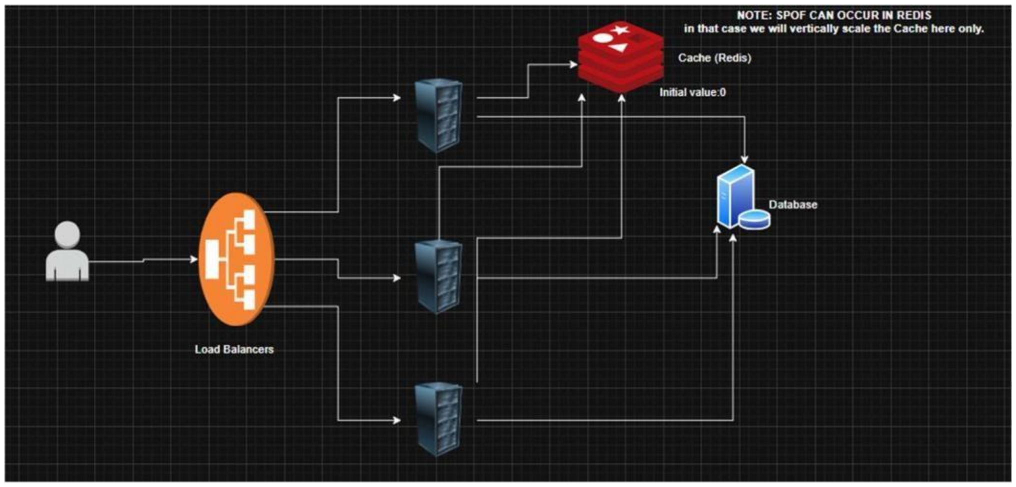
Solution: Horizontal Scaling (Multiple Servers)



Problem 2: Distributed Counter Conflict

- Multiple servers may generate same counter.

Solution:
Use Global Counter in Cache (Redis) with atomic increment.


NOTE: SPOF CAN OCCUR IN REDIS
in that case we will vertically scale the Cache here only.

Type of Database Used

Recommended

➢ NoSQL (Key-Value Store)

- ➢ Redis (Cache)
- ➢ DynamoDB / Cassandra (Persistent)

Reason:

- ➢ Fast read/write
- ➢ High availability
- ➢ Easy horizontal scaling

Type of Server

- Stateless REST API Server □ Deployed behind Load Balancer
- Performs:
  - ➢ URL validation
  - ➢ Counter fetch
  - ➢ Base62 conversion
  - ➢ DB read/write

Steps to Do Experiment

Step 1: Set Up the Project

bash

```
mkdir url-shortener && cd url-shortener
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
pip install flask
```

Step 2: Write the URL Shortener Code

Create a file app.py:

```
python

from flask import Flask, redirect, request, jsonify

import hashlib

import time

import threading

from collections import defaultdict


app = Flask(_name_)

url_map = {}

metrics = {'total_requests': 0, 'latencies': []}


def generate_short_code(long_url):

    return hashlib.md5(long_url.encode()).hexdigest()[:8]


@app.route('/shorten', methods=['POST'])

def shorten():

    start_time = time.time()

    long_url = request.json.get('url')

    short_code = generate_short_code(long_url)

    url_map[short_code] = long_url

    latency = time.time() - start_time

    metrics['latencies'].append(latency)

    metrics['total_requests'] += 1

    return jsonify({'short_url': f'http://localhost:5000/{short_code}'})
```

```
@app.route('/<short_code>')
def redirect_url(short_code):
    start_time = time.time()
    long_url = url_map.get(short_code)
    if long_url:
        latency = time.time() - start_time
        metrics['latencies'].append(latency)
        metrics['total_requests'] += 1
        return redirect(long_url, code=302)
    return 'URL not found', 404


@app.route('/metrics')
def get_metrics():
    if metrics['latencies']:
        avg_latency = sum(metrics['latencies']) / len(metrics['latencies'])
        throughput = metrics['total_requests'] / (time.time() - app_start_time)
    else:
        avg_latency = throughput = 0
    return jsonify({
        'avg_latency_sec': avg_latency,
        'throughput_req_per_sec': throughput,
        'total_requests': metrics['total_requests']
    })


def load_test(concurrent_users=10):
    import requests
    def worker():
        requests.post('http://localhost:5000/shorten', json={'url':
'https://www.example.com/page?q=test'})
    threads = [threading.Thread(target=worker) for _ in range(concurrent_users)]
    for t in threads: t.start()
    for t in threads: t.join()


if _name___== '_main_':
    app_start_time = time.time()
    app.run(debug=True)
```

Step 3: Run the Server

bash

```
python app.py
```

Step 4: Test the Endpoints

Use Postman or curl to test:

POST to http://localhost:5000/shorten with JSON {"url": "https://www.example.com"}

GET to http://localhost:5000/<short_code> to redirect

GET to http://localhost:5000/metrics to see performance data

Step 5: Perform Load Testing

Modify the load_test() function to simulate different loads (10, 50, 100 users).

Run load tests and record metrics from /metrics endpoint.

Step 6: Record Observations

Create a table to compare latency and throughput for different loads.

Sample Input

json

```
{
  "url": "https://www.google.com/search?q=system+design+course"
}
```

Sample Output

After shortening:

json

```
{
  "short_url":  "http://localhost:5000/a1b2c3d4"
}
```

Metrics after 100 requests with 10 concurrent users:

json

```
{
  "avg_latency_sec": 0.012,
  "throughput_req_per_sec":  45.7,
  "total_requests":  100
}
```

RESULT:-

A scalable and highly available URL Shortener System was successfully designed using counter-based Base62 encoding, REST APIs, NoSQL database, and distributed architecture principles.

CONCLUSION:-

The counter-based approach with centralized atomic counters provides the most efficient, scalable, and collision-free solution for a URL Shortener system handling millions of users with low latency.