WEEK-4

Designing a URL Shortener Service:

1. Introduction: A URL shortener is a service that takes a long URL and returns a shorter, unique alias that redirects to the original URL**.**
These services have become increasingly popular with the rise of social media platforms with character limits and the need for cleaner, more shareable links.
In this document, I will walk through the process of designing a scalable and efficient URL shortener service that can handle millions of URLs, provide fast redirections, and ensure high availability.

1.1 Purpose

The purpose of this document is to provide a comprehensive technical overview of designing and implementing a scalable URL Shortener Service. The goal is to describe the architecture, components, and technologies required to ensure high availability, low latency, and seamless scaling, while handling millions of requests per day. This document will serve as a guide for developers and system architects involved in creating and maintaining the service.

1.2 Overview

A URL shortener is a service that takes a long URL and converts it into a short, unique alias that redirects users to the original URL. Such services have become popular for sharing URLs on platforms that limit character counts, such as social media. Additionally, shortened URLs provide a cleaner, more shareable format. This document outlines the design considerations for building a robust URL shortener capable of handling high traffic, offering fast redirection, and ensuring long-term durability and security.
The system must support the following features:

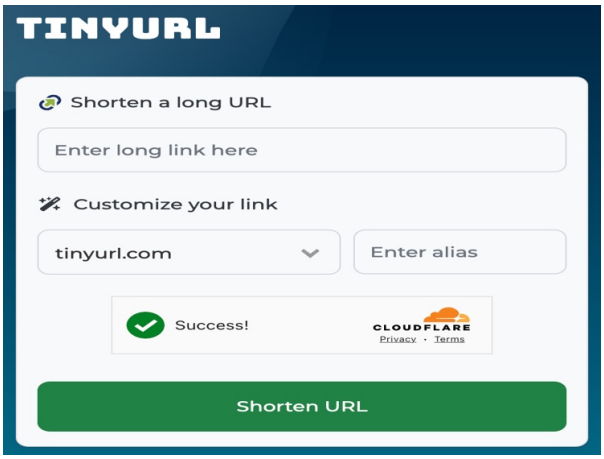
- Short URL generation
- Redirection from short URL to long URL
- Optional link expiration
- High availability, scalability, and security

1.3 Scope

This document focuses on:

- Defining the functional and non-functional requirements of the service.
- Estimating capacity needs including storage, bandwidth, and traffic handling.
- Designing a high-level architecture that covers the necessary components such as application servers, load balancers, databases, and caching layers.
- Describing performance optimization techniques and security considerations.
- Addressing key scalability challenges, such as database sharding and caching strategies.
- Outlining testing, monitoring, and deployment strategies.

This document is intended for engineers responsible for the design, development, and scaling of the URL shortener service.



2. Requirements

2.1 Functional Requirements

The system must satisfy the following core functionality:

- **Generate unique short URLs**: Given a long URL, the service should generate a unique, shorter version.
- **Redirect to the original URL**: When a short URL is accessed, the user should be redirected to the original URL quickly and seamlessly.
- **Custom aliases (optional)**: Users should be allowed to customize their short URLs (e.g., brand-specific URLs).
- **Link expiration (optional)**: URLs can be set to expire after a certain period, after which the short URL is invalid.
- **Analytics on URL usage (optional)**: Track how many times a link was clicked, where it was clicked from, and other relevant usage metrics.

2.2 Non-Functional Requirements

- **High availability**: The service must be available 99.9% of the time to avoid service disruption.
- **Low latency**: URL shortening, and redirection should occur in milliseconds to provide a smooth user experience.
- **Scalability**: The service should handle millions of requests per day and be able to scale as needed.
- **Durability**: URLs should remain functional for extended periods (years) without breaking.
- **Security**: The system must implement measures to prevent malicious use, such as phishing or spamming.

3. Capacity Planning

3.1 Traffic Estimation

To ensure that the system can handle peak loads and daily traffic,
we need to estimate the read and write throughput:

- Daily URL Shortening Requests: Approximately 1 million requests per day.
- Read

Ratio: For every URL shortening request, there will be about 100 redirects (100:1).

- Peak Traffic: Peak traffic is assumed to be 10 times the average load.

Throughput Requirements:

- Average Writes Per Second (WPS):
    - 1,000,000 requests / 86,400 seconds = ~12 WPS.
    - Peak WPS: 12 × 10 = 120 WPS.
- Average Redirects Per Second (RPS):
    - 12 WPS * 100 = 1,200 RPS.
    - Peak RPS: 120 WPS * 100 = 12,000 RPS.

3.2 Storage Estimation

For each shortened URL, the system needs to store several pieces of information:

- Short URL: 7 characters (Base62 encoded).
- Original URL: ~100 characters on average.
- Creation Date: 8 bytes (timestamp).
- Expiration Date: 8 bytes (optional).
- Click Count: 4 bytes (integer).

Storage per URL:

- Short URL: 7 bytes
- Original URL: 100 bytes
- Timestamps: 16 bytes (8 bytes each for creation and expiration)
- Click count: 4 bytes
- Total storage per URL = 127 bytes.

Annual Storage:

- Total URLs per year: 1,000,000 (requests/day) × 365 (days) = 365,000,000 URLs.
- Total storage per year: 365,000,000 URLs × 127 bytes ≈ 46.4 GB/year.

3.3 Bandwidth Estimation

Each redirect involves an HTTP 301 response with headers, estimated at around 500 bytes per response.

Daily Bandwidth:

- Total redirects/day: 100,000,000 (100 redirects per URL shortened).
- Daily bandwidth: 100,000,000 × 500 bytes = 50 GB/day.

Peak Bandwidth:

- Peak traffic: 10x average traffic.
- Peak bandwidth: 500 bytes × 12,000 RPS = 6 MB/s.

3.4 Caching Estimation

Caching is crucial for performance, especially in a read-heavy system.

Cache Size:

- Hot URLs: 20% of URLs are responsible for 80% of the traffic.
- Cache memory required:
  - 20% of 1 million writes/day = 200,000 URLs.
  - 200,000 URLs × 127 bytes = 25.4 MB/day.

Cache Hit Ratio:

- Expected cache hit ratio: 90%.
- Remaining load on the database: 10% of requests will bypass the cache and hit the database, equating to around 120 RPS (for redirect requests).

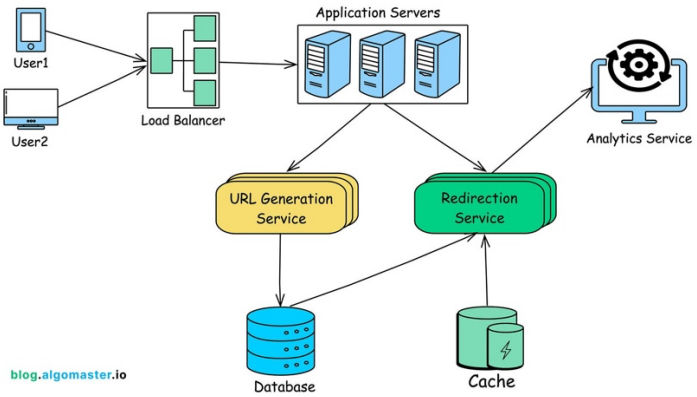
4. System Design

The URL shortener service must be designed to handle millions of requests per day while maintaining high availability, low latency, and scalability.

4.1 High-Level Architecture

The system will be divided into several components, each responsible for a specific function. These components will interact with each other to form a cohesive, scalable, and robust service:

- **Load Balancer**: Distributes incoming requests across multiple application servers.
- **Application Servers**: Handle URL shortening requests and redirections.
- **URL Generation Service**: Generates unique, short URLs and handles link expiration and custom aliases.
- **Redirection Service**: Retrieves the original URL for a given short URL and redirects users.
- **Database**: Stores mappings between short URLs and original URLs.
- **Cache**: Stores frequently accessed short URLs to reduce latency.
- **Analytics Service** (optional): Tracks usage data such as click counts and geographic distribution.



The diagram above (placeholder) shows a basic URL shortener architecture with load balancing, caching, and database layers.

4.2 Load Balancer

The **load balancer** is a critical component that evenly distributes incoming traffic across multiple **application servers**. This ensures that no single server is overwhelmed, and it provides redundancy in case any server fails.

- **Key Features**:
  - **Even Load Distribution**: Requests are spread evenly across the available servers to ensure scalability and availability.
  - **Failover**: If a server goes down, the load balancer will route traffic to the remaining healthy servers, ensuring minimal downtime.
  - **Session Stickiness** (Optional): For personalized or session-dependent URLs, session stickiness can be configured so the user's requests consistently hit the same server.
- **Tools**: Popular choices for load balancers include AWS Elastic Load Balancing (ELB), NGINX, HAProxy, or even a cloud provider's managed load balancing services.

4.3 Application Servers

The **application servers** handle incoming requests for both URL shortening and redirection. These stateless servers will process each request independently, ensuring scalability and flexibility.

- **Responsibilities**:
  - **URL Shortening**: Process POST requests to generate short URLs.
  - **Redirection**: Process GET requests to redirect users from short URLs to their corresponding long URLs.
  - **Custom Aliases & Expiration**: Handle requests for custom aliases and URL expiration logic.
- **Scalability**:
  - Multiple application servers can be deployed behind the load balancer to handle increasing traffic. Each server will handle up to a certain number of Requests Per Second (RPS) before auto-scaling is triggered.
- **Technology Stack**: This layer could be built using Python (Flask/Django), Node.js, Java (Spring Boot), or Go. These servers will interact with the database and cache to retrieve or store URL mappings.

4.4 URL Generation Service

The **URL Generation Service** is responsible for creating unique short URLs for each long URL provided by the user. It must ensure that each short URL is unique and can handle potential collisions.

- **Key Functions**:
  - **Unique ID Generation**: Generate a unique identifier using algorithms like **Base62 encoding** or **hashing (MD5/SHA-256)**.
  - **Custom Aliases**: Accept and validate user-provided aliases.
  - **Collision Handling**: Use strategies like re-hashing, appending suffixes, or checking the database for existing short URLs to handle collisions.
  - **Expiration**: Support the ability to set expiration dates for URLs. If a URL is expired, it should not redirect users and should return a proper error response.
- **Scalability**:
  - The URL generation process needs to be fast and efficient, supporting millions of requests per day.
  - In cases where short URLs are generated by incremental IDs, a **distributed ID generation system** like **Twitter's Snowflake** may be used to avoid bottlenecks.

4.5 Redirection Service

The **Redirection Service** handles the core functionality of the URL shortener: redirecting users to the original long URL when a short URL is accessed.

- **Process**:
  - Receive a request for the short URL (e.g., short.ly/abc123).
  - **Cache Lookup**: First, check the cache to see if the URL mapping is available.
  - **Database Lookup**: If not cached, query the database to retrieve the original long URL.
  - **Redirection**: Return an HTTP 301/302 response, redirecting the user to the long URL.
- **Performance**:
  - The redirection service is designed to handle large numbers of requests quickly. Using **caching** (e.g., Redis), frequently accessed URLs can be served faster.
  - Ensuring low-latency response times is critical, as users expect immediate redirection.

4.6 Database Design

The **database** stores mappings between short URLs and long URLs, along with other metadata such as creation date, expiration date, and click count.

Database Type:

- **NoSQL**: Due to the high volume of requests and the need for scalability, a NoSQL database like **DynamoDB** or **Cassandra** is preferred. These databases handle large-scale, distributed systems efficiently and provide high availability.

Schema Design:

- **URL Mapping Table**:
  - **short_url** (primary key): The short URL (7-character Base62 encoded).
  - **long_url**: The original long URL.
  - **creation_date**: Timestamp of when the URL was created.
  - **expiration_date**: Optional field specifying when the URL expires.
  - **click_count**: Integer count of how many times the URL has been clicked.
- **User Information Table** (Optional):
  - Stores custom alias preferences, account data, or usage statistics.

4.7 Cache Layer

Since the system is read-heavy (high number of redirect requests), caching plays a significant role in reducing the load on the database and improving response times.

- **In-memory Cache**: **Redis** or **Memcached** can be used to store frequently accessed URL mappings.
- **Hot URLs**: Based on the 80-20 rule, 20% of the URLs generate 80% of the traffic, so caching these hot URLs significantly improves system performance.
- **Eviction Policy**: LRU (Least Recently Used) caching policy can be used to automatically evict less frequently accessed URLs to save space.

4.8 Analytics Service (Optional)

The analytics service tracks user interactions with the shortened URLs, such as:

- **Click count**: Number of times the URL was accessed.
- **Geolocation**: Track where users are accessing the short URLs from.
- **Referrer**: Track which websites or platforms the link was shared on.

This data can be processed and displayed on user dashboards or used for business insights. The analytics service can be built using event streaming platforms like **Apache Kafka** to collect and process click events without introducing latency to the redirection service.

5. Database Design

The database design for a URL shortener needs to be highly scalable, optimized for quick lookups, and capable of storing billions of URL mappings efficiently. Given the traffic and storage needs, a NoSQL database is typically the best choice.

5.1 SQL vs NoSQL

Choosing between SQL and NoSQL depends on the scalability, consistency, and query requirements. For a URL shortener, the primary operations are simple key-value lookups, which NoSQL databases handle exceptionally well.

- **NoSQL Advantages**:
  - **Scalability**: NoSQL databases like **DynamoDB** and **Cassandra** are designed to scale horizontally, which is essential for handling millions of requests.
  - **High Availability**: Distributed architecture allows for high availability and fault tolerance.
  - **Fast Reads/Writes**: Optimized for fast reads and writes, making them suitable for high-traffic systems.
- **SQL Considerations**:
  - **Complex Queries**: SQL might be more beneficial if there are complex querying requirements, but a URL shortener typically only requires key-value lookups.
  - **ACID Transactions**: SQL databases provide strong consistency guarantees, but for a URL shortener, eventual consistency is often acceptable.

5.2 Database Schema

URL Mapping Table:

- **Primary Key**: short_url (Base62 encoded, 7 characters).
- **Attributes**:
  - long_url: The original URL.
  - creation_date: The date and time when the URL was created.
  - expiration_date (optional): The date after which the URL becomes inactive.
  - click_count: Stores the number of times the URL has been accessed.

6.System API Design

We'll design RESTful APIs that are intuitive, efficient, and scalable.
Let's break down our API design into several key endpoints:

6.1 URL Shortening API

Endpoint: POST /api/v1/shorten

This endpoint creates a new short URL for a given long URL.

Sample Request:

```
{
  "long_url": "https://example.com/very/long/url/that/needs/shortening",
  "custom_alias": "",
  "expiry_date": "2024-12-31T23:59:59Z", // optional
  "user_id": "user123"
}
```

Sample Response:

```
{
  "short_url": "http://short.url/abC123",
  "long_url": "https://example.com/very/long/url/that/needs/shortening",
  "expiry_date": "2024-12-31T23:59:59Z",
  "created_at": "2024-08-10T10:30:00Z"
}
```

6.2 URL Redirection API

Endpoint: GET /{short_url_key}

This endpoint redirects the user to the original long URL.

Sample Response:

HTTP/1.1 301 Moved Permanently
Location: https://www.example.com/some/very/long/url

7. Key Components

7.1 URL Generation Service

The **URL Generation Service** is responsible for creating a short URL that maps to the provided long URL. The service must ensure that the short URL is unique and collision-free.

- **Algorithm**:
  - Generate a unique ID using **Base62 encoding** or **hashing** methods like MD5 or SHA-256.

    - Check the database for collisions.
    - If a collision occurs, apply **re-hashing** or append a **suffix** until a unique URL is generated.
- **Custom Aliasing**: If the user provides a custom alias, the service must validate that it is unique before generating the short URL.

7.2 Redirection Service

The **Redirection Service** is responsible for looking up the long URL when a user accesses a short URL and performing the HTTP 301/302 redirection.

- **Process**:
    - Receive a GET request with the short URL key.
    - **Cache Lookup**: Check the cache for the short URL mapping.
    - **Database Lookup**: If not found in the cache, retrieve the original URL from the database.
    - Return an HTTP redirect response.

7.3 Caching for Performance

The system will use **Redis** or **Memcached** for caching frequently accessed URLs to reduce database load and improve performance.

- **Hot URLs**: The most frequently accessed URLs (20% of the total) will be cached for faster lookup.
- **Eviction Policy**: An LRU (Least Recently Used) eviction policy will be used to manage cache space.

7.4 Custom Aliasing

Users can specify custom aliases for their short URLs. The system must validate these aliases for uniqueness and allowed characters.

- **Validation**:
    - Ensure the alias is unique.
    - Validate that the alias only contains allowed characters (e.g., alphanumeric and hyphens).

7.5 Link Expiration Handling

Links can be configured to expire after a certain period. Expired links should no longer redirect users to the original URL.

- **Expiration Logic**:
    - The expiration date is stored in the database.
    - If the URL has expired, the redirection service should return an HTTP 410 (Gone) response instead of performing the redirection.

7.6 Analytics Service (Optional)

The analytics service will track the number of times each short URL is accessed and provide insights such as geolocation, device type, and referrers.

- **Data Collection**: Each redirect request will log an event that is processed asynchronously to avoid impacting redirection performance.
- **Aggregation**: Logs will be processed in batches to update the analytics database.

8. Performance Optimization

8.1 Caching Strategy

Caching is crucial in a read-heavy system like a URL shortener, where the majority of requests are for redirection. A caching strategy is implemented to reduce database load and improve response times.

- **Hot URL Caching**: The system will cache the top 20% of URLs, which account for 80% of the traffic.
- **TTL (Time to Live)**: Cached URLs will have a TTL to ensure freshness and reduce memory usage.

8.2 Database Sharding

To handle large-scale data, **sharding** will be implemented in the database to distribute the load across multiple nodes.

- **Range-Based Sharding**: Each node stores a specific range of short URLs based on their hash values or incremental IDs.
- **Hash-Based Sharding**: A hash function is applied to the short URL key to determine which shard the data should be stored in.

8.3 Load Balancing

A **load balancer** is used to distribute traffic evenly across application servers, ensuring that no single server is overwhelmed.

- **Auto-Scaling**: The system will auto-scale by adding or removing application servers based on the traffic load.

8.4 Read/Write Optimization

- **Asynchronous Writes**: In cases where write latency is critical, asynchronous writes can be used to decouple the user request from the database write.
- **Read-Heavy Optimization**: Caching, database replication, and sharding ensure that reads are optimized for speed and performance.

9. Security Considerations

Security is a crucial aspect of the URL shortener service as it can be exploited for malicious activities like phishing, spam, or attacks on users. This section outlines the necessary security measures that should be implemented.

9.1 Input Validation

Ensure that the URLs being shortened are safe and do not contain malicious content. Proper input validation will reduce the risk of storing harmful URLs in the system.

- **Sanitize Input**: Strip out any malicious characters or scripts from user-provided URLs.
- **Domain Whitelisting (Optional)**: Restrict shortening to trusted domains to prevent malicious URLs from being shortened.

9.2 Rate Limiting

To prevent abuse (e.g., a flood of URL shortening requests that could overload the system or create millions of malicious links), implement rate limiting on the API.

- **IP-based Rate Limiting**: Limit the number of API calls a single IP address can make in a given timeframe.
- **User-based Rate Limiting**: For authenticated users, implement limits to ensure fair usage.

9.3 HTTPS

All communication between the client and the server should be encrypted using **HTTPS** to protect data from eavesdropping and man-in-the-middle attacks.

- **TLS Encryption**: Use Transport Layer Security (TLS) to secure data transmission between the client, load balancer, and backend servers.

9.4 Monitoring and Alerts

Set up monitoring for unusual patterns of activity that could indicate an attack, such as:

- **DDoS Attacks**: Large spikes in traffic from a single IP address or region.
- **Link Abuse**: Detection of shortened URLs leading to phishing sites or spam.

Alert administrators immediately if such patterns are detected and implement an automatic block or rate limit for suspicious IP addresses.

10. Scalability & High Availability

To handle millions of requests efficiently, the URL shortener service must be designed with scalability and high availability in mind. This section focuses on techniques and best practices to achieve these goals.

10.1 Scaling the API Layer

The API layer can be scaled horizontally by adding more instances of application servers behind the load balancer.

- **Auto-Scaling**: Set up auto-scaling policies that add or remove application servers based on traffic volume. This ensures that the system can handle peak loads while reducing costs during low-traffic periods.
- **Load Balancing**: Use a load balancer (e.g., AWS ELB, NGINX) to distribute incoming requests evenly across application servers.

10.2 Database Replication and Sharding

To ensure the database can handle high volumes of reads and writes, implement replication and sharding.

- **Replication**: Use **database replication** to create multiple copies of the database for fault tolerance and read scalability. Write operations can go to the primary node, while read operations are distributed across replicas.
- **Sharding**: Distribute data across multiple database nodes using **sharding**. This helps in scaling the database horizontally and improves query performance.

10.3 Failover Mechanisms

Set up automated failover mechanisms to ensure that if any part of the system fails (e.g., database, application server, or load balancer), the system continues to function without downtime.

- **Active-Passive Failover**: If the primary node or server goes down, switch to a backup node/server automatically.
- **Health Checks**: Continuously monitor the health of servers, databases, and other components, and trigger failover if failures are detected.

10.4 Geo-Distributed Deployment

For global scalability and to reduce latency, deploy the system in multiple geographical regions. This ensures that users from different parts of the world are routed to the nearest data center.

- **Content Delivery Network (CDN)**: Use CDNs to cache static content and shorten the latency for users far from the server.

11. Handling Edge Cases

Edge cases and potential failures must be carefully handled to ensure system reliability and a good user experience.

11.1 Expired URLs

If a URL has expired, the system should not redirect the user to the original URL. Instead, it should return an appropriate HTTP status code.

- **HTTP 410 (Gone)**: The service should return a 410 Gone response when a user tries to access an expired URL.

11.2 Non-Existent URLs

When a user attempts to access a short URL that does not exist (e.g., mistyped or deleted), the system should handle this gracefully.

- **HTTP 404 (Not Found)**: Return a 404 Not Found response for non-existent URLs.

11.3 URL Conflicts

In cases where multiple long URLs are mapped to the same short URL (due to a hash collision or custom alias conflict), the service must detect and handle conflicts.

- **Collision Detection**: Ensure that the short URL being generated or provided (custom alias) is unique before storing it in the database.

12. Testing & Monitoring

Testing and monitoring are critical to ensure the service performs reliably, especially under heavy traffic and potential failures.

12.1 Unit and Integration Testing

- **Unit Testing**: Test individual components like the URL generation service, redirection service, and database interaction to ensure they function as expected.
- **Integration Testing**: Ensure that all components (API layer, database, cache) work together smoothly by testing the entire flow from URL shortening to redirection.

12.2 Load Testing

Load testing is essential to validate the system's performance under heavy traffic. Use tools like **Apache JMeter** or **Locust** to simulate high loads and observe system behavior.

- **Key Metrics**: Measure response time, RPS (requests per second), database query time, and overall system throughput.

12.3 Monitoring and Alerting

Set up monitoring tools like **Prometheus** or **AWS CloudWatch** to continuously track system performance.

- **Key Metrics to Monitor**:
  - **CPU & Memory Usage**: Track resource usage on application servers.
  - **API Response Time**: Measure latency for URL shortening and redirection.
  - **Cache Hit Ratio**: Monitor the percentage of requests served from the cache.
  - **Database Queries**: Track the time taken to query the database for URL lookups.

Set up **alerts** to notify administrators of performance degradation or failures (e.g., high response times, cache misses, server downtime).

13. Deployment Considerations

Reliable deployment processes are essential to ensuring that updates and fixes can be applied without disrupting the service.

13.1 Continuous Integration/Continuous Deployment (CI/CD)

- **CI/CD Pipeline**: Set up a pipeline (using **Jenkins**, **GitLab CI**, or **CircleCI**) to automate testing, building, and deploying updates to the system.
- **Automated Testing**: Integrate unit and integration tests into the pipeline to catch bugs before deployment.

13.2 Blue-Green Deployment

To ensure **zero downtime** during updates, use a **Blue-Green Deployment** strategy where two identical environments (Blue and Green) are maintained.

- **Process**:
  - Deploy the new version to the **Green** environment.
  - Run tests and validate the deployment in Green.
  - Switch traffic from **Blue** to **Green** once validation is complete.
  - If any issues arise, switch back to Blue for instant rollback.

13.3 Rollback Strategy

Have a rollback plan in place to handle failed deployments. If the new deployment introduces bugs or instability, the system should quickly revert to the previous stable version.

- **Rollback via Blue-Green Deployment**: In Blue-Green Deployment, rollback is as simple as switching traffic back to the Blue environment.
- **Database Rollback**: Ensure that database schema changes or migrations can also be safely reverted if needed.

14. Conclusion

14.1 Summary

This document outlined the design and architecture of a scalable URL shortener service capable of handling millions of requests per day. By implementing key features such as short URL generation, redirection, custom aliasing, and analytics, the service ensures a smooth user experience while maintaining high availability, low latency, and security. Key design aspects:

- A robust **NoSQL database** for storing URL mappings.
- Efficient **caching** to handle read-heavy traffic.
- Security measures like **input validation**, **rate limiting**, and **HTTPS** for safe usage.
- Scalable architecture with **load balancing**, **sharding**, and **auto-scaling** to manage high traffic.

14.2 Future Enhancements

Some potential future improvements to the system include:

- **Enhanced Analytics**: Provide more detailed analytics such as device type, referrer URLs, and real-time click tracking.
- **Multi-Tiered Caching**: Implement multi-tiered caching with edge servers or CDN integration to reduce latency further.
- **Machine Learning for URL Detection**: Integrate a machine learning model to automatically detect and block malicious URLs during input validation.

By continuously monitoring system performance and implementing these enhancements, the service can remain reliable and secure as traffic grows over time.

PART-2

Technical Documentation for Real-Time Chat Application Design

Introduction

This document outlines the technical design and architecture for a real-time chat application, featuring key functionalities like private messaging, group chat, online presence tracking, and notification services for offline users. This architecture is optimized for scalability, low latency, high availability, and efficient data storage.

1. Requirements

Functional Requirements

- **Private Messaging**: Users can exchange messages with one another in a one-to-one chat.
- **Group Messaging**: Users can join groups and participate in group conversations.
- **Online Status Tracking**: Display users' online or offline status.
- **Notifications**: Notify users when they receive messages, particularly when offline.

- **Message History**: Ensure users can retrieve their chat history upon reconnection.

Non-Functional Requirements

- **Low Latency**: Messages should be delivered in real-time.
- **High Availability**: The system should be able to handle large volumes of traffic with minimal downtime.
- **Scalability**: The architecture should support millions of daily active users and high volumes of data.
- **Data Storage**: The storage solution should be optimized for write-heavy operations.

2. High-Level Architecture

The application is composed of several key services and components:

2.1 WebSocket Server

- **Function**: Manages persistent connections between users and the server for real-time message delivery.
- **Protocol**: Uses WebSocket over TCP to enable bidirectional communication for instant message transfer.
- **Presence Management**: Tracks user connection status and relays it to other interested users (e.g., friends or group members).

2.2 HTTP Server

- **Function**: Handles non-real-time requests, such as user registration, login, joining, or leaving groups.
- **Protocol**: Uses RESTful HTTP endpoints to facilitate non-time-sensitive operations.

2.3 Chat Service

- **Function**: Manages the sending, receiving, and storing of messages.
- **Persistence**: Messages are first stored in the database before being acknowledged to ensure reliability.
- **Message Routing**: Works with the session management service to deliver messages to appropriate WebSocket servers.

2.4 Notification Service

- **Function**: Sends push notifications to offline users, alerting them to new messages.
- **Queue Management**: Queues messages intended for offline users and processes them as notifications to their devices.

2.5 Presence Service

- **Function**: Monitors the online status of users, updating their friends or group members accordingly.
- **Efficiency**: Optimizes tracking by only actively monitoring users' most frequently contacted contacts to reduce system load.

2.6 Session Management and User Mapping

- **Function**: Maps users to their current WebSocket server instances, enabling efficient message routing.
- **Data**: Stores user-session mapping and group membership data for quick access.

2.7 Database

- **Data Model**: Optimized for write-heavy operations using a NoSQL database like Cassandra.
- **Partitioning**: Ensures horizontal scalability by partitioning user data, enabling large volumes of messages to be stored and retrieved efficiently.
- **Tables**:
    - **Private Messages**: Stores direct messages between users.
    - **Group Messages**: Stores messages sent within groups, partitioned by group ID.
    - **User Membership**: Tracks user-group membership for message broadcasting.
    - **User Profile**: Holds user-specific data such as status, profile image, and login information.

3. Detailed Design

3.1 Private Messaging

- **Process**:
    1. User A sends a message to User B via the WebSocket server.
    2. Message is first stored in the database.
    3. The chat service queries the session management service to find User B's WebSocket server.
    4. Message is routed to User B’s server for delivery.
    5. If User B is offline, the message is queued for the notification service.

3.2 Group Messaging

- **Process**:
    1. User A sends a message to a group.

2. The chat service stores the message in the group messages table.
3. The group service retrieves the list of group members.
4. The message is then broadcast to each member's WebSocket server, or a notification is queued if they are offline.

3.3 Message History Retrieval

- **Process**:
  - When a user reconnects, they can retrieve undelivered messages through the get_history API.
  - This API accesses stored messages and retrieves any messages missed during the offline period.

3.4 Presence Detection

- **Mechanism**:
  - Active users send periodic heartbeats via WebSocket to the presence service.
  - The presence service maintains a list of the user's top contacts, updating them on the user's current status.
  - Other contacts may query the presence status via an HTTP request when needed.

3.5 Notification for Offline Users

- **Mechanism**:
  - When a message is undeliverable due to a user being offline, it is added to a notification queue.
  - The notification service periodically checks this queue and sends alerts to the offline user's registered device, such as through SMS or push notifications.

4. Data Flow

The following describes a typical data flow for private and group messages:

4.1 Private Message Flow

- **Step 1**: User A sends a message via WebSocket.
- **Step 2**: Message is stored and routed to User B's WebSocket server.
- **Step 3**: If User B is offline, the message is sent to the notification service for offline alerts.

4.2 Group Message Flow

- **Step 1**: User A sends a message to a group.
- **Step 2**: Group members are retrieved and checked for online status.
- **Step 3**: Online members receive the message in real time; offline members receive notifications.

5. Scalability Considerations

To handle large volumes of users and messages, the system incorporates the following strategies:

- **Load Balancing**: Distributes incoming traffic across WebSocket and HTTP servers to avoid bottlenecks.
- **Horizontal Scalability**: The database is partitioned and replicated to handle growing data volumes.
- **Queue Management**: Uses message queues to handle asynchronous tasks, such as notifications and history retrieval, reducing server load.
- **Efficient Storage**: Optimized with column-based, NoSQL storage to handle high write volumes and quick retrieval.

6. Conclusion

This real-time chat application design ensures robust, scalable communication, supporting core functionalities like private/group messaging, user status tracking, and offline notifications. The architecture prioritizes low latency and high availability, achieved through efficient use of WebSockets for real-time interactions, a scalable NoSQL database for message storage, and dedicated services for tasks such as notification handling and presence tracking.