URL Shortener Application

The **URL Shortener Application** is a robust, secure, and scalable system designed to shorten URLs, provide fast redirection, and deliver insightful usage analytics. This solution is built with **Java 21** and **Spring Boot 3** and leverages modern AWS cloud-native services to ensure high availability, performance, and security.

Key highlights of the application include:

- **Core Functionality**: Shortens long URLs and redirects users seamlessly using a RESTful API.
- **Caching**: Redis (AWS ElastiCache) is integrated to speed up URL lookups and reduce latency.
- **Persistent Storage**: DynamoDB securely stores URL mappings for durability and scalability.
- **Security**: Features OAuth2 authentication (via Google), AWS WAF for rate limiting, and AWS Secrets Manager for secure credential storage.
- **Multi-Region Deployment**: Ensures high availability using DynamoDB Global Tables and AWS Lambda across regions.
- **ETL and Analytics**: AWS Glue automates metadata extraction, while AWS OpenSearch visualizes API usage and performance.
- **Automation**: A GitHub Actions CI/CD pipeline handles infrastructure provisioning (via Terraform) and deployment to AWS.

This document provides a detailed breakdown of the application's architecture, design patterns, code implementation, infrastructure, CI/CD pipeline, and a structured roadmap for development. It also includes architecture diagrams to help visualize the system flow.

Refer to the **Table of Contents** below for a step-by-step guide to the entire solution.

Table of Contents

Step 1: Solution Overview

The **URL Shortener Application** is a production-ready, secure, and scalable system built using **Java 21** and **Spring Boot 3**, leveraging modern cloud-native architecture and AWS services.

Step 2: Key Features

1. **Shorten URLs** and redirect users seamlessly.
2. **Authentication**: Secure API access with **Google OAuth2**.
3. **Caching**: Redis (AWS ElastiCache) ensures low-latency lookups.
4. **Persistence**: DynamoDB for storing URL mappings and metadata.
5. **Rate Limiting**: Prevent abuse using **AWS WAF**.
6. **Secrets Management**: AWS Secrets Manager for secure credential storage and rotation.
7. **Multi-Region Deployment**: High availability using **DynamoDB Global Tables**.
8. **Monitoring**: AWS CloudWatch for dashboards, logs, and alarms.
9. **ETL Pipelines**: AWS Glue extracts metadata and stores results in Amazon S3.
10. **Edge Caching**: AWS CloudFront accelerates content delivery globally.
11. **CI/CD Pipeline**: GitHub Actions automate infrastructure provisioning and Lambda deployments.
12. **Analytics**: AWS OpenSearch visualizes usage data.

Step 3: Architecture

High-Level Architecture

- **API Gateway**: Entry point for all API requests.
- **AWS Lambda**: Runs the Spring Boot 3 application.
- **Redis (ElastiCache)**: Low-latency caching for shortened URLs.
- **DynamoDB**: Persistent storage for URL mappings.
- **AWS Glue**: Extracts metadata and stores results in S3 buckets.
- **CloudFront**: Global caching for API responses.
- **WAF**: Implements rate limiting to prevent abuse.
- **Secrets Manager**: Manages secure credentials with rotation.
- **CloudWatch**: Logs, metrics, and monitoring dashboards.

Data Flow Diagram

1. **Client Request**: The client sends a request to the API Gateway.
2. **API Gateway**: Forwards the request to the appropriate AWS Lambda function.
3. **Redis Cache**: Checks for a cached URL mapping to ensure low-latency response.
4. **DynamoDB**: If not found in Redis, retrieves or stores the URL mappings.
5. **AWS Glue**: Processes metadata for analysis periodically.
6. **CloudFront**: Caches responses globally for reduced latency.
7. **WAF**: Protects the application against abuse and rate limits traffic.
8. **CloudWatch**: Monitors application metrics, logs, and alarms.
9. **Secrets Manager**: Manages sensitive credentials securely.

This architecture is designed for scalability, security, and high availability using AWS managed services.

Step 4: Design Patterns

The application utilizes several **design patterns** to ensure clean code architecture, scalability, and maintainability.

1. **Singleton Pattern**
   Ensures that only a single instance of resources like Redis and DynamoDB clients is created.
   - **Use Case**: RedisTemplate and DynamoDB client initialization.
2. **Builder Pattern**
   Simplifies the creation of AWS SDK requests using clean, readable code.
   - **Use Case**: Constructing DynamoDB and SecretsManager requests.
3. **Factory Pattern**
   Leverages Spring Boot's Dependency Injection (DI) to instantiate services and manage object lifecycles.
   - **Use Case**: Injecting the RedisTemplate, DynamoDB client, and service classes.

4. **Strategy Pattern**
   Allows flexible implementation of multiple strategies for fetching or storing data.
   - **Use Case**: **Cache-first strategy** – Attempt to fetch data from Redis first; fallback to DynamoDB if the cache misses.
5. **Template Method Pattern**
   Provides an abstraction for Redis operations using Spring's `RedisTemplate`.
   - **Use Case**: Simplified interactions with Redis for `get`, `set`, and expiry management.
6. **Proxy Pattern**
   The API Gateway acts as a proxy, forwarding incoming requests to the AWS Lambda function.
   - **Use Case**: Decouples clients from the internal application logic.
7. **Observer Pattern**
   AWS CloudWatch observes and monitors logs, metrics, and application performance.
   - **Use Case**: Automatic notifications and alarms based on metrics and thresholds.
8. **Decorator Pattern**
   AWS WAF adds additional functionality (rate limiting and request filtering) without modifying the core logic of the application.
   - **Use Case**: Protects APIs by adding security rules transparently.

These design patterns enhance the code's structure, flexibility, and performance while adhering to best practices.

Step 5: Code Implementation

5.1 Dependencies (`pom.xml`)

```
<dependencies>
    <!-- Spring Boot Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!-- Spring Boot Data Redis -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
    </dependency>
    <!-- AWS SDK for DynamoDB -->
    <dependency>
        <groupId>software.amazon.awssdk</groupId>
        <artifactId>dynamodb</artifactId>
```

```
    </dependency>
    <!-- AWS SDK for Secrets Manager -->
    <dependency>
        <groupId>software.amazon.awssdk</groupId>
        <artifactId>secretsmanager</artifactId>
    </dependency>
    <!-- Spring Boot OAuth2 Client -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-oauth2-client</artifactId>
    </dependency>
    <!-- Jackson for JSON handling -->
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
    </dependency>
</dependencies>
```

5.2 Configuration (`application.yml`)

```
app:
  base-url: https://enok.tech/shorten-url

aws:
  dynamodb:
    table-name: URLMappings
  secrets:
    name: url-shortener/secrets
  region: us-east-1

spring:
  redis:
    host: localhost
    port: 6379
    timeout: 2000ms
```

5.3 Secrets Management

```
package com.example.shortener.service;

import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.stereotype.Service;
import software.amazon.awssdk.services.secretsmanager.SecretsManagerClient;
import software.amazon.awssdk.services.secretsmanager.model.GetSecretValueRequest;

import java.util.Map;

@Service
public class SecretsService {
```

```
    private final SecretsManagerClient secretsClient;

    public SecretsService() {
        this.secretsClient = SecretsManagerClient.builder().build();
    }

    public Map<String, String> getSecrets(String secretName) {
        GetSecretValueRequest request = GetSecretValueRequest.builder()
                .secretId(secretName)
                .build();
        String secretString =
secretsClient.getSecretValue(request).secretString();
        try {
            return new ObjectMapper().readValue(secretString, Map.class);
        } catch (Exception e) {
            throw new RuntimeException("Error parsing secrets from AWS Secrets
Manager", e);
        }
    }
}
```

5.4 URL Shortener Service

```
package com.example.shortener.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.stereotype.Service;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.*;

import java.util.Map;
import java.util.UUID;
import java.util.concurrent.CompletableFuture;

@Service
public class UrlShortenerService {

    @Value("${app.base-url}")
    private String baseUrl;

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    @Autowired
    private DynamoDbClient dynamoDbClient;

    public String shortenUrl(String longUrl) {
        String shortUrl = UUID.randomUUID().toString().substring(0, 8);
        CompletableFuture.runAsync(() -> storeMapping(shortUrl, longUrl));
```

```
        return String.format("%s/%s", baseUrl, shortUrl);
    }

    public String getLongUrl(String shortUrl) {
        String cachedUrl = redisTemplate.opsForValue().get(shortUrl);
        return cachedUrl != null ? cachedUrl : fetchFromDynamoDB(shortUrl);
    }

    private void storeMapping(String shortUrl, String longUrl) {
        redisTemplate.opsForValue().set(shortUrl, longUrl);
        dynamoDbClient.putItem(PutItemRequest.builder()
                .tableName("URLMappings")
                .item(Map.of(
                        "shortUrl", AttributeValue.builder().s(shortUrl).build(),
                        "longUrl", AttributeValue.builder().s(longUrl).build()))
                .build());
    }

    private String fetchFromDynamoDB(String shortUrl) {
        GetItemResponse response = dynamoDbClient.getItem(GetItemRequest.builder()
                .tableName("URLMappings")
                .key(Map.of("shortUrl",
AttributeValue.builder().s(shortUrl).build()))
                .build());
        return response.hasItem() ? response.item().get("longUrl").s() : null;
    }
}
```

5.5 Redis Configuration

```
package com.example.shortener.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisStandaloneConfiguration;
import org.springframework.data.redis.connection.lettuce.LettuceConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;

@Configuration
public class RedisConfig {

    @Bean
    public LettuceConnectionFactory redisConnectionFactory() {
        RedisStandaloneConfiguration config = new
RedisStandaloneConfiguration("localhost", 6379);
        return new LettuceConnectionFactory(config);
    }

    @Bean
    public RedisTemplate<String, String> redisTemplate() {
        RedisTemplate<String, String> template = new RedisTemplate<>();
```

```
        template.setConnectionFactory(redisConnectionFactory());
        return template;
    }
}
```

Step 6: Infrastructure as Code (Terraform)

6.1 DynamoDB Table

```
resource "aws_dynamodb_table" "url_mapping" {
  name         = "URLMappings"
  billing_mode = "PAY_PER_REQUEST"
  hash_key     = "shortUrl"

  attribute {
    name = "shortUrl"
    type = "S"
  }

  server_side_encryption {
    enabled = true
  }

  tags = {
    Name        = "URLMappingsTable"
    Environment = "Production"
  }
}
```

6.2 AWS Lambda Function

```
resource "aws_lambda_function" "url_shortener_lambda" {
  function_name = "url-shortener-lambda"
  runtime       = "java21"
  handler       = "com.example.shortener.LambdaHandler::handleRequest"
  role          = aws_iam_role.lambda_exec_role.arn

  s3_bucket = "your-s3-bucket-name"
  s3_key    = "lambda/url-shortener.jar"

  environment {
    variables = {
      DYNAMODB_TABLE_NAME = aws_dynamodb_table.url_mapping.name
      APP_BASE_URL        = "https://enok.tech/shorten-url"
    }
  }
```

```
  tags = {
    Name        = "UrlShortenerLambda"
    Environment = "Production"
  }
}
```

6.3 API Gateway

```
resource "aws_apigatewayv2_api" "api_gateway" {
  name          = "url-shortener-api"
  protocol_type = "HTTP"
}

resource "aws_apigatewayv2_stage" "api_stage" {
  api_id      = aws_apigatewayv2_api.api_gateway.id
  name        = "prod"
  auto_deploy = true
}

resource "aws_apigatewayv2_integration" "lambda_integration" {
  api_id           = aws_apigatewayv2_api.api_gateway.id
  integration_type = "AWS_PROXY"
  integration_uri  = aws_lambda_function.url_shortener_lambda.invoke_arn
}
```

6.4 Redis (ElastiCache)

```
resource "aws_elasticache_cluster" "redis_cluster" {
  cluster_id           = "url-shortener-redis"
  engine               = "redis"
  node_type            = "cache.t2.micro"
  num_cache_nodes      = 1
  port                 = 6379
  parameter_group_name = "default.redis6.x"

  tags = {
    Name        = "UrlShortenerRedis"
    Environment = "Production"
  }
}
```

6.5 WAF (Rate Limiting)

```
resource "aws_wafv2_web_acl" "waf_rate_limit" {
  name  = "url-shortener-waf"
  scope = "REGIONAL"

  default_action {
    allow {}
  }

  rule {
    name     = "RateLimitRule"
    priority = 1

    action {
      block {}
    }

    statement {
      rate_based_statement {
        limit              = 2000
        aggregate_key_type = "IP"
      }
    }

    visibility_config {
      cloudwatch_metrics_enabled = true
      metric_name                = "RateLimitRuleMetrics"
      sampled_requests_enabled   = true
    }
  }

  tags = {
    Name        = "UrlShortenerWAF"
    Environment = "Production"
  }
}
```

6.6 Secrets Manager

```
resource "aws_secretsmanager_secret" "url_shortener_secrets" {
  name = "url-shortener-secrets"

  tags = {
    Name        = "UrlShortenerSecrets"
    Environment = "Production"
  }
}

resource "aws_secretsmanager_secret_version" "url_shortener_secrets_value" {
  secret_id     = aws_secretsmanager_secret.url_shortener_secrets.id
  secret_string = jsonencode({
```

```
    "dbUsername" = "your-username",
    "dbPassword" = "your-password"
  })
}
```

6.7 CloudFront

```
resource "aws_cloudfront_distribution" "cdn" {
  enabled = true

  origin {
    domain_name = aws_apigatewayv2_api.api_gateway.api_endpoint
    origin_id   = "apiGatewayOrigin"
  }

  default_cache_behavior {
    target_origin_id       = "apiGatewayOrigin"
    viewer_protocol_policy = "redirect-to-https"
    allowed_methods        = ["GET", "POST"]
    cached_methods         = ["GET"]
    compress               = true
  }

  viewer_certificate {
    cloudfront_default_certificate = true
  }

  tags = {
    Name        = "UrlShortenerCDN"
    Environment = "Production"
  }
}
```

6.8 Outputs

```
output "api_gateway_endpoint" {
  description = "API Gateway endpoint"
  value       = aws_apigatewayv2_api.api_gateway.api_endpoint
}

output "lambda_function_arn" {
  description = "Lambda function ARN"
  value       = aws_lambda_function.url_shortener_lambda.arn
}

output "redis_endpoint" {
  description = "Redis endpoint"
  value       = aws_elasticache_cluster.redis_cluster.primary_endpoint_address
}
```

```
output "cloudfront_url" {
  description = "CloudFront distribution URL"
  value       = aws_cloudfront_distribution.cdn.domain_name
}
```

6.9 Terraform File Structure

```
terraform/
│
├── main.tf                 # Core Terraform resources for Lambda, DynamoDB, API
Gateway
├── redis.tf                # Redis ElastiCache configuration
├── waf.tf                  # AWS WAF configuration for rate limiting
├── secrets_manager.tf      # AWS Secrets Manager configuration
├── cloudfront.tf           # CloudFront for edge caching
├── outputs.tf              # Outputs of resource endpoints
├── variables.tf            # Configuration variables for reusability
├── provider.tf             # AWS provider and Terraform backend configuration
└── backend.tf              # (Optional) Remote backend configuration
```

Step 7: CI/CD Pipeline (GitHub Actions)

To automate infrastructure provisioning, deployment of the Lambda function, and API Gateway integration, we use **GitHub Actions** as the CI/CD pipeline.

7.1 GitHub Actions Workflow File

Create a `.github/workflows/deploy.yml` file in your project repository:

```
name: Deploy to AWS

on:
  push:
    branches: [main]

jobs:
  deploy:
    name: Deploy Terraform and Lambda
    runs-on: ubuntu-latest

    steps:
      # Step 1: Checkout Code
      - name: Checkout Repository
        uses: actions/checkout@v3
```

```
      # Step 2: Set up AWS CLI
      - name: Configure AWS CLI
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: us-east-1

      # Step 3: Set up Terraform
      - name: Set up Terraform
        uses: hashicorp/setup-terraform@v2

      - name: Terraform Init
        run: terraform init
        working-directory: ./terraform

      - name: Terraform Apply
        run: terraform apply -auto-approve
        working-directory: ./terraform

      # Step 4: Package and Deploy Lambda
      - name: Package Lambda Function
        run: |
          mkdir -p target
          mvn clean package
          cp target/url-shortener.jar ./terraform/url-shortener.jar

      - name: Upload Lambda to S3
        run: |
          aws s3 cp ./terraform/url-shortener.jar s3://your-s3-bucket/lambda/url-
shortener.jar

      - name: Update Lambda Function
        run: |
          aws lambda update-function-code \
            --function-name url-shortener-lambda \
            --s3-bucket your-s3-bucket \
            --s3-key lambda/url-shortener.jar

      # Step 5: Verify Deployment
      - name: Verify API Deployment
        run: |
          echo "Deployment Successful. Access API via: https://your-api-gateway-
endpoint"
```

7.2 CI/CD Pipeline Steps Explained

1. **Trigger**:
    - Pipeline runs on any **push to the `main` branch**.
2. **Checkout Code**:

- Pulls the latest code from the GitHub repository.

3. **Configure AWS CLI**:
   - Sets up AWS credentials using GitHub Secrets (`AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`).
4. **Terraform Deployment**:
   - Initializes (`terraform init`) and applies the infrastructure (`terraform apply`).
   - Deploys resources like **DynamoDB, Redis, Lambda, WAF, and API Gateway**.
5. **Package Lambda Function**:
   - Compiles the Spring Boot project into a **JAR** file using Maven.
6. **Upload to S3**:
   - Uploads the packaged Lambda JAR file to the specified **S3 bucket**.
7. **Update AWS Lambda Function**:
   - Updates the Lambda function code to use the newly uploaded JAR file.
8. **Verify Deployment**:
   - Outputs the API Gateway endpoint for accessing the deployed API.

7.3 Directory Structure for GitHub Actions

```
.github/
└── workflows/
    └── deploy.yml    # CI/CD pipeline configuration
```

7.4 Required GitHub Secrets

Ensure the following secrets are configured in your GitHub repository for the CI/CD pipeline to work securely:

1. **AWS_ACCESS_KEY_ID**
   - Description: AWS Access Key ID for authenticating with AWS services.
   - Example Value: `AKIAIOSFODNN7EXAMPLE`
2. **AWS_SECRET_ACCESS_KEY**
   - Description: AWS Secret Access Key for authenticating with AWS services.
   - Example Value: `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`
3. **AWS_REGION**
   - Description: AWS region where resources will be deployed.
   - Example Value: `us-east-1`

4. **S3_BUCKET_NAME**
    - Description: Name of the S3 bucket where the Lambda JAR file will be uploaded.
    - Example Value: `your-s3-bucket-name`
5. **LAMBDA_FUNCTION_NAME**
    - Description: Name of the Lambda function to be updated with the new JAR file.
    - Example Value: `url-shortener-lambda`
6. **TERRAFORM_BACKEND_BUCKET** *(Optional)*
    - Description: S3 bucket used as the backend for Terraform state files.
    - Example Value: `terraform-backend-bucket`

7.5 Outputs and Monitoring

1. **Logs**
    - All deployment logs and pipeline activity are visible in the **GitHub Actions** workflow under the "Actions" tab of your repository.
    - Logs include Terraform execution details, Lambda function updates, and S3 upload confirmations.
2. **Verification**
    - The pipeline outputs the **API Gateway endpoint** to validate that the application has been deployed successfully.
    - Example Output:

```
Deployment Successful. Access API via: https://<api-gateway-endpoint>
```

3. **Monitoring**
    - **AWS CloudWatch** automatically captures logs, metrics, and events for the Lambda function.
    - Monitor key metrics such as:
        - **Invocation Count**: Total number of Lambda function executions.
        - **Execution Duration**: Time taken to execute the Lambda function.
        - **Error Rate**: Number of failed invocations.
        - **Throttling**: Count of throttled requests due to rate limits.
4. **Notifications**
    - Configure GitHub Actions to notify you via email or Slack when the pipeline succeeds or fails.
    - Add integration with services like **AWS SNS** or third-party tools like **PagerDuty** for alerts.
5. **Debugging**
    - If errors occur during deployment or function execution, CloudWatch logs can be analyzed for detailed stack traces and error messages.

- Terraform outputs will include details of any resource creation failures.

7.6 Benefits of CI/CD Pipeline

1. **Automation**
   - Eliminates manual steps for provisioning infrastructure and deploying code.
   - Ensures consistency across environments (e.g., development, staging, production).
2. **Fast Feedback**
   - Automatically deploys changes upon a push to the `main` branch.
   - Provides immediate feedback about build or deployment failures.
3. **Infrastructure as Code**
   - Terraform manages all AWS resources in a version-controlled manner.
   - Enables repeatable, consistent, and auditable deployments.
4. **Reliability**
   - Ensures the latest code and infrastructure updates are deployed seamlessly.
   - Automatically rolls out changes without downtime when properly configured.
5. **Visibility**
   - All deployment logs are captured in GitHub Actions workflows.
   - Provides insights into each stage of the pipeline, including resource creation, Lambda updates, and S3 uploads.
6. **Security**
   - Secrets like `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` are securely stored using GitHub Secrets.
   - Reduces risk by avoiding manual handling of credentials.
7. **Scalability**
   - Supports frequent changes, allowing development teams to scale the project efficiently.
   - CloudWatch logs and monitoring ensure application performance at scale.
8. **Notifications and Alerts**
   - Integrate with tools like Slack or AWS SNS to receive real-time alerts about deployment successes, failures, or errors.

By leveraging GitHub Actions, this CI/CD pipeline ensures **automation, consistency, and reliability** while reducing deployment overhead and enabling rapid iteration.

Step 8: Roadmap

The roadmap outlines the phases for building, deploying, and enhancing the **URL Shortener Application**. Each phase focuses on incremental improvements to achieve a secure, scalable, and production-ready system.

Phase 1: Core API Development

- Build a REST API to shorten URLs and redirect users.
- Implement **DynamoDB** for persistent URL storage.
- Add **Redis** for caching frequently accessed URLs.
- Expose endpoints:
  - `POST /shorten` → Accepts long URLs and returns shortened URLs.
  - `GET /{shortUrl}` → Redirects to the original URL.

Phase 2: Security Enhancements

- Add **Google OAuth2** for user authentication.
- Protect endpoints to allow only authenticated access.
- Integrate **AWS Secrets Manager** for secure credential management.

Phase 3: Infrastructure Deployment

- Provision AWS resources using **Terraform**:
  - API Gateway
  - Lambda
  - DynamoDB
  - Redis (ElastiCache)
  - CloudFront
  - WAF
  - Secrets Manager
- Manage Terraform state with an S3 backend for consistency.

Phase 4: CI/CD Pipeline

- Implement a **GitHub Actions** pipeline:
  - Automate infrastructure provisioning using Terraform.
  - Package and deploy the Lambda function to AWS.
  - Upload the application JAR file to S3.
- Verify deployments and monitor pipeline execution.

Phase 5: Rate Limiting and Security

- Integrate **AWS WAF** to protect APIs from abuse:
  - Apply rate-based rules to limit requests per IP.
- Use **IAM roles** for least-privilege access to AWS resources.
- Enable encryption for DynamoDB, Redis, and S3.

Phase 6: Multi-Region High Availability

- Deploy **DynamoDB Global Tables** to replicate data across regions.
- Deploy Lambda functions and API Gateway endpoints in multiple regions.
- Use **Route 53** to route traffic to the closest regional endpoint.

Phase 7: Monitoring and Observability

- Integrate **AWS CloudWatch** for monitoring:
  - Lambda function logs, errors, and metrics.
  - API Gateway request logs.
- Set up CloudWatch alarms for critical issues like throttling, high latency, or errors.
- Create **CloudWatch dashboards** to visualize application health.

Phase 8: ETL Pipelines and Analytics

- Use **AWS Glue** to periodically extract metadata (headers, content) from shortened URLs.
- Store processed data in **Amazon S3** for long-term storage.
- Integrate **AWS OpenSearch** to analyze and visualize usage statistics:
  - Number of shortened URLs.
  - URL access patterns.
  - Performance metrics.

Phase 9: Optimization and Scalability

- Optimize the Lambda function for performance and cost efficiency.
- Enable **CloudFront** to cache API responses globally and reduce latency.
- Use **Auto Scaling** for ElastiCache to handle increased traffic.
- Conduct load testing to ensure scalability under heavy loads.

Roadmap Benefits:

- Incremental, phased development for easier implementation.
- Ensures the application evolves to be **secure**, **scalable**, and **highly available**.
- Provides clear milestones for development, testing, and deployment.
- Enables future extensions like detailed analytics, real-time monitoring, and advanced caching.

By following this roadmap, the **URL Shortener Application** can progress from a basic proof of concept to a robust, enterprise-grade solution.

Step 9: Architecture Diagrams (Application and Infrastructure)

To visually represent the architecture, you can use a tool like **draw.io** to create the following diagrams:

9.1 Application Architecture Diagram

Components:

1. **Client**
    - Represents users interacting with the system via HTTP requests.
2. **API Gateway**
    - Acts as the entry point for all incoming requests.
3. **AWS Lambda**
    - Processes requests:
        - `POST /shorten` → Generates and stores shortened URLs.
        - `GET /{shortUrl}` → Fetches and redirects to the original URL.
4. **Redis (ElastiCache)**
    - Caches frequently accessed URLs for faster lookups.
5. **DynamoDB**
    - Serves as the persistent database for storing shortened and original URLs.
6. **Secrets Manager**
    - Stores and retrieves secure credentials (e.g., DB credentials).
7. **AWS WAF**
    - Protects the API Gateway by applying rate limiting and security rules.
8. **CloudFront**
    - Globally caches API Gateway responses for faster delivery to users.

Data Flow:

1. **Client** → Sends HTTP requests to **API Gateway**.
2. **API Gateway** → Proxies requests to **AWS Lambda**.
3. **AWS Lambda**:
    - Checks **Redis** for cached data.
    - If not found, queries **DynamoDB**.
    - Returns the response to the client and caches the result in **Redis**.
4. **Secrets Manager** → Provides secure credentials to AWS Lambda.
5. **AWS WAF** → Secures API Gateway with rate limits and IP filtering.
6. **CloudFront** → Caches API responses globally to reduce latency.

9.2 Infrastructure Diagram

Components:

1. **GitHub Actions**
    - CI/CD pipeline for deployment automation.
2. **Terraform**
    - Infrastructure as code to provision resources.
3. **AWS Resources**:
    - **API Gateway** → Manages API endpoints.
    - **Lambda** → Runs the serverless application.
    - **DynamoDB** → Persistent database for storing URL mappings.
    - **Redis (ElastiCache)** → Provides caching for low-latency lookups.
    - **CloudFront** → Delivers content globally with edge caching.
    - **WAF** → Adds rate limiting and security rules.
    - **Secrets Manager** → Stores secure credentials.
    - **S3** → Stores the Lambda deployment package.
4. **CloudWatch**
    - Monitors Lambda function logs, metrics, and alarms.
5. **AWS Glue**
    - Periodically processes data (e.g., metadata extraction).
6. **OpenSearch**
    - Visualizes analytics and usage statistics.

Connections and Flow:

1. **GitHub Actions** → Deploys infrastructure using Terraform and updates Lambda function code.
2. **Terraform** → Provisions all AWS resources:
    - API Gateway
    - AWS Lambda
    - DynamoDB
    - Redis (ElastiCache)
    - CloudFront
    - WAF
    - Secrets Manager
    - S3
    - Glue
3. **Client** → Sends requests to **CloudFront**, which forwards them to **API Gateway**.

4. **API Gateway** → Routes requests to **AWS Lambda**.
5. **AWS Lambda**:
    - Queries **Redis** for cached URLs.
    - Falls back to **DynamoDB** for persistent storage if data is not cached.
    - Updates cache and returns the response.
6. **Secrets Manager** → Provides secure credentials to **AWS Lambda**.
7. **CloudWatch** → Captures logs, metrics, and errors for Lambda and API Gateway.
8. **AWS Glue** → Extracts metadata periodically and stores processed data in **S3**.
9. **OpenSearch** → Visualizes processed metadata and API usage statistics.

9.3 Suggested Diagram Layout in draw.io

Application Architecture Diagram Layout:

- **Left Side**:
    - Client → API Gateway → AWS WAF (Layer in front of API Gateway).
- **Middle**:
    - API Gateway → AWS Lambda.
- **Right Side**:
    - AWS Lambda → Redis (top) → DynamoDB (bottom).
- **Bottom**:
    - AWS Secrets Manager → AWS Lambda.
- **Top**:
    - CloudFront (connecting to API Gateway).

Infrastructure Diagram Layout:

- **Top**:
    - GitHub Actions → Terraform.
- **Middle**:
    - Terraform → AWS Resources:
        - API Gateway
        - AWS Lambda
        - Redis (ElastiCache)

    - DynamoDB
    - CloudFront
    - WAF
    - Secrets Manager
    - S3
    - AWS Glue.

- **Bottom**:
    - AWS Glue → S3 → OpenSearch (analytics).
- **Side**:
    - CloudWatch → AWS Lambda, API Gateway (for monitoring).

9.4 Tools to Create the Diagrams

1. **draw.io (Diagrams.net)**:
    - Free and widely used tool to design architecture diagrams.
    - Website: https://app.diagrams.net/
2. **Lucidchart**:
    - A more advanced paid tool with templates for AWS architecture.
3. **AWS Architecture Icons**:
    - Official AWS icons for designing diagrams: https://aws.amazon.com/architecture/icons/

By following these guidelines, you can create clear and professional diagrams for both the **Application Architecture** and **Infrastructure**.

Let me know if you'd like further details or adjustments! 🚀

Step 10: Conclusion

The **URL Shortener Application** delivers a secure, scalable, and production-ready solution leveraging modern **Java 21**, **Spring Boot 3**, and AWS cloud services. The design incorporates industry best practices, cloud-native tools, and clean architecture principles to ensure high performance and extensibility.

Key Achievements:

1. **Core Functionality**
    - RESTful API for shortening and retrieving URLs.

- **DynamoDB** for persistent storage and **Redis** for caching to optimize response times.

2. **Security**
    - **Google OAuth2** for user authentication.
    - **AWS WAF** for API protection and rate limiting.
    - **AWS Secrets Manager** for secure credential management with automatic rotation.
3. **Infrastructure as Code**
    - Full infrastructure provisioning using **Terraform**:
        - API Gateway
        - Lambda
        - DynamoDB
        - Redis
        - CloudFront
        - WAF
        - Secrets Manager
4. **CI/CD Automation**
    - Implemented a robust **GitHub Actions pipeline** for automating infrastructure deployment and Lambda code updates.
5. **High Availability and Scalability**
    - Multi-region deployments using **DynamoDB Global Tables** and AWS Lambda.
    - Global edge caching with **AWS CloudFront**.
6. **Monitoring and Observability**
    - **CloudWatch** for logging, monitoring, and alarms.
    - Centralized dashboards for real-time application performance tracking.
7. **ETL Pipelines and Analytics**
    - Automated data processing with **AWS Glue**.
    - Usage analytics and reporting using **AWS OpenSearch**.

Final Architecture Recap:

The final architecture combines the following AWS services:

- **API Gateway** → Routes HTTP requests.
- **AWS Lambda** → Handles business logic with a serverless approach.
- **DynamoDB** → Stores URL mappings securely.
- **Redis (ElastiCache)** → Caches URL lookups for faster responses.
- **AWS WAF** → Adds rate limiting and API security.
- **Secrets Manager** → Manages sensitive credentials securely.
- **CloudFront** → Globally caches API responses for improved latency.
- **AWS Glue** → Extracts metadata for data analytics.

- **AWS OpenSearch** → Provides visualized usage insights.
- **CloudWatch** → Monitors logs, errors, and metrics.

Future Enhancements:

1. **Custom User Dashboards**
    - Allow users to view statistics and manage their shortened URLs.
2. **Real-Time Analytics**
    - Use AWS Kinesis or DynamoDB Streams to process URL access events in real-time.
3. **Advanced Monitoring**
    - Integrate AWS X-Ray for detailed tracing and performance insights.
4. **Enhanced ETL Pipelines**
    - Automate further metadata extraction and storage for reporting.
5. **Scalability Testing**
    - Conduct load testing to identify and optimize bottlenecks under heavy loads.

Final Thoughts:

By following this design and roadmap, the **URL Shortener Application** is optimized for:

- **High Performance**: Fast URL lookups with caching and serverless execution.
- **Security**: End-to-end security with OAuth2, WAF, and Secrets Manager.
- **Scalability**: Elastic AWS services handle increasing traffic seamlessly.
- **Maintainability**: CI/CD pipelines and infrastructure as code ensure smooth operations.

This solution can be extended to meet evolving requirements, such as analytics, custom user features, and enhanced security. By leveraging the full power of AWS, the application is ready to scale and serve global users efficiently.

Congratulations on reaching the conclusion! 🚀
Let me know if you'd like to dive deeper into any aspect of the solution or need additional guidance.