Experiment 1

Student Name: Sandeep Kumar
UID: 23BCS11489
Branch: CSE
Section/Group: KRG 3-B

Semester: 6th
Date of Performance:09/01/2026

Subject Name: System Design
Subject Code: 23CSH-314

1. Aim:

To design and evaluate a URL shortener system that transforms long URLs into short, unique links while maintaining high availability, scalability, low latency, and fast redirection. The system also includes optional features such as custom short URLs, link expiration, and user authentication.

2. Objective:

- To understand the functional and non-functional requirements of a large-scale distributed system.
- To design RESTful APIs for creating and redirecting URLs. ○ To identify key entities such as users, short URLs, and long URLs.
- To analyse CAP theorem trade-offs and apply eventual consistency where needed. ○ To design both high-level and low-level architectures for a scalable URL shortener system.
- To explore and compare different approaches for generating short URLs based on performance.

3. Tools Used:

- **Python** – Used for backend logic and implementing URL generation algorithms.
- **Flask** – A lightweight web framework for building RESTful APIs.
- ○ **Draw.io** – Used to design high-level and low-level system architecture diagrams.
- **Visual Studio Code** – Used as the development environment for writing and testing the code.

4. System Requirements:

A. Functional Requirements

- The system should allow users to convert a long URL into a unique, shortened URL. ○ The system should support premium users with additional features, including:
  - The option to create a custom short URL of their choice.
  - The ability to set an expiration date after which the short URL becomes inactive.

- When a user accesses a short URL, the system should efficiently redirect them to the original long URL.

- The system should ensure that only valid and active short URLs are redirected, and expired or invalid links are handled appropriately.

B. Non – Functional Requirements

The system is expected to support 100 million daily active users with approximately 1 million users generating short URLs each day.

- The system must maintain low latency, with URL shortening and redirection operations completing within 20 ms.

- The service should be highly available, operating 24×7 without downtime.

- The system should ensure data consistency, especially to maintain correct mappings between short URLs and long URLs.

- The architecture must be highly scalable, supporting both vertical and horizontal scaling to handle increasing traffic.

- Every generated short URL must be globally unique, with no collisions across the system.

5. API Design:

Pre-defined HTTP Methods:

- **GET** – Used to fetch data from the database.
- **POST** – Used to insert new data into the database.

- **PUT / PATCH** – Used to update existing data.
- **DELETE** – Used to remove data from the database.

Local Host Server: https://127.0.0.1/shorten

1. POST API – Create Short URL Endpoint: POST /shorten

Request Body:

```
{
  "url": "long_url",
  "custom_url": "custom_code",
  "expiry_date": "expiry_date"
}
```

Logic:

Accepts a long URL as input.

- Optionally allows a custom short URL and an expiration date (for premium users).
- Generates a unique short code if a custom URL is not provided.

Response:

```
{
  "short_url": "https://127.0.0.1/123ABC",
  "short_code": "123ABC"
}
```

2. GET API – Redirect to Long URL

Endpoint:
GET /<short_code>

Example:
https://127.0.0.1/123ABC

Logic: ○ Fetches the corresponding long URL from the database using the short code. ○ Redirects the user to the original long URL if it exists and is not expired.

6. Database Schema:

Table 1: User

- id – Unique identifier for the user ○ name – User's name ○ phone – User's phone number ○ other details – Additional userrelated information

Table 2: URL Mapping

- id – Unique identifier for the URL entry ○ long_url – Original long URL ○ short_url – Generated short URL ○ custom_url – Custom short URL (optional) ○ expiry_date – Expiration date of the short URL
- user_id – Reference to the user who created the short URL

7. High-Level Design (HLD) :

The system is based on a Client–Server–Database architecture:

- The client sends a request to create a short URL or to access an existing short URL. ○ The server handles the business logic, including short URL generation and validation. ○ The database stores the mapping between short URLs and their corresponding long URLs. ○ During redirection, the server retrieves the original long URL from the database and redirects the user accordingly.

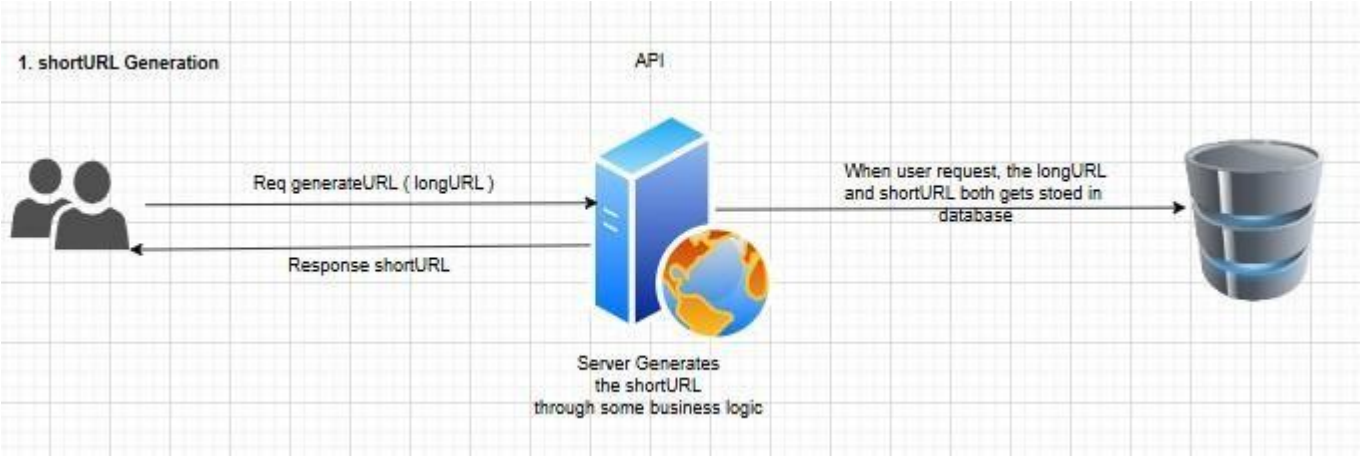


Fig. (a): shortURL Generation

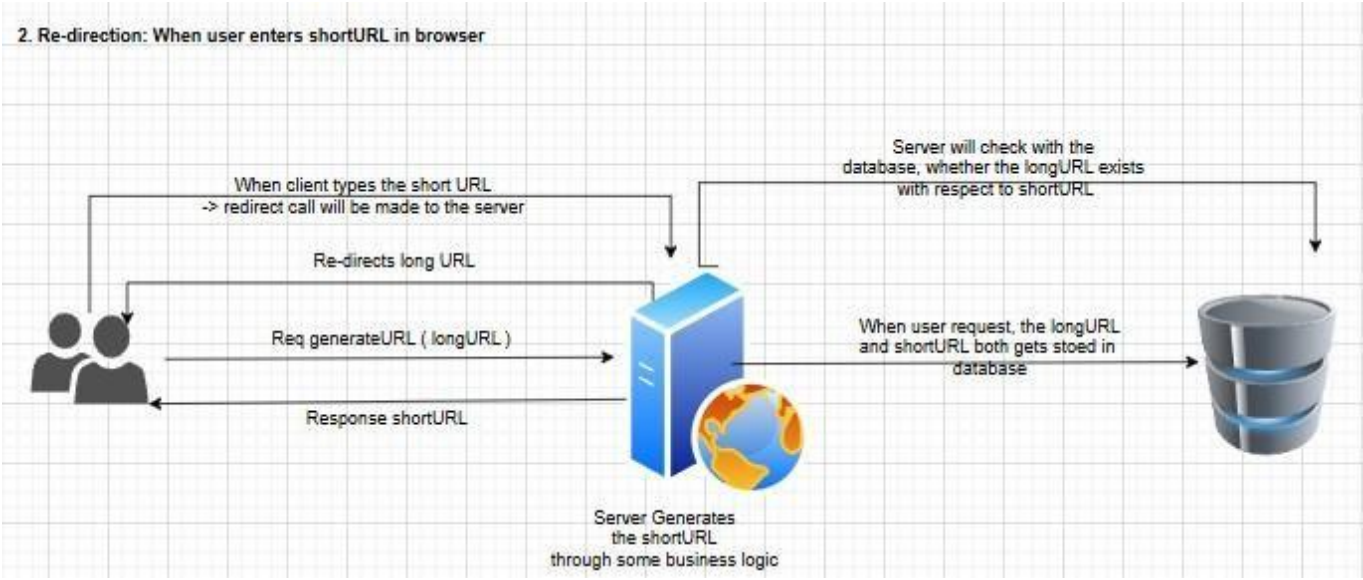


Fig. (b): Re-direction: When user enters shortURL in browser

8. Low- Level Design (LLD):

In the High-Level Design, we have only outlined the core components such as the client, server, and database. However, we have not yet specified important design details, including:

- The type of database required (for example, SQL vs NoSQL) and the reasons for choosing it. ○ The type of server needed and the kind of computations and business logic that will be handled within the server, such as URL generation, validation, caching, and redirection logic.

These details are addressed in the low-level design, where implementation choices and internal workflows are clearly defined.

Approach 01:

The Method: Encryption/Hashing
The system uses algorithms like **MD5** or **SHA1** to turn the long link into a unique string.

The Problems

- **Length:** Encrypted strings are naturally too long for a "short" URL.
- **Collisions:** If you try to shorten the string (e.g., taking only the first 4 characters), different long URLs might end up with the same 4-character code.

- **Database Lag:** To fix collisions, the server must check the database every time to see if a code is already taken. This causes **high latency** (slowness) because:
    1. It requires a full table scan.
    2. Multiple checks may be needed if collisions keep happening.

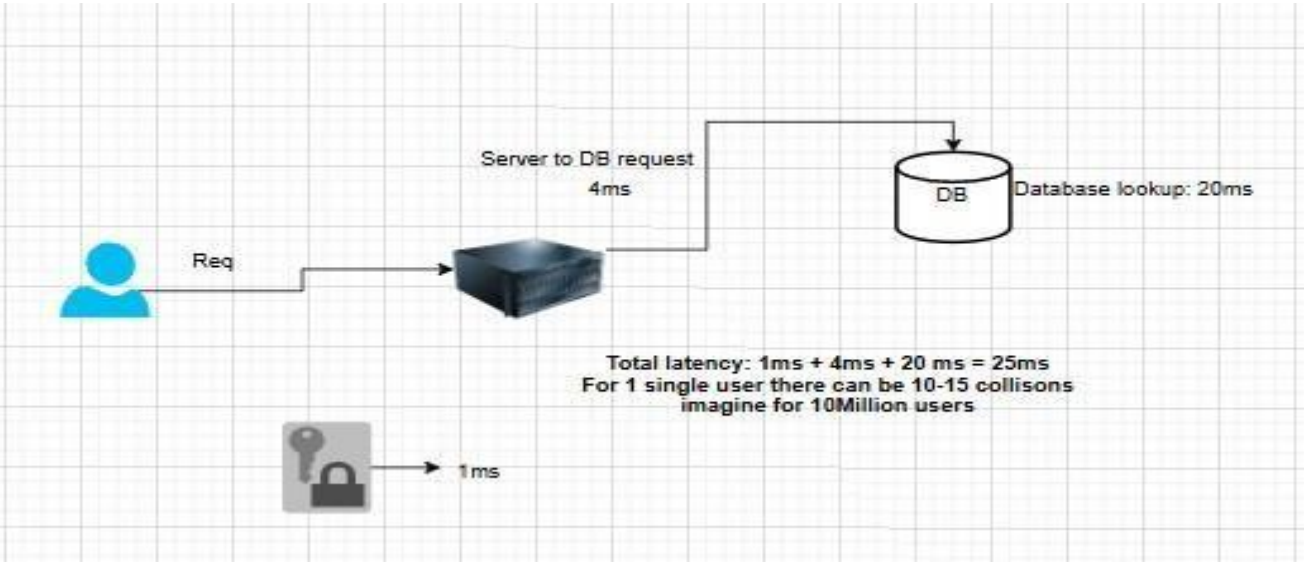


Fig. (c): Approach (1):- Encryption

Approach 02:

The system uses a **Global Counter** to create unique short links. ○ Each time a user submits a long URL, the server increments a number (e.g., 10000 becomes 10001). ○ This number is then converted into a short string (the "short URL"). ○ This ensures that every URL gets a unique ID and no two links are the same.

The Problem:

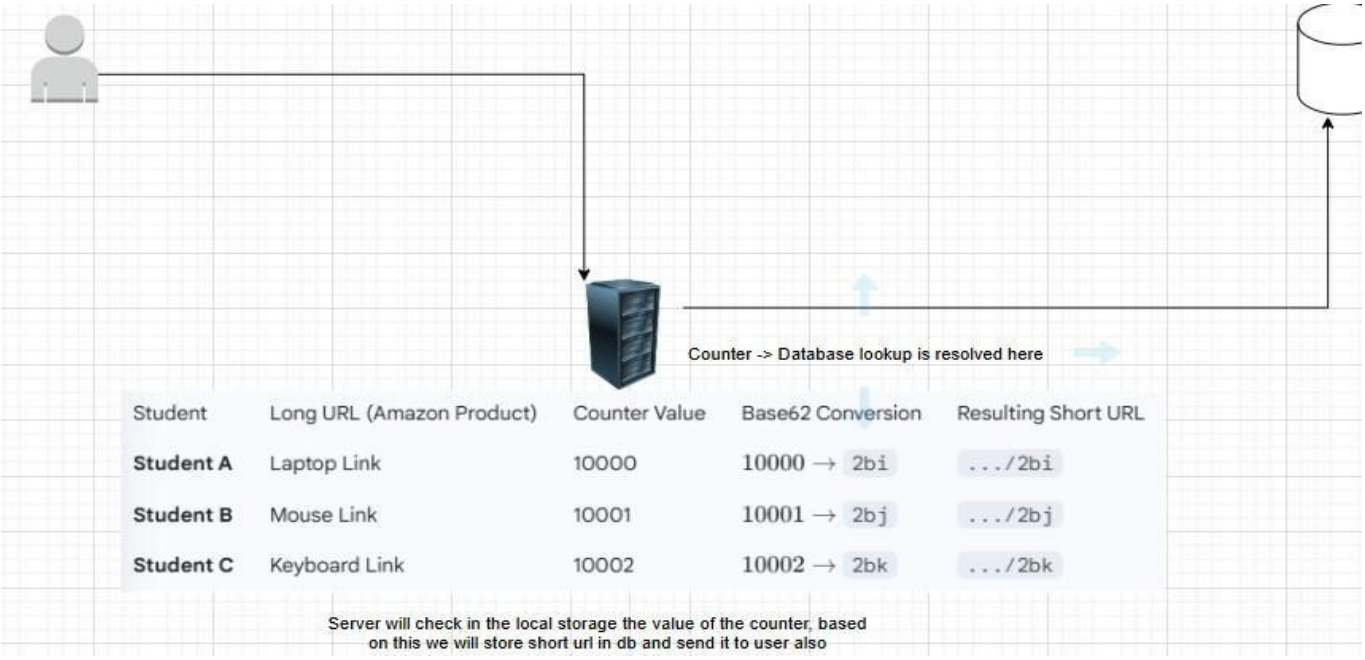
The current design is a **Monolith**, meaning it relies on a single server and a single counter.

- **Bottleneck:** A single server cannot handle 100 million users.
- **Concurrency:** If multiple users hit the server at the exact same millisecond, the counter might fail to update correctly, or the server might crash under the heavy load.

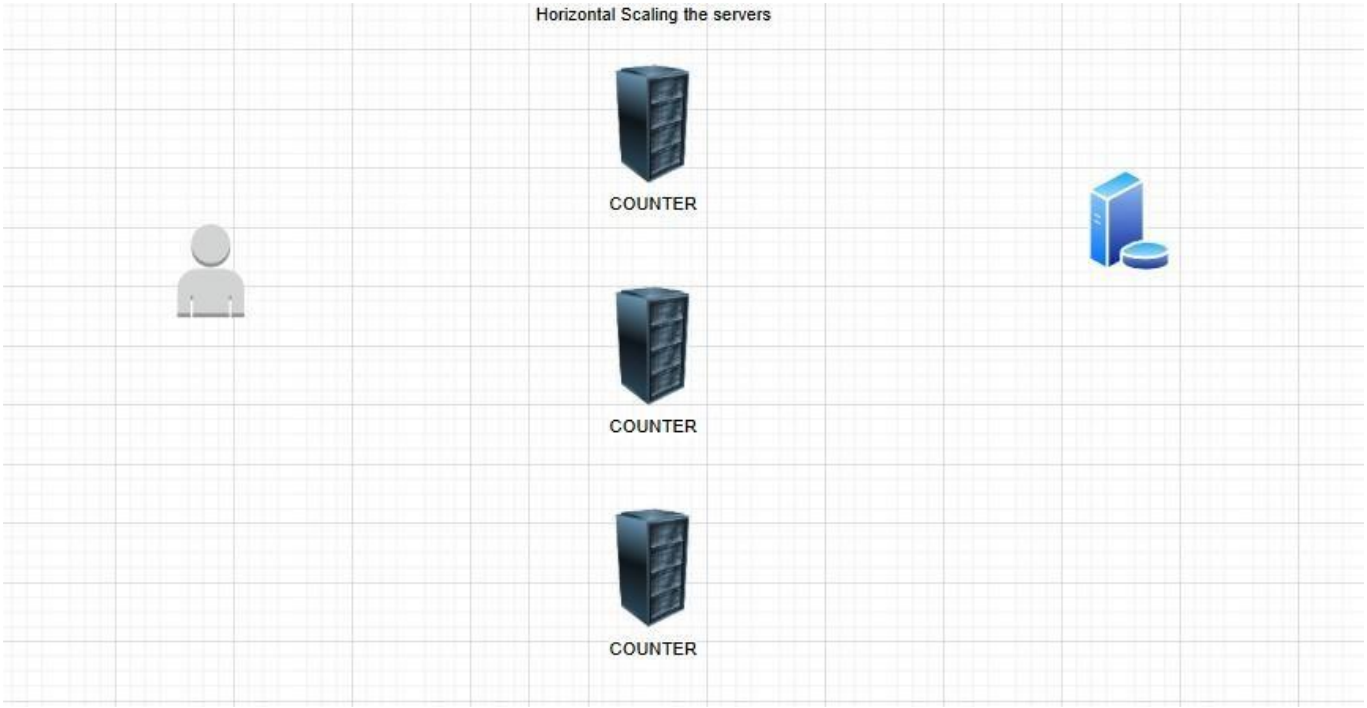
Resolution:

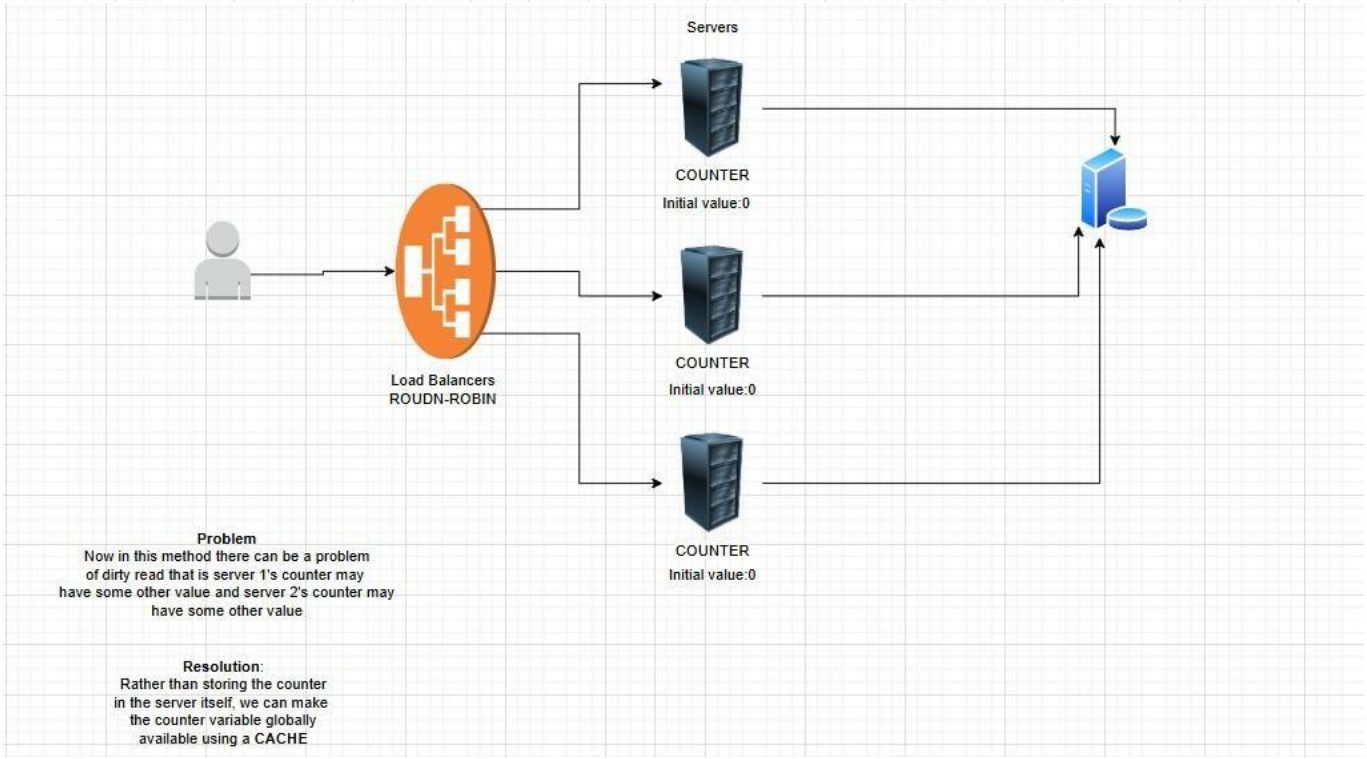
Since one giant server (**Vertical Scaling**) isn't enough for 100 million people, you must use **Horizontal Scaling**.

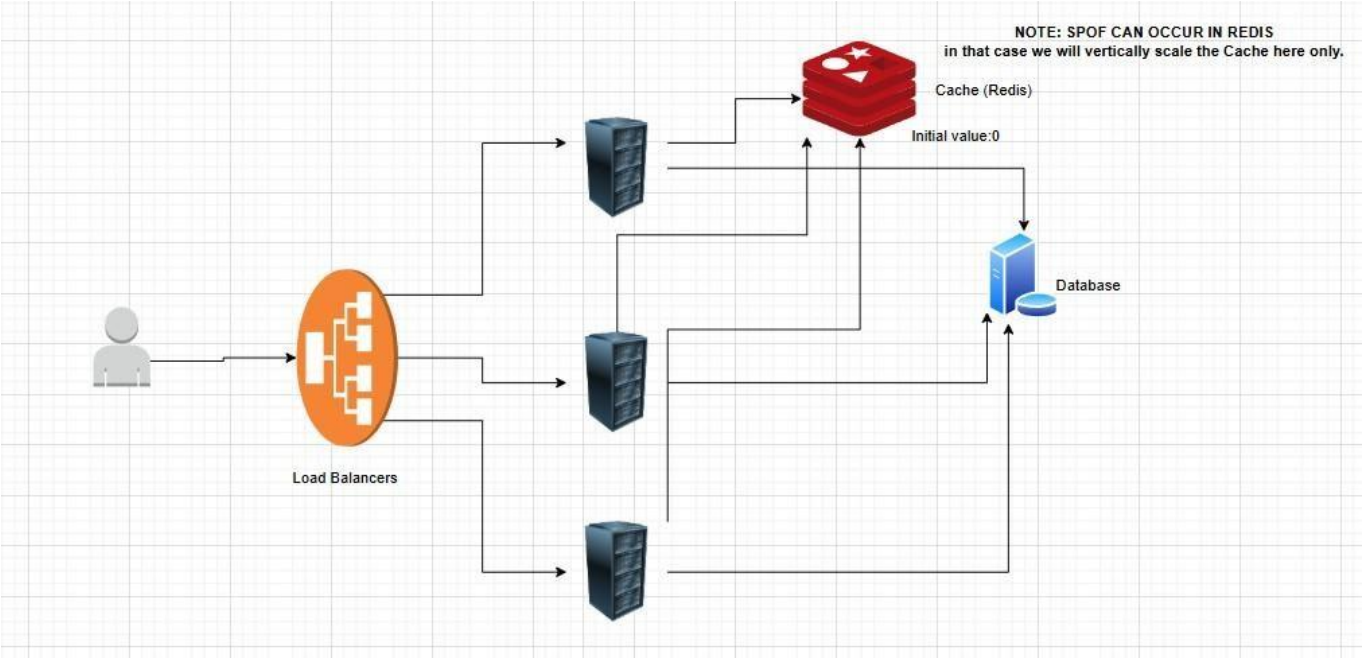
- You "scale out" by adding many smaller servers to handle the traffic.
- To make this work, the system usually switches to a **Distributed Counter** method (often using a "Range Handler"). Instead of every server asking for the next number one by one, each server is given a "block" of numbers (e.g., Server A gets 1–1000, Server B gets 1001–2000) so they can work independently without crashing.



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


Horizontal Scaling the servers
COUNTER
COUNTER
COUNTER


Servers
COUNTER
Initial value:0
COUNTER
Initial value:0
COUNTER
Initial value:0
Load Balancers
ROUDN-ROBIN
Problem
Now in this method there can be a problem of dirty read that is server 1's counter may have some other value and server 2's counter may have some other value
Resolution:
Rather than storing the counter in the server itself, we can make the counter variable globally available using a CACHE



Fig(c): Improved Versions

9. Scalability Solution:

- The system uses horizontal scaling with multiple application servers behind a Round Robin load balancer.
- A Redis cache maintains a centralized counter to generate unique short URLs using atomic increments for fast and conflict-free access.
- The database stores the final mapping between short URLs and long URLs. ○ For 100 million users, a monolithic architecture would require extreme vertical scaling on a single server, which is costly and limited. ○ A distributed architecture with multiple servers (S1, S2, S3, S4) enables horizontal scaling, better performance, and higher availability.

10. Learning Outcomes:

- Gained experience in designing a real-world, scalable system. ○ Understood REST API design and distributed system concepts such as the CAP theorem.
- Learned different URL shortening techniques and their trade-offs. ○ Developed an understanding of scalability, caching, load balancing, and the need for low latency and high availability.