1. Introduction

This document outlines the design, development, and deployment considerations of the URL Shortener API, created to meet both functional and operational requirements. The project addresses how to generate a short URL from a long URL, redirect to the original URL when the short version is accessed, and support the expiration of links. Additionally, this document highlights the architecture, algorithms used, system interface, and deployment considerations.

2. Functional Requirements

The project was built around the following core functionalities, as specified in the problem statement:

- **Generate a unique short URL**: The system accepts a long URL and generates a shortened version.
- **Redirect users to the original URL**: When a user accesses the short URL, they are redirected to the original URL.
- **Support link expiration**: The short URL has an expiration time, after which the link will no longer work, and accessing it will return a 404 error.

3. Operational Requirements

- **High availability**: The system should ensure that the short URLs can be accessed with minimal downtime.
- **Scalability**: The solution needs to scale as the number of URLs and requests grows. Key considerations include horizontal scaling, database partitioning, and caching.

4. High-Level Design (HLD)

4.1 System Architecture

The architecture is based on a **client-server model** with a clear separation between the backend (FastAPI) and frontend (Streamlit). Below is the architecture overview:

- **Frontend**: A user interface built with **Streamlit** that allows users to input long URLs, view all URLs, and manage them.
- **Backend (FastAPI)**: Provides API endpoints for shortening URLs, redirecting, and managing URL data.
- **Database**: An SQL-based database (SQLite for development) to store URLs and associated metadata (expiration time, creation time, etc.).
- **Caching Layer (Optional for Future Scaling)**: Using a cache like **Redis** to store frequently accessed URLs and reduce database load.

The architecture ensures modularity, ease of expansion, and scalability.

5. System Interface Definition

5.1 Create Short URL API

Generates a shortened URL from a long URL, with optional expiration date.

- **Endpoint:** POST /api/v1/shorten
- **Parameters:**
  - original_url **(string, required):** The original long URL that needs to be shortened.
  - expiration_time **(datetime, optional)**: Expiration time in years (default 1 year)
- **Request Body**:

```
{
  "original_url": "https://www.linkedin.com/in/harmanpunn/",
  "expiration_time": "2024-12-31 23:59:59.000000"
}
```

- **Response:**
  - short_url **(string):** The shortened URL generated by the service.
  - original_url **(string)**: The original URL
  - expiration_time **(datetime):** The expiration date of the shortened URL
  - short_url_key **(string):** The key of the short url (ex. qP5XqY)

5.2 Redirect API

Redirects users from a shortened URL to the original long URL.

- **Endpoint:** GET /{short_url}
- **Parameters:**
  - short_url **(string, required):** The short URL that needs to be resolved to the original URL.
- **Response:**
  - **Redirects to the** original_url

5.3 URL Management API

Retrieves a list of all URLs shortened, with metadata.

5.3.1 Get all short urls

- **Endpoint:** GET /api/v1/urls
- **Response:**
  - An array of URL objects, where each object contains the following fields:
  - short_url **(string):** The shortened URL generated by the service.
  - original_url **(string)**: The original URL
  - expiration_time **(datetime):** The expiration date of the shortened URL

5.3.2 Get metadata for a specific url

- **Endpoint:** GET /api/v1/urls/{short_url}
- **Response:**
    - short_url **(string):** The shortened URL generated by the service.
    - original_url **(string)**: The original URL
    - expiration_time **(datetime):** The expiration date of the shortened URL

5.4 Delete Short URL API

Deletes a specified shortened URL from the service.

- **Endpoint:** DELETE/api/v1/{short_url}
- **Parameters:**
    - short_url **(string, required):** The shortened URL identifier to be deleted**.**
- **Response:**
    - status **(string):** Confirmation of deletion or error message if the operation fails.

6. Database Schema and Design

Key Observations:

- **Scale**: We anticipate needing to store billions of records over time.
- **Data Size**: Each record we store is relatively small (less than 1KB).
- **No Relationships**: There are no complex relationships between records
- **Read-heavy**: Our service is primarily read-heavy, with far more read operations than writes.

The database schema consists of a single table to store URL data:

Normalization of URL: Before storing a URL, we normalize it to prevent different encodings from generating different short URLs for the same resource.

Database Choice:

Given the large scale and the absence of relationships between records, a NoSQL database such as **DynamoDB**, **Cassandra, or MongoDB** would be a suitable choice for a real-world production environment. These databases offer better scalability and performance for high-volume, read-heavy applications compared to traditional relational databases.

For this project, however, I have chosen to use **SQLite** due to its simplicity and ease of setup. While SQLite is sufficient for this task, in a real-world scenario, I would opt for a more scalable NoSQL database to accommodate the expected workload and data volume.

7. Design and Algorithm

7.1 Initial Approach (Basic Hashing)

The first version of the algorithm used **MD5 hashing** to generate a short URL by taking the MD5 hash of the original URL and truncating it to 6 characters:

```
short_url = hashlib.md5(normalized_url.encode()).hexdigest()[:6]
```

Problems with Approach 1:

1. **Collision Risk**: By truncating the MD5 hash to 6 characters, we drastically reduced the number of unique URLs that could be generated. This increases the likelihood of collisions (i.e., different URLs being mapped to the same short URL) as the system scales.
2. **Non-URL-Safe Characters**: While MD5 ensures uniqueness, truncating the hash could result in characters that are not necessarily safe for URLs, leading to potential issues when using certain characters in URLs.

7.2 Improved Approach (Base62 Encoding with Collision Handling)

The second approach improves on the initial MD5 truncation method by adding Base62 encoding and collision handling, making the system more scalable and reliable.

1. **Generate MD5 Hash of Normalized URL**
    - The MD5 hash of the normalized URL is generated, and the first 10 characters are converted into a number for Base62 encoding..

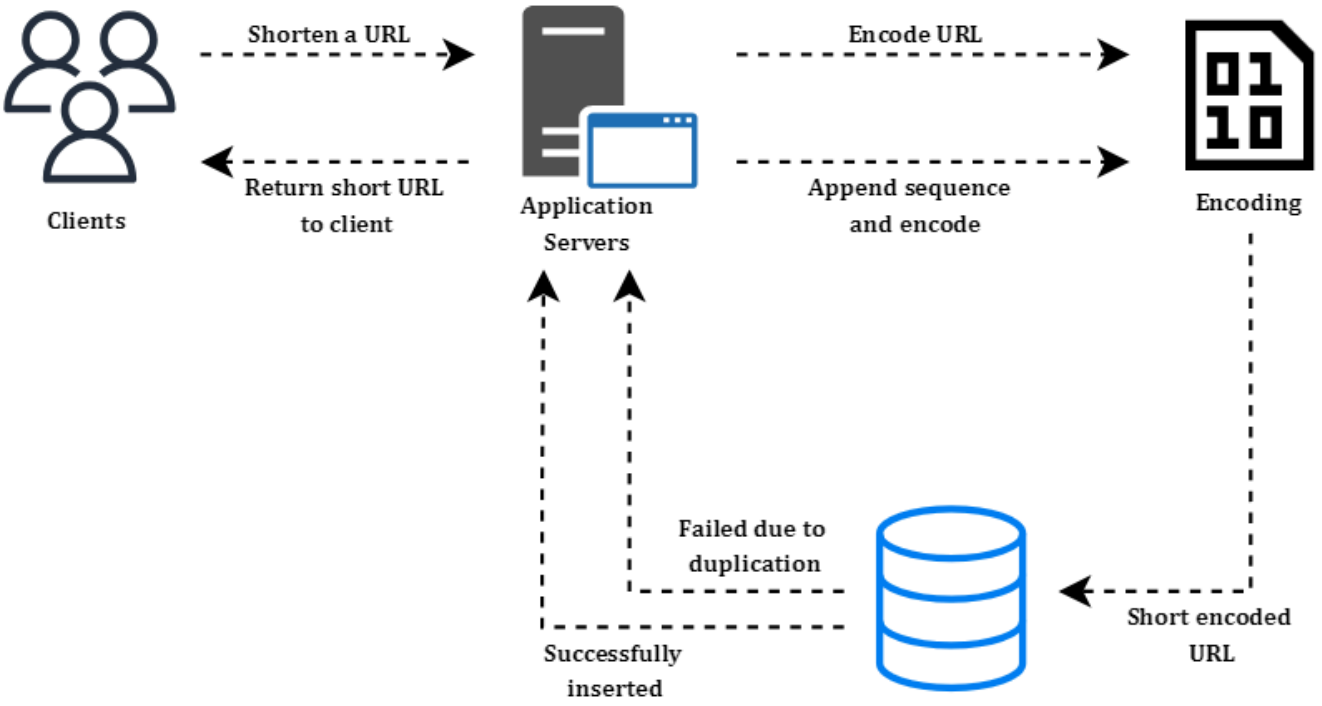
```
md5_hash = hashlib.md5(normalized_url.encode()).hexdigest()
hash_number = int(md5_hash[:10], 16)
```

2. **Base62 Encoding for Short URL Generation:**
    - The numeric value from the MD5 hash is Base62 encoded, using only URL-safe, human-readable characters ([A-Z, a-z, 0-9]).
    - The system generates a **6-character** Base62 string, which provides approximately **57 billion** possible combinations (62^6), significantly reducing the risk of collisions.

```
short_url = base62_encode(hash_number)[:6]
```

3. **Collision Handling**
    - Once a short URL is generated, the system checks for collisions in the database. If a collision is found, a new Base62 string is generated, and the process repeats until a unique URL is created.

```
while db.query(URL).filter(URL.short_url == short_url).first() is not None:
    short_url = base62_encode(random.randint(0, 62**6))
```



URL Shortening Workflow

7.3 Design Considerations

- **High Availability**: Distribute traffic using load balancing and health checks, implement auto-scaling, and deploy across multiple regions for redundancy.
- **Scalability**: Use horizontal scaling with stateless servers and database partitioning with leader-follower replication for handling increased traffic.
- **Caching**: Utilize Redis or Memcache with LRU eviction policy, updating the cache on database retrieval for improved latency and reduced load.
- **Continuous Deployment and Monitoring**: Implement CI/CD pipelines for automated testing and deployment, and monitor system health with tools like Prometheus or Datadog.
- **Security and User Management**: Enforce authentication (OAuth2, JWT), HTTPS, and rate limiting to ensure secure communication and prevent abuse.
- **Purging and Cleanup**: Use lazy deletion for expired URLs and schedule periodic cleanup services to reduce database load.

- **Data Replication and Disaster Recovery**: Ensure database replication for fault tolerance and perform regular backups for disaster recovery.

8. Deployment Considerations

Deploying a production-ready URL shortener application requires a systematic approach to ensure scalability, security, and operational efficiency.

8.1 Deployment Strategy

- **Zero-Downtime Deployment**: Implement **Blue-Green Deployment** or **Canary Deployment** to ensure that application updates can be rolled out without disrupting service. These strategies involve switching traffic between environments or incrementally releasing updates to small user subsets.
- **Script Automation**: Automate the entire deployment process using tools like **Jenkins**, **GitLab CI/CD**, or **GitHub Actions**. Use **Infrastructure as Code (IaC)** tools such as **Terraform** or **AWS CloudFormation** to provision infrastructure consistently and reproducibly.
- **Version Control & Release Management**

8.2 Monitoring and Logging

- **Performance Monitoring**: Use tools like **AWS CloudWatch** to monitor system metrics like CPU, memory, and network traffic. Application-level metrics, including **request latency** and **error rates**, should be monitored to maintain performance.
- **Centralized Logging**: Set up a centralized logging system using **Elastic Stack (ELK)**, **Splunk**, or **Papertrail** to aggregate logs from all instances. Use tools like **Datadog** for real-time error tracking and alerts.

8.3 Security Considerations

- **Authentication & Authorization**: Implement **OAuth2** or **JWT** for secure user authentication. Use **Role-Based Access Control (RBAC)** to manage access privileges.
- **SSL/TLS Encryption**: Secure all communication using **SSL certificates**. Ensure HTTPS is enforced for all user interactions with the API.
- **Vulnerability Management**: Regularly conduct **security audits** and **penetration tests**.
- **Secrets Management**: Use **AWS Secrets Manager** to store sensitive data like API keys, tokens, and credentials securely.

8.4 Rollback and Recovery

- **Rollback Plan**: Implement automated rollback mechanisms to revert to a stable version if a deployment issue arises.
- **Backup and Disaster Recovery**: Automate **database backups** using **RDS snapshots** or **AWS S3**. Ensure that backups are scheduled regularly and replicated across multiple regions to prevent data loss. S

8.5. CI/CD

- **Automated Testing**: Integrate **unit tests**, **integration tests**, and **end-to-end tests** into the CI/CD pipeline to test the application on each commit.
- **Automated Deployment**: Use CI/CD tools like **Jenkins**, or **GitHub Actions** to automate the deployment across dev, staging, and production environments.

8.6 Scalability and Load Balancing

- **Auto-Scaling**: Set up **auto-scaling** dynamically adjust the number instances based on traffic. This ensures cost-effective scaling during peak usage periods.
- **Database Scaling**: Use **read replicas** to scale read-heavy operations. Consider database **sharding** or **partitioning** to distribute write load efficiently.
- **Cache Scaling**: Scale the caching layer (e.g., **Redis**, **Memcached**) horizontally by adding more cache nodes and using **replication** for load distribution.

9. Simplifications and Future Enhancements

To meet the time constraints and simplify the project, several decisions were made regarding the technology stack and implementation details. In a real-world production scenario, these would be revised as follows:

9.1 Database Choice:
For this project, I chose **SQLite** for simplicity, as it is easy to set up and sufficient for demonstrating functionality. However, in a production environment where scalability and fault tolerance are critical, a NoSQL database such as **MongoDB** or **DynamoDB** would be more suitable due to its ability to handle large volumes of data and high read/write throughput.

9.2 Frontend Framework:
I used **Streamlit** for the frontend because it is quick to set up and allowed me to build a simple interface within the project's time frame. However, **Streamlit has a limitation in that it reruns the entire script for every user interaction**, which can make it slower and less efficient for larger applications or more interactive user interfaces. In a real-world scenario, frameworks like **React** or **Next.js** would be preferable as they offer better performance, more control over rendering, and the ability to create a highly responsive and interactive user experience.

9.3 Caching:
No caching layer was implemented for this project due to time limitations. In a production setting, caching frequently accessed URLs using **Redis** or **Memcached** would significantly improve performance by reducing the load on the database for read-heavy operations.

9.4 Purging and Cleanup:
I am also performing **lazy cleanup**, whenever a user tries to access an expired link, the system will delete the link and return an error to the user. This approach helps reduce database load by delaying the deletion process until it's necessary.

9.5 Enhancements:

Future improvements could include implementing **analytics** to track URL usage patterns, adding **user authentication** so that users can create and manage their own short URLs, and supporting **custom URLs**. Furthermore, implementing a **rate limiter** and **monitoring tools** (e.g., Prometheus, Grafana) would ensure operational stability.

Example schema to include analytics and users: