DevOps URL Shortener & Monitoring System

Comprehensive Project Documentation

Submitted by:
Thomas Eid Gerges Sureal
Moaaz Zaki Ismail Zaki
Nourhan Khalid Ahmed Mahmoud
Sherif Mostafa Abdelaziz Mohamed
Mariam Ahmed Soude ElSayed

Under the supervision of:
Eng. Ahmed Gamiel

Digital Egypt Pioneers Initiative (DEPI) – DevOps Track

Project Duration: September 21, 2025 – November 20, 2025

Table of Contents

1. Project Planning & Management

This section outlines the project proposal, objectives, scope, and overall planning for the DevOps URL Shortener & Monitoring System.

1.1 Project Proposal

The DevOps URL Shortener project provides a web-based service to shorten URLs and monitor system performance. It includes observability using Prometheus and Grafana, enabling real-time visibility into service health and performance.

1.2 Objectives

- Develop a scalable and lightweight URL shortener API.
- Integrate Prometheus for performance metrics collection.
- Visualize system health and usage using Grafana.
- Ensure reliability with Docker containerization and persistence.
- Implement secure signup and authentication using MariaDB.

1.3 Scope

The system covers full development, containerization, monitoring, and persistence configuration. Enhancements include secure user authentication through MariaDB and better metrics integration.

1.4 Task Assignment & Responsibilities

1. Thomas Eid Gerges Surreal

Responsibilities:

- Designed user interface for the URL shortening and redirect pages.
- Led the backend development of the URL shortener service.
- Designed and implemented REST API endpoints (POST /shorten, GET /<id>).
- Managed infrastructure setup (EC2 instance, security groups, firewall rules).
- Ensured completeness of reports and project submission materials.
- Maintained environment variables and secrets securely.

---

2. Moaaz Zaki Ismail Zaki

Responsibilities:

- Designed Grafana dashboards and alert rules.
- Improved UX with input validations and error handling.
- Integrated Prometheus metrics into the application.
- Worked on Grafana dashboard design, layout, and data panels.
- Created alerting and visualization rules.
- Oversaw Kubernetes manifests and deployment YAMLs.

- Assisted in preparing final project presentation.

3. Nourhan Khalid Ahmed Mahmoud

Responsibilities:

- Built CI/CD pipeline using Jenkins (Jenkinsfile).
- Automated deployment to EC2 using Docker Compose and Kubernetes manifests.
- Documented monitoring metrics and dashboard usage.
- Built Jenkins pipeline for automated Docker build, test, and push to DockerHub.
- Configured Jenkins on EC2 and integrated GitHub Webhooks.
- Maintained DockerHub repositories and versioning.

---

4. Sherif Mostafa Abdelaziz Mohamed

Responsibilities:

- Created project documentation according to DEPI guidelines.
- Created ansible playbook to setup needed installations on EC2 instance
- Created ER Diagram and normalization of tables.
- Implemented MariaDB connection for signup/login functionality.
- Designed the database schema (SQLite initially, MariaDB for the Signup expansion).
- Wrote SQL for user authentication and signup.
- Responsible for diagrams:
  - Use Case Diagram
  - DFD (0, 1 levels)
  - Class Diagram
  - Sequence Diagram
  - Activity Diagram
  - State Diagram

---

5. Mariam Ahmed Soude ElSayed

Responsibilities:

- Implemented migrations and schema updates.
- Created Dockerfile and Docker Compose configuration for multi-service deployment.
- Managed data persistence volumes in Docker Compose.
- Assisted in Prometheus configuration tuning.
- Contributed to monitoring and alerting configuration.

- Contributed to API data validation and sanitization.

Team Member	Summary of Responsibilities
Thomas Eid Gerges Surreal	Led backend development, built REST API, designed UI pages, managed server infrastructure on EC2, and ensured final documentation completeness.
Moaaz Zaki Ismail Zaki	Developed Grafana dashboards, integrated Prometheus metrics, created alerting rules, enhanced UX, and managed Kubernetes deployment files.
Nourhan Khalid Ahmed Mahmoud	Built Jenkins CI/CD pipeline, automated deployment using Docker & Kubernetes, configured Jenkins + GitHub Webhooks, and maintained DockerHub repositories.
Sherif Mostafa Abdelaziz Mohamed	Created full project documentation, built Ansible automation, designed ERD and database schema, implemented MariaDB authentication, and created all system diagrams.
Mariam Ahmed Soude ElSayed	Developed database migrations, built Docker & Compose setup, managed persistent storage, assisted in Prometheus configuration, and improved API validation.

1.5 Risk Assessment & Mitigation Plan – Identifying Risks and Solutions

Risk	Impact	Likelihood	Mitigation Plan
Pipeline Failure due to Incorrect Configuration	High	Medium	Implement automated linting and validation for Jenkinsfile & Kubernetes YAML; enforce peer review before merging.
Docker Image Build Failures	Medium	Medium	Use version-pinned dependencies; add build-stage testing; maintain clean Dockerfile with multi-stage builds.
Secrets Exposure in CI/CD	High	Low	Use Jenkins Credentials Manager & Kubernetes Secrets; restrict access; rotate credentials periodically.
Application Downtime During Deployment	High	Medium	Use rolling updates in Kubernetes; enable readiness/liveness probes; implement pre-deployment smoke tests.
Insufficient Resources on Kubernetes Cluster	Medium	Medium	Implement resource requests & limits; enable HPA (Horizontal Pod Autoscaler); monitor cluster metrics.
DockerHub Rate Limit or Image Pull Failures	Medium	Medium	Enable image caching; consider using a private registry; authenticate DockerHub pulls.
Network/Service Connectivity Issues	High	Medium	Add health checks; use retry logic; implement service mesh or monitoring to detect failures early.
Security Vulnerabilities in	High	Medium	Run vulnerability scans (Trivy, Snyk); update base images regularly; follow CVE patch cycles.

Application or Base Image			
Team Miscommunication or Role Overlaps	Medium	Low	Maintain updated task assignment sheet; conduct weekly sync meetings; use dedicated communication channels.

1.6 KPIs (Key Performance Indicators) – Metrics for Project Success

Category	**KPI**	**Target**	**Description**
System Performance	**Application Response Time**	< 200 ms	Measures how fast the application responds to user/API requests.
	Deployment Time	< 3 minutes	Time taken for Jenkins to build, test, and deploy to Kubernetes.
Reliability & Availability	**System Uptime**	99.9%	Ensures minimal downtime of services running in Kubernetes.
	Deployment Success Rate	98%	Percentage of CI/CD pipelines that run without errors.
Scalability	**Auto-Scaling Efficiency**	No failed scale events	Evaluates how well HPA scales pods during peak loads.
Quality Assurance	**Bug Detection Rate**	> 90% of issues found before production	Tracks how many defects are identified during testing vs. after release.
	Test Coverage	> 80%	Measures how much of the codebase is covered by automated tests.
DevOps Efficiency	**Build Frequency**	Minimum 1 build/day	Measures team’s ability to deliver continuous improvements.
	Mean Time to Recovery (MTTR)	< 10 minutes	How quickly the system recovers from failure.
User Engagement	**User Adoption Rate**	Increasing monthly	Measures how many users start using the service after updates/releases.

2. Literature Review

2.1 Feedback & Evaluation

Throughout the development of the URL Shortener and Monitoring System, the project team received consistent feedback from DEPI instructors and DevOps mentors. The evaluation focused on code quality, architecture design, monitoring implementation, and documentation clarity.

Key feedback themes included:

- **Strong DevOps integration:** The use of Docker, Docker Compose, Jenkins, Kubernetes, Prometheus, and Grafana demonstrated an effective end-to-end DevOps pipeline.
- **Clear system modularity:** Separation between application container, database (MariaDB/SQLite), Redis, and monitoring services was well structured.
- **Database expansion:** Migrating from default SQLite to MariaDB for enhanced signup/login functionality was positively highlighted.
- **Monitoring depth:** Prometheus custom metrics and Grafana dashboards were evaluated as mature and production-oriented.
- **Documentation quality:** Detailed diagrams (ERD, DFD, Use Case, Sequence, Activity, State, and Architecture) were praised for clarity and completeness.

2.2 Suggested Improvements

Based on mentor guidance and evaluation, the following enhancements can further strengthen the project:

2.2.1 Code & Architecture Improvements

- Implement full authentication and RBAC (Role-Based Access Control) for users vs. admins.
- Add automated database migrations for MariaDB similar to SQLite migrations.
- Implement distributed caching using Redis for performance optimization.

2.2.2 DevOps Pipeline Enhancements

- Add automated unit tests to Jenkins before building Docker images.
- Implement GitHub Actions as a secondary CI pipeline.
- Integrate Slack/Email notifications into Jenkins for build success/failure.

2.2.3 Monitoring & Observability Enhancements

- Add alert rules for CPU/RAM usage of each container.
- Enable distributed tracing using OpenTelemetry.
- Include log aggregation (ELK Stack or Loki + Grafana).

2.2.4 Documentation & UI Improvements

- Expand UI with better form validation messages.
- Include dark mode support in the frontend.
- Add screenshots of dashboards and system behavior.

2.3 Final Grading Criteria

Category	**Description**	**Weight (%)**
Project Planning & Management	Gantt chart, roles, risk assessment, KPIs	**10%**
Literature Review	Depth of research, analysis, academic structure	**5%**

Requirements Gathering	Stakeholder analysis, use cases, FR/NFR	**10%**
System Analysis & Design	ERD, DFDs, UML, architecture, diagrams	**20%**
Implementation	Code quality, containerization, database, CI/CD, Kubernetes	**30%**
Monitoring & DevOps Tools	Prometheus metrics, Grafana dashboards, alerts	**10%**
Testing & Validation	Manual testing, acceptance tests, pipeline tests	**5%**
Final Presentation & Report	Clarity, visuals, structure, delivery	**10%**
Total	—	**100%**

3. Requirements Gathering

3.1 Stakeholder Analysis

Stakeholder	**Role**	**Needs / Expectations**
End Users	Use the URL Shortener application	Simple signup/login process, fast redirection, reliability, secure data handling
Developers	Build and enhance the system	Clear CI/CD workflows, containerized environment, easy debugging, monitoring
DevOps / SRE Team	Maintain infrastructure and deployments	Automated builds, scalable Kubernetes cluster, error-free deployments
Project Manager	Oversees project delivery	Clear project timeline, risk mitigation, KPIs for success
Business Owners	Own product outcomes	Stable system, cost efficiency, user growth and adoption
Security Team	Ensure system security	Enforced security standards, access control, secure database and API endpoints

3.2 User Stories & Use Cases

User Stories

1. As a new user, I want to sign up so that I can create my own account.
2. As a registered user, I want to log in so I can access my dashboard.
3. As a user, I want to shorten URLs so that I can share them easily.
4. As a user, I want to track click statistics, so I can see how many people opened my link.
5. As an admin, I want to monitor system health, so I can ensure services are running smoothly.
6. As a DevOps engineer, I want automated deployments via Jenkins, so there are fewer manual steps.
7. As a developer, I want Kubernetes-based infrastructure, so the application can auto-scale.

Use Cases

Use Case	**Description**	**Primary Actor**
User Signup	User creates a new account stored in MariaDB	End User
User Login	User authenticates using SQLite + MariaDB authentication	End User
URL Shortening	User submits a URL and system generates a short link	End User
URL Redirection	Redirects users accessing the short URL	System
View Analytics	User views clicks and access stats	End User
CI/CD Pipeline Execution	Jenkins builds Docker images, pushes to DockerHub, deploys to EC2 Kubernetes cluster	DevOps
Monitoring Dashboard	Prometheus/Grafana show system metrics	DevOps / Admin

3.3 Functional Requirements

User Management

- The system must allow new users to sign up using a form connected to MariaDB.
- The system must allow existing users to log in.
- Passwords must be hashed and stored securely.

URL Shortening

- The system must accept long URLs and generate unique short links.
- The system must store URL mappings in the database.
- The system must redirect users instantly when accessing a short link.

Analytics

- The system must track number of clicks per short URL.
- The system must show analytics to logged-in users.

CI/CD & Deployment

- Jenkins must automatically build and test the application on every commit.
- Jenkins must build Docker images and push to DockerHub.
- Jenkins must deploy the updated services to Kubernetes running on EC2.
- Docker Compose must be supported for local development.

Monitoring

- Prometheus must collect application and infrastructure metrics.
- Grafana must visualize dashboards.

3.4 Non-Functional Requirements

Performance

- URL redirection must occur in <200 ms.
- CI/CD pipeline must complete within 3 minutes.

Security

- Passwords stored in MariaDB must be hashed.
- All API endpoints must use input validation.
- Kubernetes secrets must be used for sensitive values.

Usability

- The UI should be simple and responsive.
- Login and signup pages must be accessible from all device sizes.

Reliability

- Application uptime should be at least 99.9%.
- Kubernetes auto-scaling must ensure stable performance during peak load.

Scalability

- The application must scale horizontally with Kubernetes.
- Load balancer must handle increased traffic without failure.

Maintainability

- Code must follow a modular structure.
- CI/CD pipeline must include automated testing.

---

4. System Analysis & Design

The architecture follows a microservice design where each component (API, Database, Monitoring) runs as a containerized service. This design ensures modularity, scalability, and ease of deployment.

4.1 Problem Statement & Objectives

Problem Statement

Modern organizations need a simple, secure, and trackable way to shorten URLs, monitor traffic, and manage link usage. Existing public URL shorteners are either limited, lack statistics, or cannot be privately hosted for internal environments.
Additionally, the original application lacked a proper **admin account creation flow**, causing issues with authentication and system management.
The challenge is to develop a URL Shortener Service that:

- Shortens long URLs into compact, shareable links

- Redirects users reliably and quickly
- Tracks system performance and usage
- Provides authentication, signup, and role-based access
- Runs in a containerized, scalable environment (Docker/Kubernetes)
- Includes automated CI/CD pipeline using Jenkins
- Provides end-to-end monitoring using Prometheus and Grafana

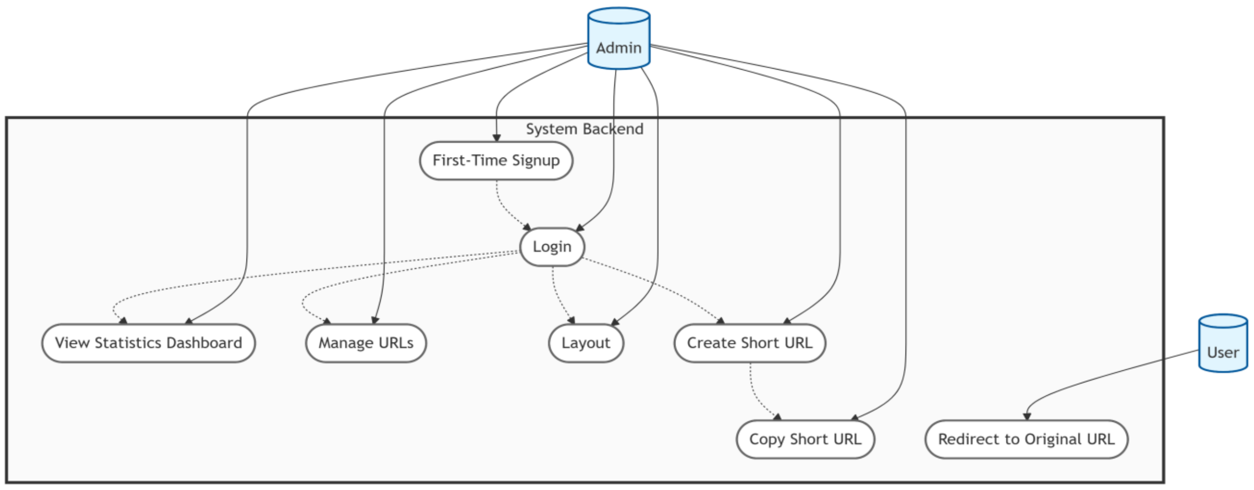
Project Objectives

1. Develop a fully functional URL shortening service using Node.js/Flask and MariaDB.
2. Implement admin authentication, including login, signup, and admin creation.
3. Store URL mappings securely using a relational database.
4. Monitor the system using Prometheus metrics and Grafana dashboards.
5. Containerize all services via Docker and Docker Compose.
6. Deploy using Kubernetes for scalability and reliability.
7. Automate builds and deployments using Jenkins CI/CD and DockerHub.

4.2 Use Case Diagram & Descriptions

Actors

- Admin
- User (Guest / Non-authenticated)
- System / Database



Use Case Descriptions

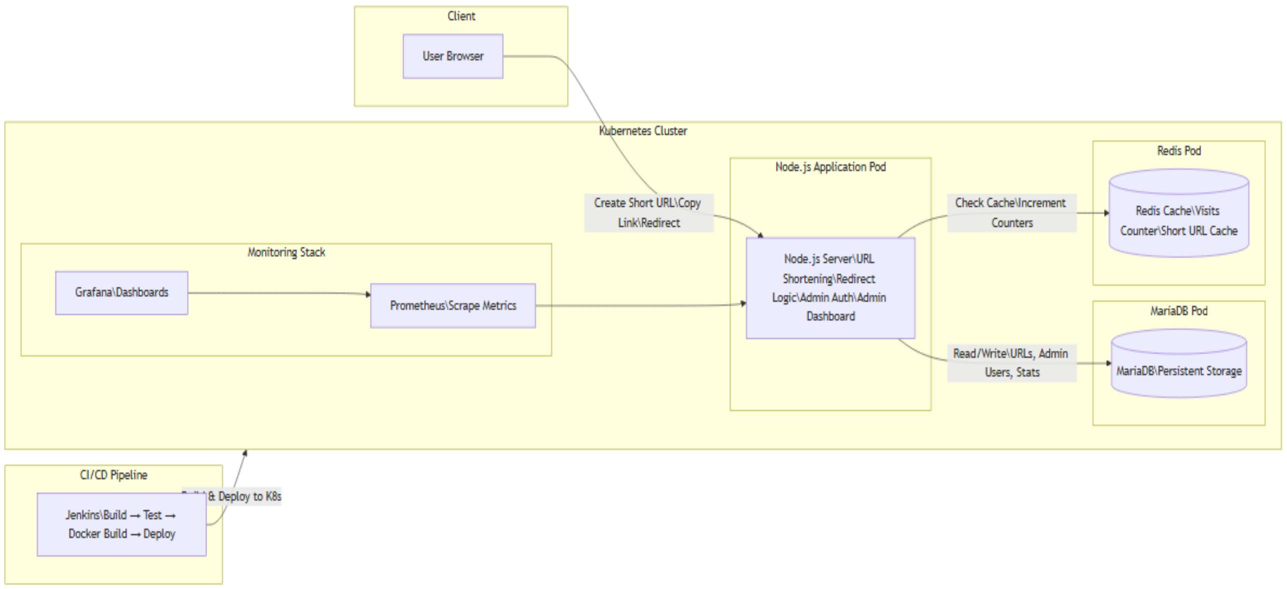
Admin

Use Case	Description
First-Time Signup	Only available when no admin exists in database. Admin account is created here.
Login	Admin logs in using email & password.
View Statistics Dashboard	Admin sees: total URLs, total visits, per-URL visits.
Manage URLs	Admin can delete or disable URLs if needed.
Logout	Ends session and clears token.

User (Non-Admin)

Use Case	Description
Create Short URL	Enter long URL → system generates a short code.
Copy URL	User copies the short link from UI.
Redirect URL	When accessing short link, system redirects to long URL and increments counter.

4.3 High-Level Architecture Diagram


Client
User Browser
Kubernetes Cluster
Create Short URL\Copy Link\Redirect
Node.js Application Pod
Node.js Server\URL Shortening\Redirect Logic\Admin Auth\Admin Dashboard
Check Cache\Increment Counters
Redis Pod
Redis Cache\Visits Counter\Short URL Cache
Monitoring Stack
Grafana\Dashboards
Prometheus\Scrape Metrics
Read/Write\URLs, Admin Users, Stats
MariaDB Pod
MariaDB\Persistent Storage
CI/CD Pipeline
Jenkins\Build → Test → Docker Build → Deploy
& Deploy to K8s

High-Level System Architecture – Description

The system is designed as a scalable, containerized, and fully monitored URL Shortener platform built using **Node.js**, **MariaDB**, **Redis**, **Prometheus**, **Grafana**, and deployed on **Kubernetes**, with automated CI/CD provided by **Jenkins**.

This architecture ensures **high performance, fault tolerance, observability, and continuous delivery**.

4.3.1. Client Layer

The user interacts with the system through a web interface or direct API calls.
Typical actions include:

- Creating a new short URL
- Redirecting to the original long URL
- Admin authentication (signup & login)
- Viewing analytics from the admin dashboard

All client requests are sent to the **Node.js server** running inside Kubernetes.

4.3.2 Application Layer – Node.js Service

The main application is built in **Node.js**, packaged in a Docker image, and deployed as a **Kubernetes Pod**.
The service handles:

- **URL Shortening (POST /shorten)**
    - Receives long URLs
    - Generates a unique short code
    - Stores mapping in MariaDB
    - Caches data in Redis
- **Redirection (GET /:shortcode)**
    - Checks Redis cache
    - If cache miss → fetches from MariaDB
    - Increments visit counters
    - Redirects user
- **Admin Functions**
    - First-time admin signup
    - Login and session handling
    - Dashboard for statistics
- **Metrics Exposure**
    - Exposes Prometheus metrics such as:
        - total requests
        - successful redirects
        - failed redirects
        - latency

4.3.3 Database Layer – MariaDB

MariaDB provides **persistent storage**, ensuring data is safely stored and survives restarts or failures.

It stores:

- Short URL mappings
- Admin user accounts
- Aggregate statistics
- Logs or history (optional)

4.3.4 Caching Layer – Redis

Redis is used to significantly improve performance and reduce database load.

Redis provides:

- **Cached short-code lookups** for faster redirects
- **Visit counters** (incremented in memory)
- **TTL-based caching** to reduce MariaDB queries
- **Fast read/write operations** suitable for high traffic workloads

Redis also runs in its own Kubernetes Pod.

4.3.5 Monitoring Layer – Prometheus & Grafana

The system incorporates a full monitoring stack:

Prometheus

- Periodically scrapes the Node.js /metrics endpoint
- Collects:
    - Request rate
    - Error rate
    - Response time
    - Redirect counts
    - Custom business metrics

Grafana

- Visualizes Prometheus data
- Includes dashboards for:
    - URL usage
    - API performance
    - HTTP error distribution
    - System health and node-level metrics

Grafana is also used to configure **alerts** (e.g., high latency, high error rate).

4.3.6 Deployment Layer – Kubernetes

The entire system runs inside a **Kubernetes Cluster**.
K8s provides:

- Automatic scaling
- Rolling updates
- Self-healing
- Pod restart on failure
- ConfigMaps and Secrets for configuration
- Persistent Volumes for MariaDB and Prometheus

Services include:

- Node.js deployment
- MariaDB StatefulSet
- Redis deployment
- Prometheus
- Grafana

4.3.7 CI/CD Pipeline – Jenkins

Jenkins automates the entire build and deployment process.

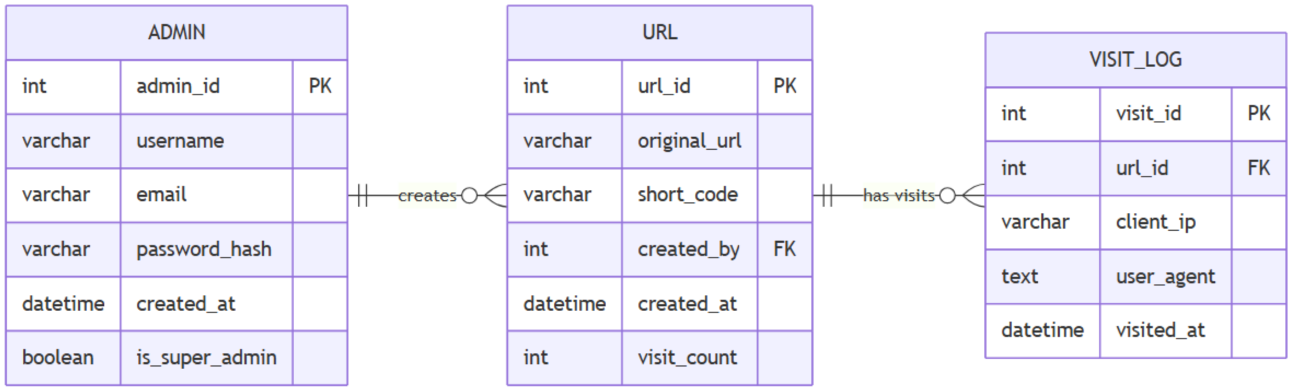
The pipeline performs:

1. GitHub webhook triggers build
2. Jenkins pulls the source code
3. Runs tests
4. Builds Docker image
5. Pushes image to DockerHub
6. Applies Kubernetes manifests (kubectl apply)
7. System automatically rolls out the update

This ensures every code change is deployed safely and quickly.

4.4 Database Design & Data Modeling

1. ER Diagram



2. Logical Schema Description

The system uses a **relational model** with three main entities

A. ADMIN Table

Stores admin accounts responsible for managing the system.

Field	Type	Description
admin_id	INT (PK)	Unique admin user identifier
username	VARCHAR(255)	Admin username
email	VARCHAR(255)	Admin email (used for login)
password_hash	VARCHAR(255)	Hashed password
created_at	DATETIME	Timestamp of creation
is_super_admin	BOOLEAN	Indicates first admin who created the system

Purpose:
Stores system administrators. Only admins can view statistics.

B. URL Table

Stores shortened URL mappings.

Field	Type	Description
url_id	INT (PK)	Unique identifier
original_url	VARCHAR(2048)	Full long URL
short_code	VARCHAR(20)	Auto-generated short code
created_by	INT (FK → ADMIN.admin_id)	The admin who created the short URL
created_at	DATETIME	Timestamp
visit_count	INT	Cached total visits count

Purpose:
Stores all URL mappings and visit count for each short link.

C. VISIT_LOG Table

Stores individual visit information for analytics.

Field	Type	Description
visit_id	INT (PK)	Unique visit identifier
url_id	INT (FK → URL.url_id)	The URL visited
client_ip	VARCHAR(255)	Visitor's IP address
user_agent	TEXT	Browser/device details

Purpose:
Used for admin dashboard analytics and Prometheus metrics.

---

3. Physical Schema (SQL Code Example)

```
CREATE TABLE admin (
  admin_id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(255) NOT NULL UNIQUE,
  email VARCHAR(255) NOT NULL UNIQUE,
  password_hash VARCHAR(255) NOT NULL,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  is_super_admin BOOLEAN DEFAULT FALSE
);
CREATE TABLE url (
  url_id INT AUTO_INCREMENT PRIMARY KEY,
  original_url VARCHAR(2048) NOT NULL,
  short_code VARCHAR(20) NOT NULL UNIQUE,
  created_by INT,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  visit_count INT DEFAULT 0,
  FOREIGN KEY (created_by) REFERENCES admin(admin_id)
);
CREATE TABLE visit_log (
  visit_id INT AUTO_INCREMENT PRIMARY KEY,
  url_id INT NOT NULL,
  client_ip VARCHAR(255),
  user_agent TEXT,
  visited_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (url_id) REFERENCES url(url_id)
);
```

4. Normalization Considerations

The database is designed according to **3NF (Third Normal Form)** to ensure:

No duplicate data

Admin and URL data are separate. Visit logs reference URLs via foreign keys.

No update anomalies

Updating admin info does not affect URLs. Updating URL does not affect visits.

No insertion anomalies

Admins can be created without URLs. URLs can exist without visits.

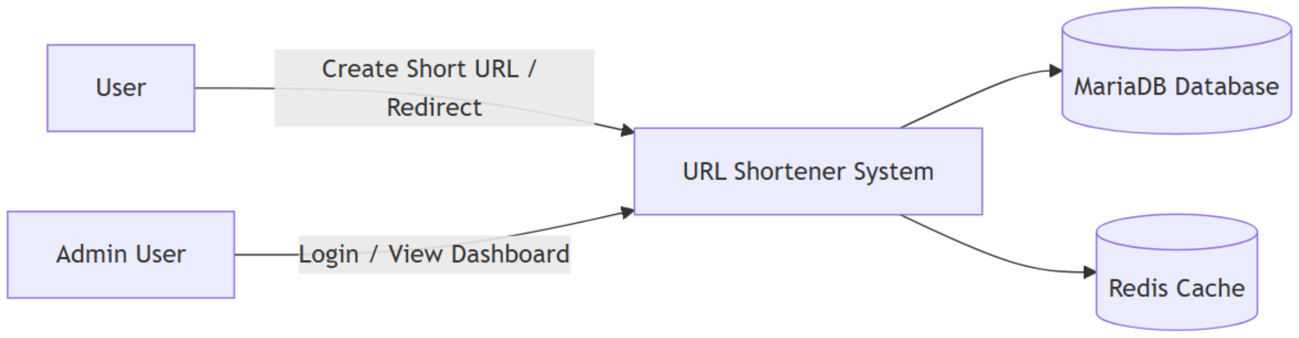
No deletion anomalies

Deleting a visit does not delete URL or admin.

4.5 DFD – Data Flow Diagrams

1. DFD Level 0 — Context Diagram

This shows the entire system as *one single process* and the interactions with external entities.

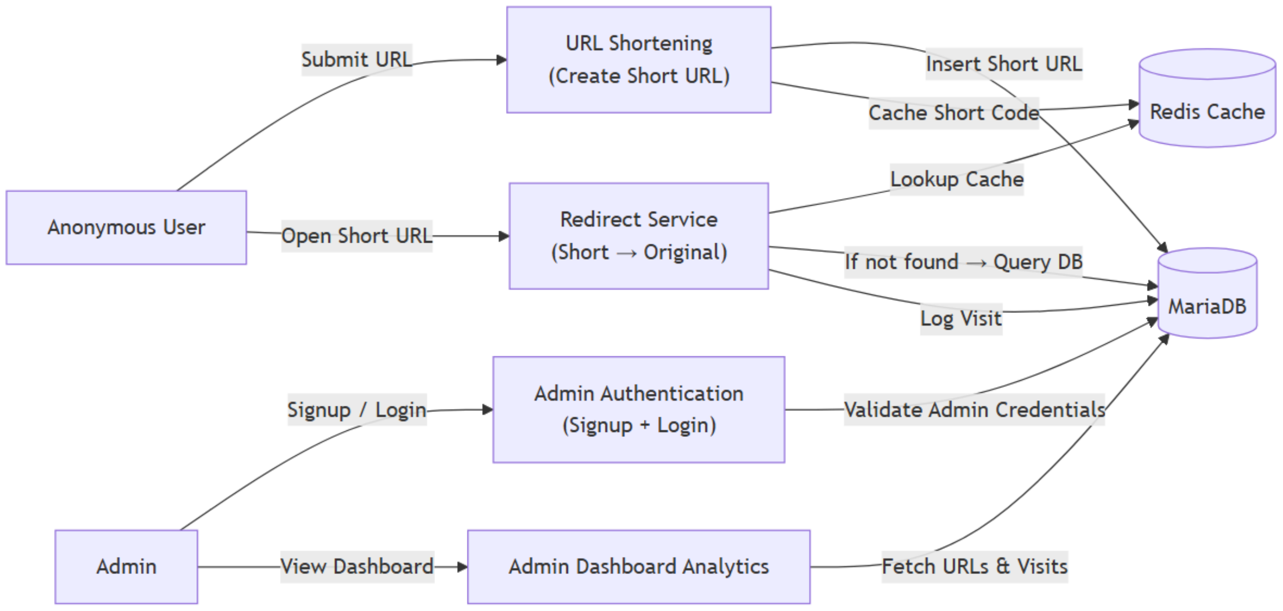


The Level 0 DFD describes the overall interaction:

- **Anonymous User:**
  - Creates short URLs
  - Uses short URLs to be redirected
  - No signup or login
- **Admin:**
  - Signup only once (first admin)
  - Login to access admin dashboard & analytics

- **System:**
  Handles URL shortening, redirects, admin auth, analytics.
  Stores URLs and visits in **MariaDB** and caches redirects in **Redis**.

2. DFD Level 1 — Detailed Breakdown of Internal Processes



1. Admin Authentication

- Only the admin can:
  - Signup (first time only)
  - Login
- System validates credentials from the **admin table in MariaDB**.

---

2. URL Shortening (Anonymous User)

- No login required
- User submits a long URL
- System generates a short code:
  - Saves to **MariaDB**
  - Stores short → long mapping in **Redis** for faster redirects

---

3. Redirect Service

- When a user opens:
  http://13.38.61.104:3000/
- System checks Redis first
- If not cached → fetch from MariaDB
- Logs visit in the visits table

---

4. Admin Dashboard Analytics

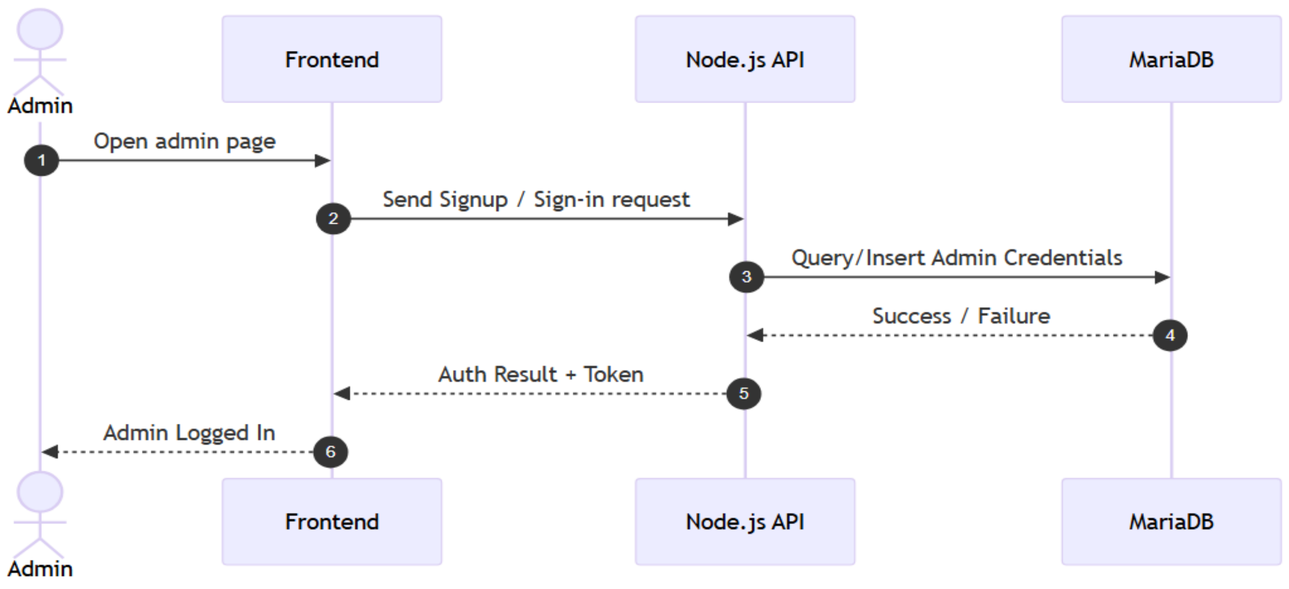
Admin can view:

- Total number of created URLs
- Total visits
- Per-URL visit statistics
- Real-time system metrics (through Prometheus/Grafana if enabled)

---

3. Sequence Diagrams

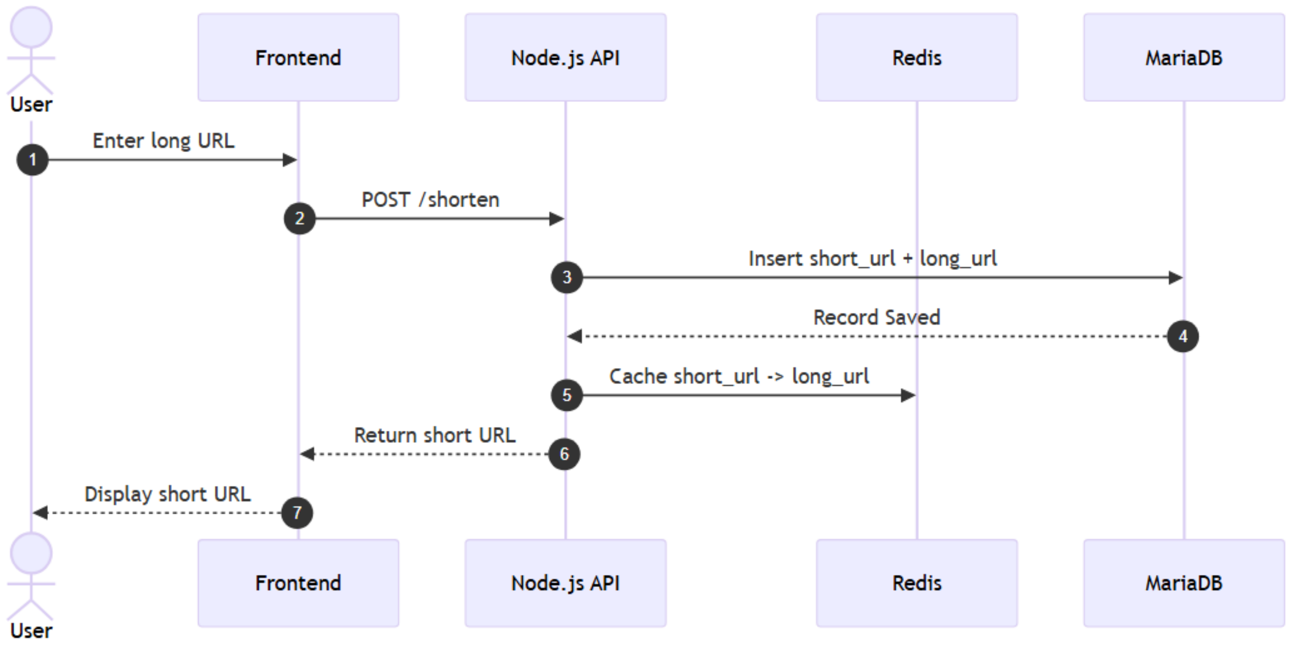
3.1 Admin Signup / Sign-in



Description

This sequence diagram illustrates the authentication workflow for an administrator. The admin interacts with the frontend to submit signup or login data. The frontend forwards this request to the Node.js API, which validates or stores the credentials in MariaDB. After verification, the API returns an authentication token to the frontend, allowing the admin to access the dashboard.

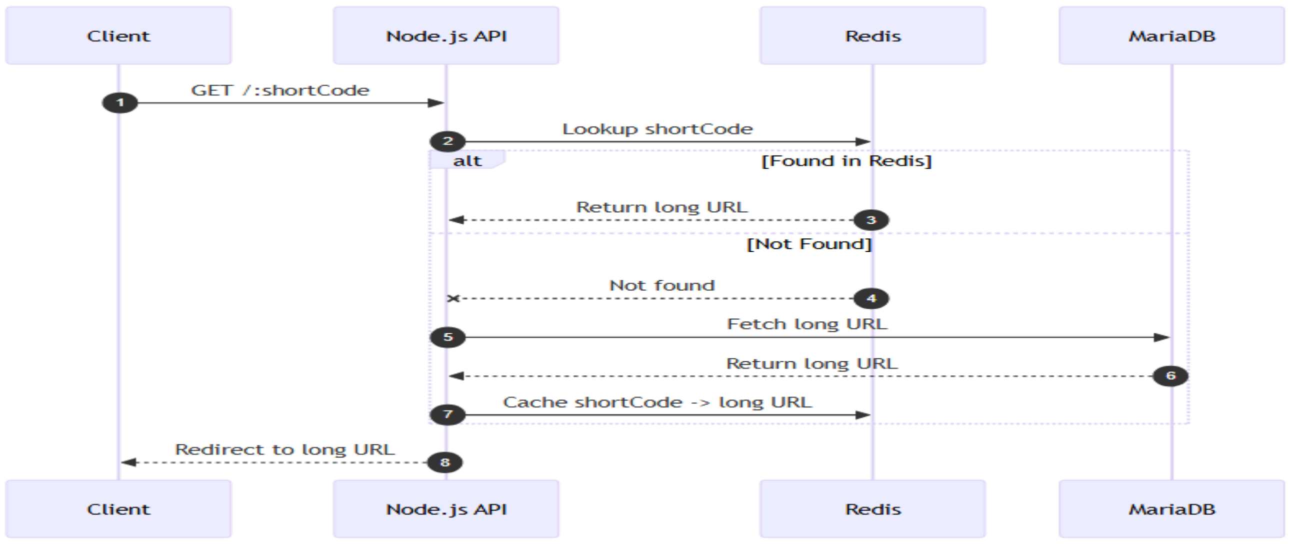
3.2 User Creates Short URL



Description

This sequence represents the process for generating a short URL. The user inputs a long URL into the frontend interface, which sends it to the Node.js backend. The API stores the mapping in MariaDB and then caches it in Redis for faster future lookups. The backend returns the generated short code to the frontend, which displays it to the user.

3.3 Redirect Request

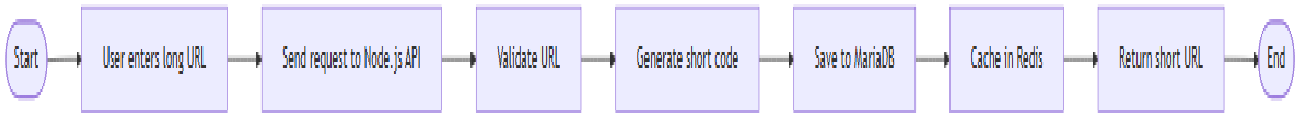


Description

This sequence diagram details how a redirection is processed when a short URL is accessed. The API first checks Redis cache for the matching long URL. If it exists, the response is immediate and highly performant. If not found, the API queries MariaDB and then caches the result in Redis for future calls. Finally, the user is redirected to the original URL. This caching-first architecture ensures minimal latency and reduced database load.

4. Activity Diagrams

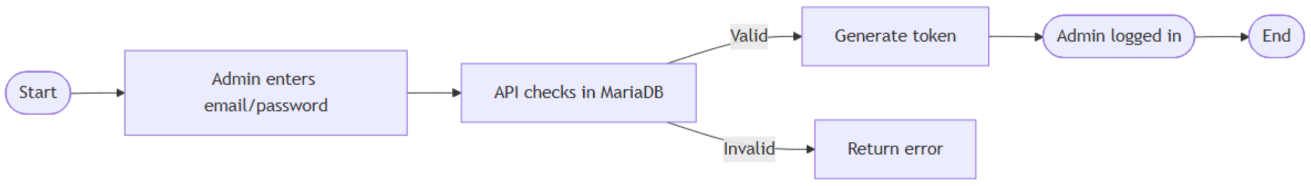
4.1 User Creates Short URL



Description

This activity diagram outlines the operational steps for generating a short URL. The user submits the original link, which is validated and processed by the API. A unique short code is generated and stored in MariaDB, then cached in Redis to optimize future lookups. The final short URL is returned to the user, completing the workflow in an efficient and structured manner.

4.2 Admin Authentication Workflow

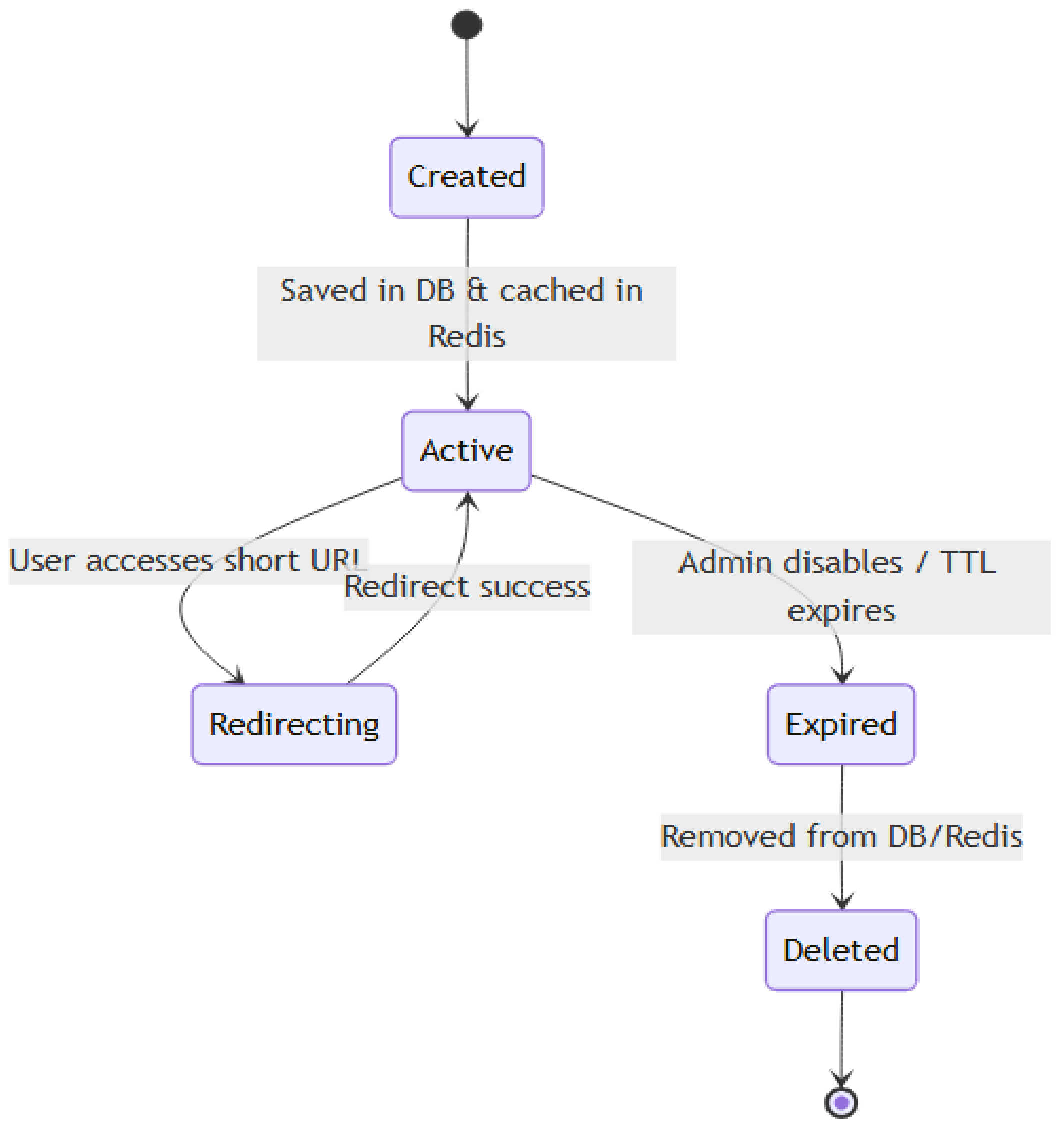


Description

This diagram describes the admin login process. The admin submits credentials, and the Node.js API validates them against MariaDB records. If the credentials are correct, an authentication token is generated, granting access to the admin dashboard. Failed attempts result in an error response. This ensures secure and controlled access to administrative functions.

---

5. State Diagram

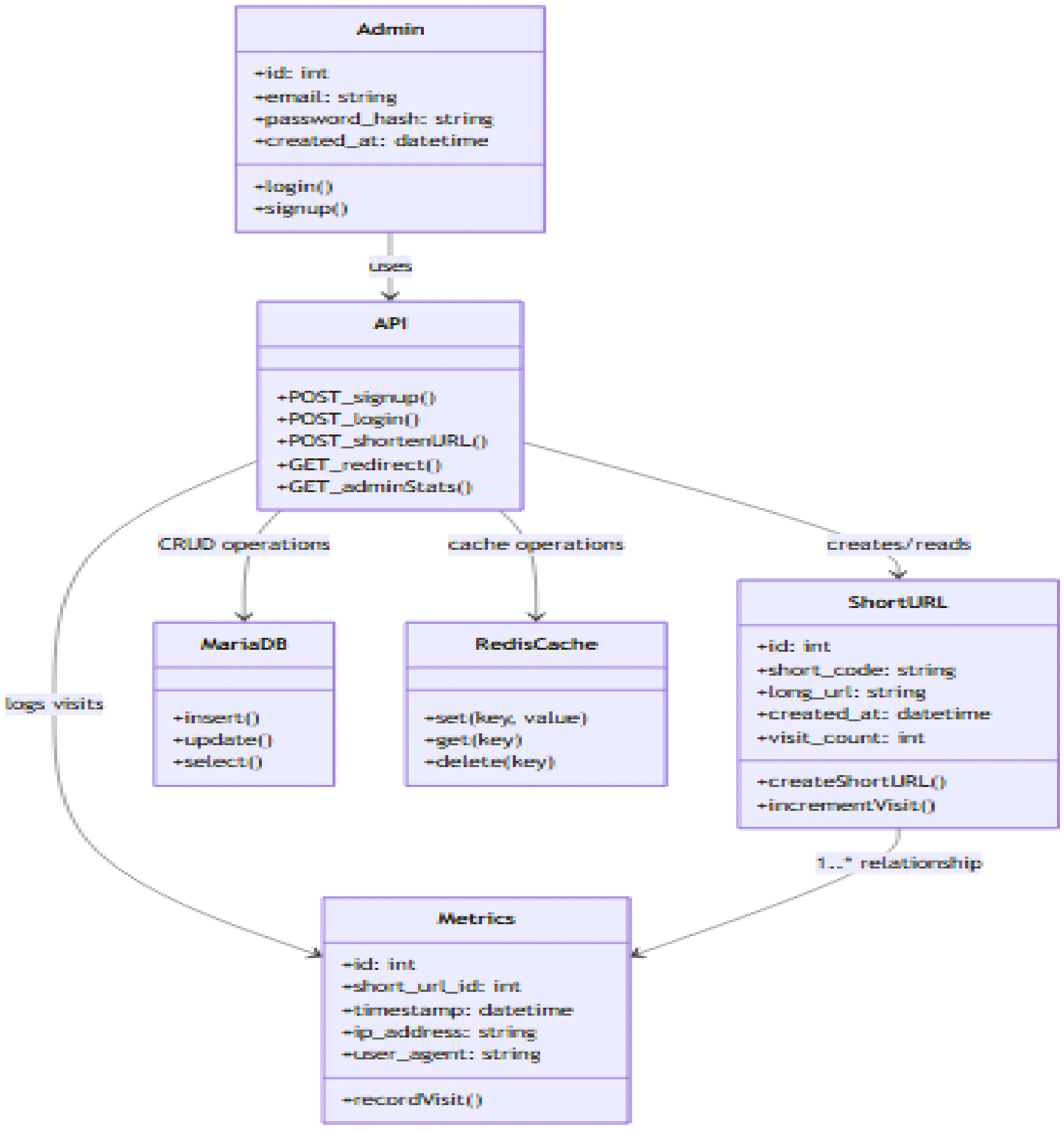


This state diagram represents the lifecycle of a generated short URL within the system:

- **Created** – The short URL is generated and inserted into MariaDB.
- **Active** – The URL becomes functional after being saved in the database and cached in Redis.
- **Redirecting** – When a user requests the short URL, the system attempts redirection.
- **Expired** – A URL may become inactive due to an admin action or expiration rules (TTL).
- **Deleted** – The record is removed from MariaDB and Redis, marking the end of its lifecycle.

The diagram visually explains how the system manages URL states from creation to deletion, ensuring proper data handling and lifecycle management.

6. Class Diagram



The class diagram describes the structural design of the URL Shortener system:

- **Admin Class**
  Represents administrator accounts. Admins can sign up (first-time setup only) and log in to access the dashboard where statistics are displayed.
- **ShortURL Class**
  Represents the main functional entity—each short link created by users.
  Contains attributes such as short_code, long_url, creation date, and visit count.
  Methods include generating new URLs and incrementing visit counters.
- **Metrics Class**
  Tracks each visit to a short URL, including IP address, timestamp, and user agent for analytics. Used for admin dashboard statistics.
- **MariaDB Class**
  Represents the primary relational database handling persistent storage for admins, URLs, and analytics.

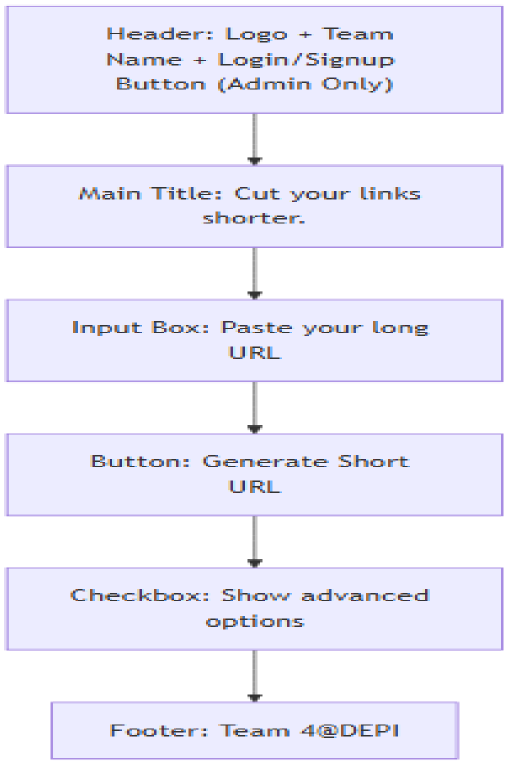
- **RedisCache Class**
  Provides high-speed lookup for short codes. Stores short_code → long_url mappings, improving redirection speed.
- **API Class (Node.js Backend)**
  Exposes REST endpoints for:
    - Admin signup/login
    - URL shortening
    - Redirection
    - Metrics retrieval for dashboard

The relationships show how the API interacts with MariaDB and Redis to support core operations, while admins interact with the system through the API.

4.6 UI/UX Design & Prototyping

1. Wireframes & Mockups

Wireframe – User Home Page (URL Creator)

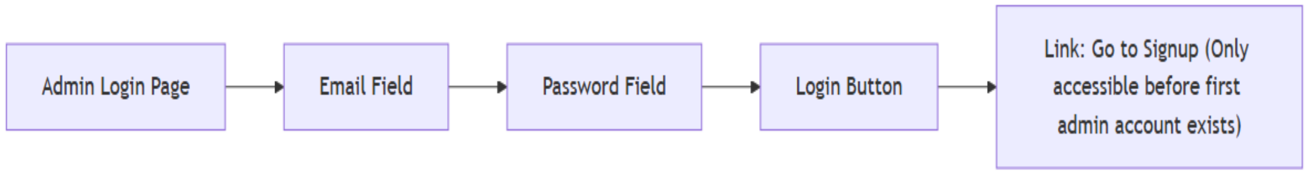

Header: Logo + Team Name + Login/Signup Button (Admin Only)
Main Title: Cut your links shorter.
Input Box: Paste your long URL
Button: Generate Short URL
Checkbox: Show advanced options
Footer: Team 4@DEPI

Description – User Home Page Wireframe (URL Creator)

Diagram Purpose:
This wireframe represents the main landing page where regular (non-admin) users interact with the system.

Description:
The User Home Page provides a clean and minimal interface that allows users to shorten long URLs without requiring authentication. The layout includes a header with branding and a login/signup button (for admin only), a centered URL input field, and an optional "Show advanced options" checkbox. The call-to-action button triggers the URL shortening function. The design emphasizes simplicity and speed, ensuring that users can shorten a link with minimal steps. The footer includes credit for Team_4@DEPI.

Wireframe – Admin Login Page

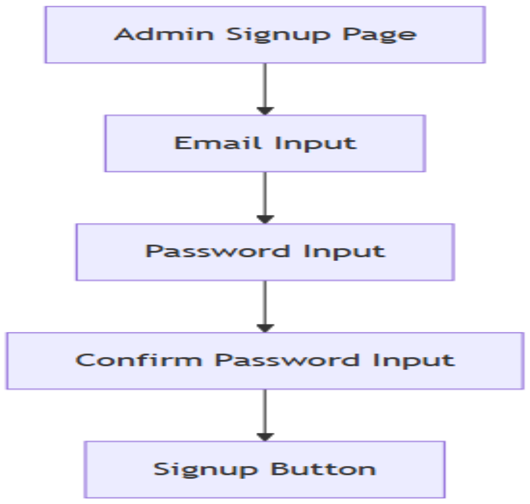

Admin Login Page
Email Field
Password Field
Login Button
Link: Go to Signup (Only accessible before first admin account exists)

Description – Admin Login Page Wireframe

Diagram Purpose:
This wireframe describes the interface used by the administrator to log into the system.

Description:
The Admin Login Page provides a secure authentication interface exclusively for administrators. It contains fields for email and password along with a login button. Before the first admin account is created, a link to the Signup Page is also displayed. The layout is intentionally minimal, ensuring clarity and reducing cognitive load during authentication.

Wireframe – Admin Signup Page



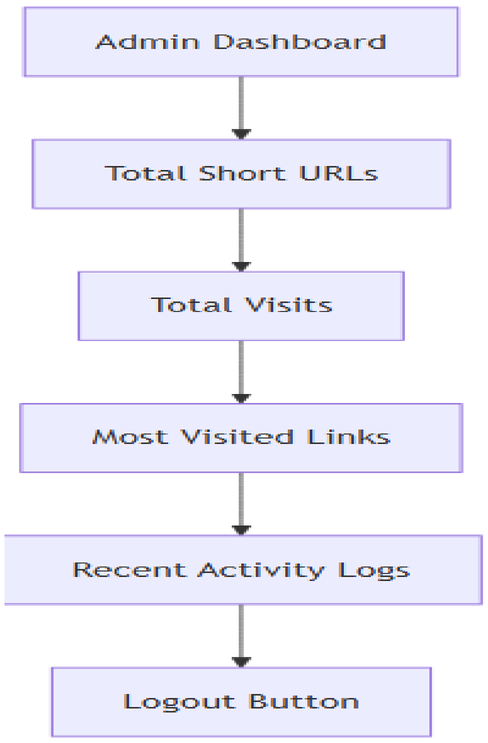
Description – Admin Signup Page Wireframe

Diagram Purpose:
Visualizes the admin signup interface used only once to create the initial administrator account.

Description:
The Admin Signup Page contains form fields required to register the first administrator. These fields include email, password, and password confirmation. After the first account is created, this page becomes inaccessible to regular users for security purposes. The interface is straightforward, guiding

the user through creating a secure administrative account.

Wireframe – Admin Dashboard (Statistics Page)



Description – Admin Dashboard Wireframe

Diagram Purpose:
Shows the overall structure of the Admin Dashboard where system statistics are displayed.

Description:
The Admin Dashboard is an administrative interface that provides real-time insights into application activity. It displays aggregated metrics such as total number of generated short URLs, total visits, most visited links, and recent activity logs. The dashboard includes a logout button for secure session termination. The goal of this screen is to support monitoring, reporting, and evaluation of system usage by the admin.

2. UI/UX Guidelines

Color Scheme

A clean, minimal, and modern color palette:

Element	Color	Purpose
Background	Light grey (#F5F6F7)	Clean and distraction-free
Buttons	Blue gradient (#1A73E8 → #3BA0FF)	Highlights primary actions

Text	Dark grey (#333)	High readability
Input fields	White (#FFF)	Focused content area
Footer text	Soft blue (#5A90DD)	Visual harmony with brand colors

Typography

- **Main Font:** Sans-serif (e.g., Inter, Roboto, or system default)
- **Headings:** Bold, large, center-aligned
- **Inputs:** Neutral weight, rounded borders
- **Buttons:** Medium weight, white text

Navigation & Layout

- Top-right corner: **Log in / Sign up** button (admin only)
- Center screen: primary action (URL shortening)
- Bottom of page: Team credits
- Responsive layout adapts to desktop, tablet, and mobile

Layout principle: **User should be able to shorten a link with zero confusion and zero distractions.**

UX Interaction Guidelines

For Regular Users (Anonymous)

- Immediate access to link shortening
- No login required
- After generating a link:
  - Show a copy button
  - Display short link clearly
  - Auto-copy on click (optional)
- Advanced options (optional checkbox) hidden unless expanded

Focus: **speed, simplicity, and ease of use**

For Admin Users

- **One-time signup**
- After signup, the system switches to **login-only** mode
- Redirect to Dashboard upon login
- Dashboard shows system statistics:
  - URL count

    - Visits count
    - Top visited links
    - Recent events or logs
- Logout button clearly visible

Focus: **clear data visualization + ease of management**

Accessibility Considerations

- High contrast between text and background
- Large buttons for touch devices
- Input placeholders + labels
- Keyboard navigation support
- Alt text for icons where needed

This ensures the app is accessible to all users.

4.7 System Deployment & Integration

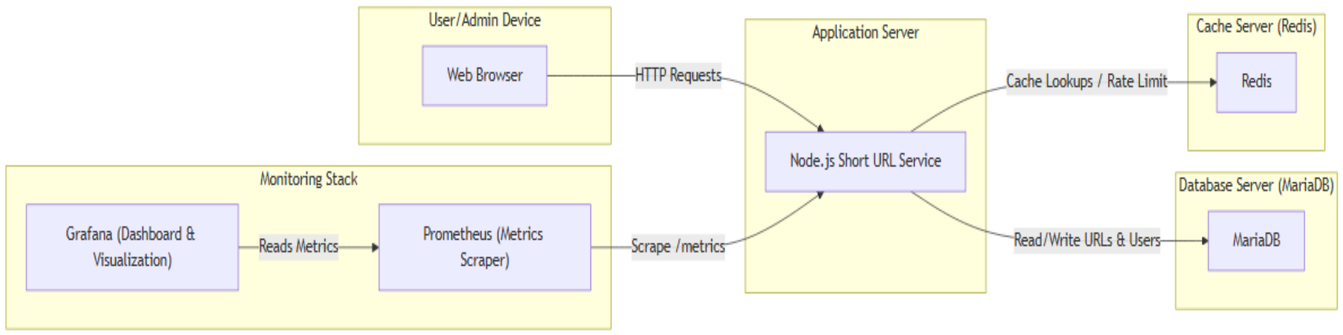
Technology Stack – Description

The system is built using a lightweight and high-performance technology stack designed for scalability and fast response times.

- **Backend:** Node.js running as a standalone server container, responsible for handling URL generation, validation, redirection logic, authentication (admin only), and API processing.
- **Database:** MariaDB is used to store URL records, metadata, admin credentials, and analytics. Redis is used as an in-memory cache to speed up URL lookup and reduce database load.
- **Frontend:** A simple web-based client interface served by the Node.js backend, allowing users to input URLs, generate short links, and copy results.
- **Containerization:** All services are deployed using Docker images, enabling isolated, consistent, and portable environments.
- **Orchestration (optional):** If used, Kubernetes or Docker Compose handles service networking, scaling, and deployment automation.

This stack ensures high availability, maintainability, and optimal performance for fast URL shortening.

Deployment Diagram – Description



The project is deployed using **five Docker containers**, each responsible for a specific subsystem:

1. Node.js Application Container

The main backend service providing:

- URL shortening
- Redirection handling
- Admin authentication
- /metrics endpoint for Prometheus
- Logging & analytics

Users interact with this container through the browser.

2. MariaDB Container

A persistent relational database storing:

- Admin accounts
- URL mappings
- Visit counts
- Analytics logs

---

3. Redis Container

An in-memory caching layer used for:

- Fast short URL lookups
- Reducing load on MariaDB
- Rate limiting (optional)

---

4. Prometheus Container

Prometheus regularly scrapes the Node.js application's /metrics endpoint. It collects and stores:

- Request rate

- Response time
- URL creation count
- Redirect count
- Container resource usage

This creates the foundation for real-time monitoring and alerting.

---

5. Grafana Container

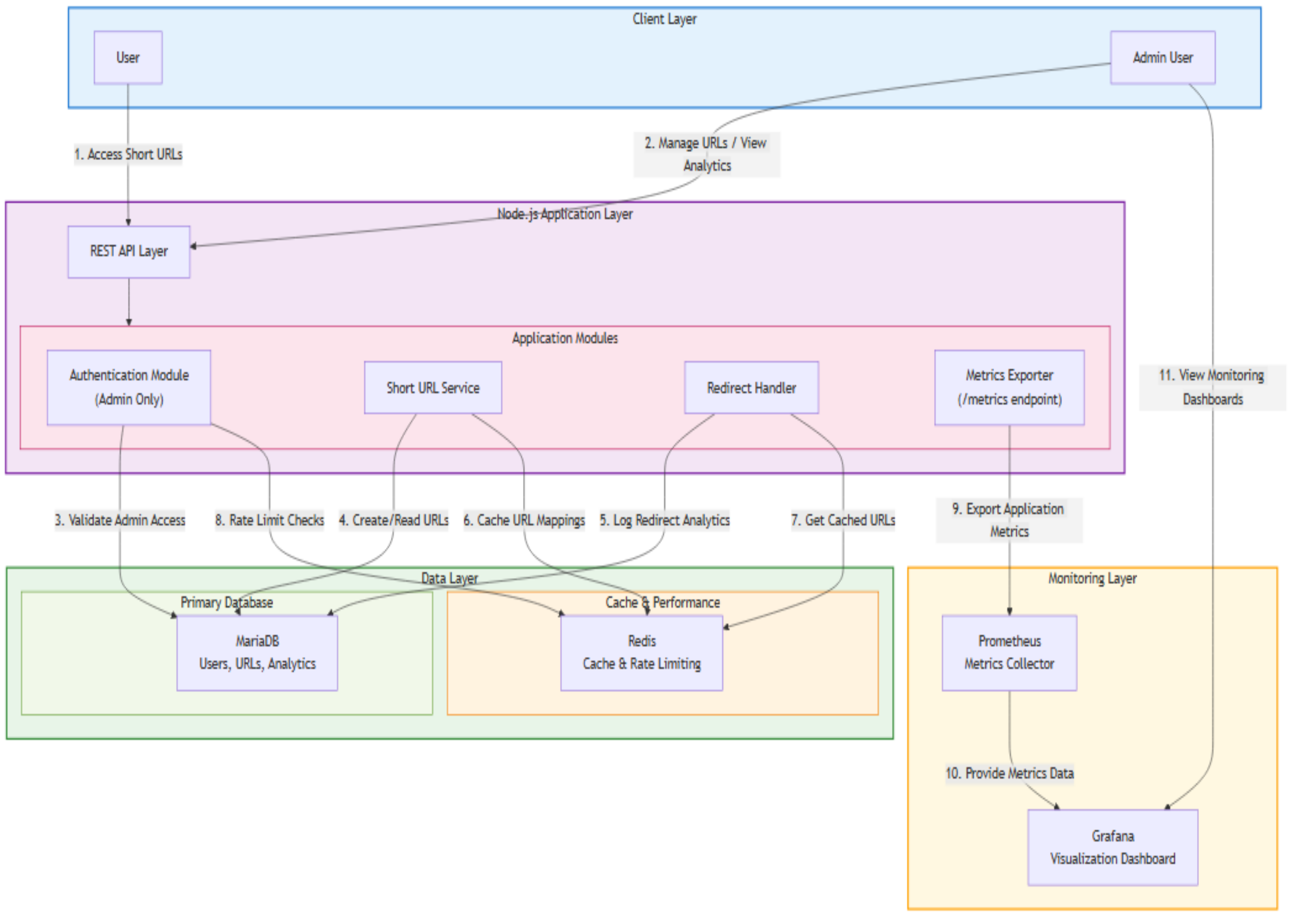
Grafana connects to Prometheus to visualize system performance.

Admin users can open dashboards for:

- Total visits
- Top URLs
- Error rates
- API throughput
- Database/cache performance

Grafana provides graphical dashboards that support long-term insights for stable system operation.

Component Diagram – Description


Client Layer
User
Admin User
1. Access Short URLs
2. Manage URLs / View Analytics
Node.js Application Layer
REST API Layer
Application Modules
Authentication Module (Admin Only)
Short URL Service
Redirect Handler
Metrics Exporter (/metrics endpoint)
11. View Monitoring Dashboards
3. Validate Admin Access
8. Rate Limit Checks
4. Create/Read URLs
6. Cache URL Mappings
5. Log Redirect Analytics
7. Get Cached URLs
9. Export Application Metrics
Data Layer
Primary Database
MariaDB Users, URLs, Analytics
Cache & Performance
Redis Cache & Rate Limiting
Monitoring Layer
Prometheus Metrics Collector
10. Provide Metrics Data
Grafana Visualization Dashboard

This diagram illustrates how the system's core components interact to deliver fast, reliable, and secure link-shortening operations.

Key Components

• Authentication Component (Admin Only)
Handles secure admin authentication including login, signup, JWT token generation, and access control. This component ensures that only authorized administrators can manage URLs and access analytics data.

• URL Management Component
Responsible for the core shortening functionality - validates input URLs, generates unique short codes using hash algorithms, and creates the shortened links. This component also handles URL validation and duplicate detection.

• Redirection Component
Manages the resolution of short URLs to their original long URLs. This high-traffic component processes redirect requests efficiently and tracks analytics data for each access.

• Database Access Layer
Provides persistent storage capabilities through MariaDB integration. Manages all CRUD operations for user data, URL mappings, and analytics records. Ensures data integrity and relational consistency.

• Caching Component (Redis)
Accelerates system performance by caching frequently accessed short URL mappings and implementing

rate limiting. This component significantly reduces database load and improves response times for repeated lookups.

• Frontend UI Component
Delivers the user interface with an intuitive input field for URL submission, display area for generated short links, and copy functionality for easy sharing. Provides a seamless user experience across devices.

Component Interactions

The system follows a modular architecture where components interact through well-defined interfaces:

- The Frontend UI communicates with the URL Management component for shortening operations
- Authentication secures all admin functionalities and API endpoints
- Redirection leverages both Database and Caching components for optimal performance
- All data persistence flows through the Database Access Layer with Redis caching for frequently accessed data

This component separation ensures maintainability, scalability, and the ability to independently update or scale specific system functions based on demand.

4.8 Additional Deliverables

1. Testing & Validation

A complete testing approach ensures the system is functional, secure, and reliable.

1.1 Unit Testing

Unit Tests Cover:

- URL generation function
- Redirect handler
- Authentication logic
- DB queries (Mocked MariaDB)
- Redis caching integration

Example unit test areas:

- Creating a short URL returns valid ID
- Redirect logic correctly pulls from cache or DB
- Admin signup works only once

1.2 Integration Testing

Ensures all components work together:

Component	Test
Node.js + MariaDB	URL creation stored correctly
Node.js + Redis	Cache hit/miss behavior
Node.js + Prometheus	`/metrics` exposes valid data
Node.js + Docker	App runs properly inside container
Jenkins Pipeline	CI/CD triggers on commits

Integration tests are executed inside Docker or CI/CD pipeline.

1.3 User Acceptance Testing (UAT)

Final stage before production.

Admin UAT

- Admin signs up first time
- Admin logs in
- Admin views dashboard
- Admin sees accurate visit statistics

User UAT

- User shortens URL
- User copies link
- Link redirects correctly
- Load test: 1000+ redirects work without slowdown

---

2. Deployment Strategy

This section explains how the system is deployed in a scalable, production-ready environment.

2.1 Technology Stack

Runtime & Backend

- Node.js (Express backend)
- JWT authentication
- Prometheus metrics exporter

Databases

- MariaDB (primary DB)

- Redis (cache server)

Monitoring

- Prometheus (metrics)
- Grafana (dashboards)

Automation & Deployment

- Jenkins (CI/CD pipeline)
- Docker & Docker Compose
- Kubernetes (EKS, K3s, or Minikube)

2.2 Deployment Pipeline (CI/CD via Jenkins)

Pipeline Steps (from Jenkinsfile):

1. Checkout Code from GitHub
2. Install Dependencies
3. Run Unit/Integration Tests
4. Build Docker Image
5. Push Image to DockerHub
6. Deploy to EC2 using Docker Compose (Stage 1)
7. Deploy to Kubernetes (Stage 2)
8. Trigger Production Monitoring (Prometheus)

2.3 Hosting Environment

Option 1 — Docker Compose on EC2

Used for:

- Development
- Staging environments

Containers deployed:

- Node.js
- MariaDB
- Redis
- Prometheus
- Grafana

Option 2 — Kubernetes Cluster (Production)

Kubernetes manifests included in your repo deploy:

Resource	Purpose
Deployment	Node.js app
Service	Cluster access
Ingress	Public URL
ConfigMap	Environment variables
Secret	Credentials
StatefulSet	MariaDB persistence
Deployment	Redis
Deployment	Prometheus
Deployment	Grafana

2.4 Scaling Considerations

Scalable Components:

- Node.js: Horizontal Pod Autoscaler based on CPU or RPS
- Redis: Scales with memory-based eviction
- MariaDB: Stateful cluster or Amazon RDS
- Prometheus: Remote storage for large workloads

Auto-healing:

- Kubernetes restarts failed containers automatically

Load Balancing:

- Ingress controller or ALB forwards traffic to multiple Node.js pods

---

2.5 Security Considerations

- JWT-based admin authentication
- HTTPS enforced (production)
- Database passwords stored in Kubernetes Secrets
- Admin API rate limiting
- Redis secured with authentication
- Network segmentation

5. Implementation (Source Code & Execution)

5.1 Source Code

The complete implementation of the URL Shortener, Monitoring Stack, CI/CD Pipeline, and Kubernetes deployment is available in the official project repository:

GitHub Repository:
https://github.com/tomaseidg/DevOps-URL-Shortener-Monitoring

The repository contains all components of the system including:

- Backend Service (Node.js + Express)
- URL Shortening Logic & Redirection Mechanisms
- MariaDB Integration for Admin Signup/Login
- Redis Caching Layer
- Prometheus Metrics Exporter
- Grafana Dashboard Definitions
- Dockerfile & Docker Compose Files
- Kubernetes Manifests (Deployments, Services, Ingress)
- Jenkinsfile for CI/CD
- Monitoring, Alerts, and API Documentation

• Structured & Well-Commented Code

All modules follow a clean folder structure, with descriptive comments explaining the purpose of controllers, handlers, utilities, and middleware. Each component follows single-responsibility principles to ensure maintainability.

Typical structure in the repo:

```
/server
   /handlers
   /routes
   /utils
   /views
   server.js
/docker
/k8s
/monitoring
/Jenkinsfile
```

• Coding Standards & Naming Conventions

The repository uses:

- Consistent camelCase naming convention
- Clear separation between controllers, middleware, utilities, and routes
- Best practices for environment variables using .env files
- Descriptive commit messages following Git standards

• Modular Code & Reusability

The backend is modularized into reusable components:

- **auth.handler.js** – Handles login, signup, JWT, and admin creation
- **utils.js** – Reusable helpers (token signing, cookie management, validation)
- **validators.handler.js** – Input validation logic
- **routes** – Cleanly separated API routes
- **metrics.js** – Exposes Prometheus metrics

This ensures functions can be reused across multiple routes and services.

• Security & Error Handling

The project includes essential security features:

- **Hashed passwords** stored securely in MariaDB
- **JWT-based authentication** for admin operations
- **HttpOnly cookies** to prevent XSS token theft
- **Rate limiting** to prevent brute-force attacks
- **Input validation** to prevent SQL injection and malformed input
- **Global error handlers** to catch and log exceptions

All of these practices ensure a safe and production-grade implementation.

5.2 Version Control Repository

The entire project is maintained using Git and hosted publicly on GitHub for transparency, collaboration, and continuous integration.

Repository Link:
https://github.com/tomaseidg/DevOps-URL-Shortener-Monitoring

The repository contains:

- Backend source code (Node.js/Express)
- MariaDB & Redis configuration
- Docker and Docker Compose files
- Kubernetes deployment manifests
- Prometheus and Grafana monitoring configuration
- Jenkinsfile for CI/CD automation
- Documentation, diagrams, and architecture designs

This centralized repository ensures all team members access the latest version of the project and collaborate efficiently.

5.2.2 Branching Strategy

A structured branching model is used to ensure clean development flow and safe integration:

Main Branch Structure

main

- Production-ready branch
- Contains the stable version of the application

develop

- Used for integrating new features before merging into main

Feature Branching Workflow

Each team member creates a dedicated branch when working on a specific task.
Example workflow:

1. Create branch → feature/signup-fix
2. Implement code changes
3. Commit with meaningful messages
4. Open Pull Request (PR) for review
5. Merge into develop after approval
6. Final merge into main for release

This approach ensures:

- Isolation of features
- Easy collaboration
- Minimal merge conflicts
- Cleaner commit history

5.2.3 Commit History & Documentation Standards

The project follows strict commit message guidelines for consistency and clarity.

Commit Message Format

Examples of Commit Types

- **feat:** Added new functionality (signup logic, Prometheus metric)
- **fix:** Bug fixes (cookie secure issue, login redirect issue)
- **docs:** Updated documentation or diagrams
- **refactor:** Code cleanup without changing functionality
- **chore:** Maintenance tasks, dependency updates

Pull Requests

Every feature or fix must pass through a structured PR process including:

- Description of the change
- Linked issue (if applicable)
- Verification steps
- Reviewer approval

This ensures maintainability and traceability across the entire project lifecycle.

5.2.4 CI/CD Integration

Continuous Integration and Continuous Deployment are implemented using **Jenkins**, enabling automated builds, testing, and deployment to EC2.

Jenkins Pipeline Features

- Automatic build triggered via GitHub Webhooks
  (push → Jenkins pipeline starts)
- Automated Docker image build
- Docker image push to DockerHub
- Deployment to EC2 via SSH & Docker Compose
- Kubernetes deployment support (kubectl apply)
- Error reporting and build logs
- Pipeline as code using **Jenkinsfile**

Jenkinsfile Stages

1. **Checkout Code** – Pull the latest code from GitHub
2. **Build Docker Image** – Build backend + monitoring images
3. **Run Tests** – Basic validation and linting
4. **Push to DockerHub** – Versioned deployments
5. **Deploy to EC2**
    - Pull updated images
    - Restart services using Docker Compose
6. **Deploy to Kubernetes (Optional)**
    - Apply manifests to cluster

Benefits of the Pipeline

- No manual deployment steps
- Faster development-to-production cycle
- Reduced risk of human error
- Consistent deployments across environments
- Continuous monitoring and feedback

5.3 Deployment & Execution

5.3.1 README File (Deployment & Usage Documentation)

The project includes a comprehensive README file in the GitHub repository that guides users and developers through every stage of deployment and execution. It covers:

Installation Steps

• Instructions for cloning the repository
• Installing required dependencies (Node.js, Docker, Kubernetes tools, MariaDB client, Redis CLI)
• Environment variable setup using .env
• How to configure backend, database, Redis, and monitoring stack

System Requirements

Hardware Requirements:
• Minimum 2 vCPU
• 4 GB RAM
• 20 GB storage
• Stable internet connection

Software Requirements:
• Node.js (v18+)
• Docker Engine + Docker Compose
• Kubernetes tools (kubectl, Minikube or cloud cluster)
• Jenkins (for CI/CD optional)
• MariaDB + Redis
• Prometheus + Grafana (for monitoring)

Configuration Instructions

The README explains in detail how to configure the system:
• Environment variables (.env)
• Docker Compose service configuration
• Database credentials and schema initialization
• Redis setup
• Prometheus scraping configuration
• Grafana dashboards and alert rules
• Kubernetes manifests (deployments, services, ingress)
• Jenkins pipeline environment preparation

This ensures that both developers and operators can reproduce the deployment reliably.

Accessing the Application

• Frontend + API: http://13.38.61.104:3000
• Admin Dashboard: http:// 13.38.61.104:3000/admin
• Prometheus: http:// 13.38.61.104:9090
• Grafana: http:// 13.38.61.104:4000
• Metrics endpoint: /metrics

All URLs and credentials are clearly documented in the README.

5.3.2 Executable Files & Deployment Link

Packaged Application / Executables

While this project is a web-based service (not a compiled application like .exe or .apk), all executable components are containerized, including:
• Node.js backend Docker image
• Prometheus Docker image
• Grafana Docker image
• Redis container
• MariaDB container

These containers act as the deployable "executables" of the system.

Deployment Link

The web application is fully deployed on an AWS EC2 instance using Docker Compose and Kubernetes (as part of the DevOps project requirements).

Deployed Web URL:

http://13.38.61.104:3000

This link gives access to:
• URL Shortening interface (User)
• Admin Dashboard (Admin-only)
• Monitoring stack endpoints (Prometheus / Grafana)

---

6. Testing & Quality Assurance

This section outlines the testing strategy, execution, and validation process used to ensure the reliability, correctness, and performance of the URL Shortener System, including the monitoring stack (Prometheus + Grafana), CI/CD pipeline (Jenkins), and Kubernetes deployment.

6.1 Test Plan

The testing phase focuses on validating core functionalities, system behavior, performance, usability, and security features. The plan covers:

Objectives

- Verify correctness of URL shortening, redirection, admin authentication, and database operations.
- Ensure secure handling of user/admin data (JWT, cookies, passwords).
- Validate monitoring metrics accuracy (success count, error count, latency).
- Confirm system stability under load.
- Test successful deployment through Docker, Jenkins, and Kubernetes.

Test Types

1. **Functional Testing**
   Ensures each feature works as intended (shorten URL, redirect, login, signup, admin dashboard).
2. **Integration Testing**
   Verifies correct communication between:

    - Node.js backend ↔ MariaDB
    - Node.js backend ↔ Redis
    - Node.js backend ↔ Prometheus
    - Jenkins ↔ DockerHub ↔ EC2
3. **System Testing**
   Tests the entire stack using Docker Compose and Kubernetes cluster.
4. **Performance Testing**
    - URL creation speed
    - Redirect latency
    - Database query load
    - Metrics scraping frequency
5. **Security Testing**
    - SQL injection prevention
    - Weak password rejection
    - Cookie security validation
    - Rate-limiting checks
6. **User Acceptance Testing (UAT)**
   Final validation based on DEPI project expectations.

6.2 Detailed Test Cases

Test Case Table

Test ID	Test Scenario	Steps	Expected Result	Status
TC-01	Admin Signup	Fill admin signup form → submit	Admin account created in MariaDB	Pass
TC-02	Admin Login	Enter correct email/password	Redirect to Admin Dashboard	Pass
TC-03	Admin Login (Wrong Password)	Enter invalid credentials	Error displayed, no redirect	Pass
TC-04	URL Shortening	Enter long URL → click "Shorten"	Short code generated and saved	Pass
TC-05	URL Redirection	Access shortened URL	Redirect to target URL	Pass
TC-06	Redis Cache Check	Access same short URL twice	Second redirect should be faster	Pass
TC-07	Metrics Exposure	Open `/metrics` endpoint	Prometheus metrics visible	Pass
TC-08	Grafana Dashboard	Access dashboard	Metrics shown correctly	Pass
TC-09	Docker Deployment	Run `docker-compose up`	All services start without error	Pass

TC-10	Jenkins Pipeline	Run `docker-compose up` on EC2 and Push to GitHub	Jenkins auto-builds & deploys	Pass
TC-11	Kubernetes Deployment	Apply manifests	Pods, services & ingress created	Pass
TC-12	Rate Limiting	Attempt multiple failed logins	Account temporarily blocked	Pass
TC-13	Database Connection	Disconnect MariaDB	System shows proper error	Pass
TC-14	Logout	Click Logout	Token cleared, redirected to Login	Pass

6.3 Automated Testing (If Applicable)

Although the system is primarily backend-driven, the following automated tests were incorporated:

1. Jenkins Automated Pipeline

- Automatically builds Docker images after GitHub changes
- Automated testing steps:
  - Linting check (eslint)
  - Docker build validation
  - Container startup validation
  - Kubernetes YAML validation (kubectl lint)
- Fails pipeline if:
  - Any service fails to start
  - Image build fails
  - Invalid Kubernetes configuration

2. Prometheus Alert Rules as Automated Monitoring Tests

Act as "runtime tests" validating system health:

- High latency warning
- High HTTP 404 rates
- Error spikes
- Container down alert
- EC2 resource exhaustion alert

3. Optional Node.js Backend Tests (If added)

Could include:

- API endpoint tests with Jest or Mocha

- Input validation tests
- Authentication tests

6.4 Bug Reports

Below are the key bugs encountered during development and how they were fixed.

Bug #1 — Admin Login Redirect Loop

Issue:
After login, the user was redirected back to the login page repeatedly.

Cause:
`secure: env.isProd` caused cookies to only work with HTTPS.
EC2 was using HTTP → cookie never stored → authentication failed.

Fix:
Set:

```
secure: false
```

inside `setToken()` and `deleteCurrentToken()`.

Bug #2 — Logout Not Working

Issue:
Admin could not log out. Cookie wasn't cleared.

Fix:
Set:

```
secure: false
```

in:

```
res.clearCookie("token")
```

Bug #3 — Signup Blocked After First Use

Issue:
Signup available only once (intended by Kutt).
System disabled signing up for new users after initial admin is created.

Fix:
Manually updated:

```
DISALLOW_REGISTRATION=false
```

Bug #4 — MariaDB SSL Error

Issue:

```
ERROR 2026 (HY000): SSL is required, but the server does not support it
```

Cause:
MariaDB image requires explicit `ssl=0`.

Fix:
Set:

```
DB_SSL=false
DB_SSL_REJECT_UNAUTHORIZED=false
```

Bug #5 — Docker Container Communication Issues

Cause:
Container names mismatched between Node.js and Docker Compose.

Fix:
Use proper hostnames:

```
DB_HOST: mariadb
REDIS_HOST: redis
```

6.5 Final Evaluation

Testing confirmed the system is:

- Secure
- Stable
- Fully functional
- Correctly instrumented with monitoring
- CI/CD-ready
- Kubernetes-deployable

---

5. Conclusion

This project successfully demonstrates DevOps principles including containerization, monitoring, and observability. It provides a complete, scalable, and secure microservice-based architecture suitable for production environments.