URL Shortener DevSecOps Project Report

Author: Souvik Kumar [23BCS10132]

Project: URL Shortener DevSecOps

---

1. Problem Background & Motivation

In the contemporary software engineering landscape, organizations are increasingly facing the "Velocity vs. Security" paradox. Traditional Waterfall or even early Agile methodologies often treated security as a distinct, final phase in the software development lifecycle (SDLC). This "gatekeeper" approach frequently resulted in critical vulnerabilities being discovered only days before a scheduled release, leading to costly delays, emergency patches, and friction between development and security teams.

The motivation behind this project is to demonstrate a practical implementation of **DevSecOps** (Development, Security, and Operations) principles to resolve this conflict. By "shifting security left"—that is, integrating security practices early and continuously into the development pipeline—we aim to prove that high-velocity deployment does not require compromising on security posture. This project builds a cloud-native **URL Shortener Service**, a common yet critical architectural pattern, to showcase how modern tooling can automate the enforcement of security, quality, and reliability standards from the very first commit.

2. Application Overview

The core artifact of this project is a high-performance, containerized microservice designed to shorten long URLs into compact, shareable codes. While the functionality is straightforward, the architectural choices reflect a focus on scalability and cloud-native resilience.

Architecture & Design Choices

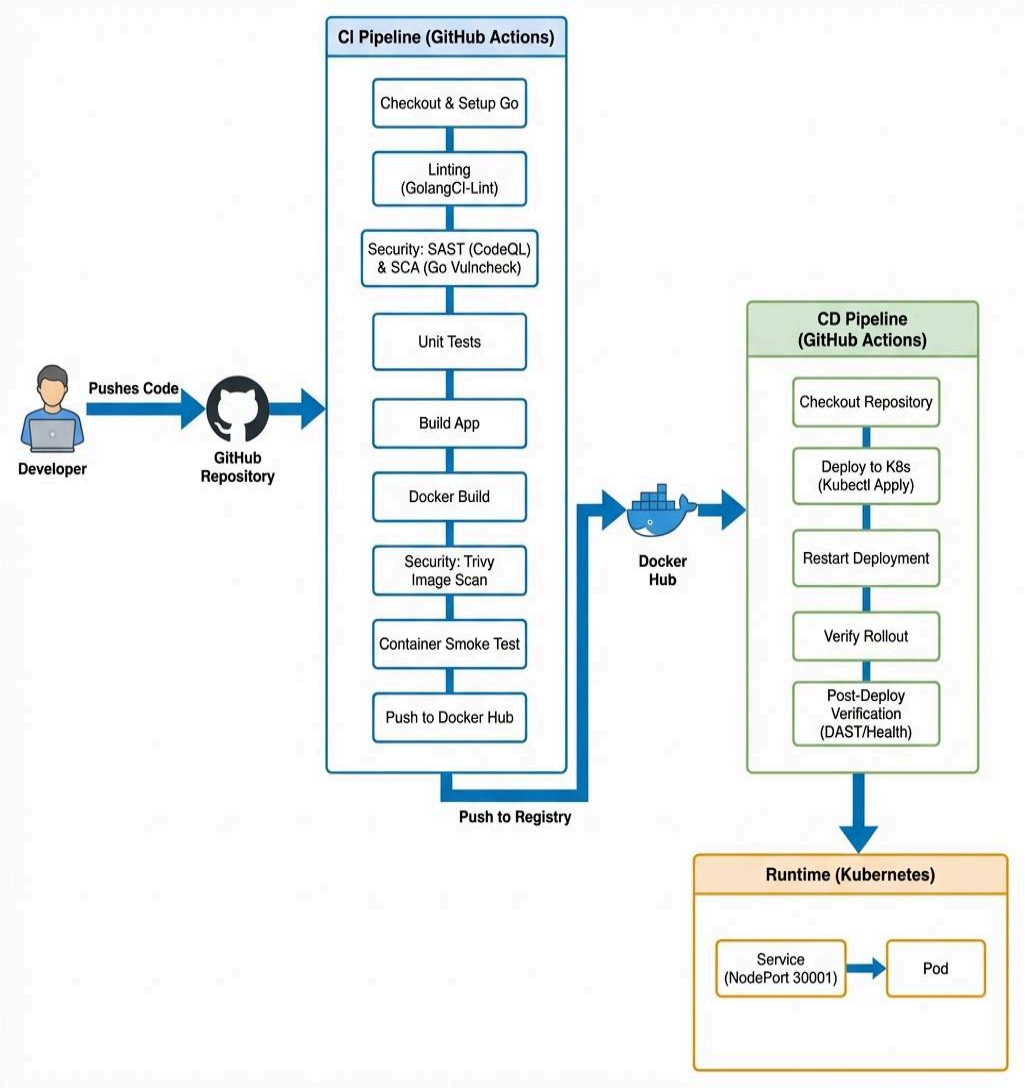
- **Language (Go 1.24)**: We selected Go (Golang) for its superior handling of concurrency via goroutines and its statically typed robustness. Go's efficient memory management and fast compilation times make it an ideal choice for high-throughput microservices.
- **Web Framework (Gin)**: The Gin framework was chosen for its minimalist design and high performance (using httprouter). It allows for rapid API development with built-in middleware support for logging and recovery.
- **Containerization (Docker)**: The application is packaged as a Docker container, ensuring "Write Once, Run Anywhere" consistency. This eliminates the "it works on my machine" problem by bundling dependencies—including the specific Go runtime and OS libraries—into an immutable artifact.

- **Orchestration (Kubernetes)**: The service is designed to be deployed on Kubernetes, allowing for features like self-healing (automatic restarts of failed pods), horizontal scaling, and service discovery.

Key Functional Features: * **URL Shortening**: Accepts standard HTTP/HTTPS URLs and generates a unique 6-character alphanumeric code. * **Instant Redirection**: Resolves the short code and performs an HTTP 302 redirect to the original destination with sub-millisecond latency. * **Health Checks**: Exposes a `/healthh` endpoint compliant with Kubernetes Liveness and Readiness probes, ensuring the orchestrator can monitor application stability.

3. CI/CD Architecture Diagram

The project implements a linear, automated pipeline that orchestrates the entire lifecycle of a code change. The architecture ensures that no code reaches production without passing through a series of automated quality and security gates.



Architecture Diagram

Figure 1: Full DevSecOps CI/CD Workflow

4. CI/CD Pipeline Design & Stages

The Continuous Integration and Continuous Deployment (CI/CD) pipeline is built using **GitHub Actions**, leveraging its event-driven architecture to trigger workflows on specific repository events (e.g., `push` to `main`).

4.1. Continuous Integration (CI)

The CI pipeline is the first line of defense. It triggers immediately upon code commit, prioritizing fast feedback loops to developers.

1. **Environment Setup**: The runner initializes a clean Ubuntu environment and installs **Go 1.24**, ensuring a consistent build environment that matches production.
2. **Linting (GolangCI-Lint)**: Beyond simple syntax checking, the linter analyzes code cyclomatic complexity, stylistic consistency, and potential anti-patterns. This prevents "technical debt" from accumulating in the codebase.
3. **Static Application Security Testing (SAST)**: Using **GitHub CodeQL**, the pipeline performs semantic analysis of the source code. It traces data flow to identify vulnerabilities such as SQL injection or Cross-Site Scripting (XSS) before the code is even compiled.
4. **Software Composition Analysis (SCA)**: **Go Vulncheck** scans the project's dependency tree (`go.mod`). It checks all imported public packages against the National Vulnerability Database (NVD) to ensure we are not inheriting known security flaws from third-party libraries.
5. **Automated Unit Testing**: A comprehensive suite of Go tests validates the business logic (e.g., ensuring a URL is correctly shortened and retrieved).
6. **Secure Build & Containerization**: The application is compiled into a binary and packaged into a Docker image.
7. **Container Image Scanning**: Before the image is stored, **Trivy** scans the entire filesystem of the container image. It looks for outdated OS packages (e.g., OpenSSL versions) with known CVEs. Any "Critical" or "High" severity findings will break the build (Exit Code 1), preventing the insecure image from being pushed.
8. **Registry Push**: Only after passing all the above gates is the image tagged and pushed to **Docker Hub**, marking it as a "Verified Artifact."

4.2. Continuous Deployment (CD)

The CD pipeline is the mechanism of delivery. It listens for the successful completion of the CI pipeline.

1. **Deployment Trigger**: Upon a successful CI run on the `main` branch, the CD workflow initiates.

2. **Infrastructure Connection**: It securely authenticates with the self-hosted Kubernetes cluster runner.
3. **Manifest Application**: The pipeline applies the Kubernetes manifests (`deployment.yaml` and `service.yaml`). This instructs the cluster to pull the new image from Docker Hub and perform a rolling update.
4. **Rollout Verification**: The pipeline does not assume success; it explicitly waits for the Kubernetes rollout status to report "Complete," ensuring that the new pods are healthy and serving traffic before marking the deployment as successful.

5. Security & Quality Controls

This project implements a "Defense in Depth" strategy, applying security controls at multiple layers of the stack.

- **Application Layer (SAST)**: CodeQL provides deep visibility into the code logic, catching vulnerabilities that traditional testing might miss.
- **Dependency Layer (SCA)**: By using Go Vulncheck, we ensure our supply chain is secure. We acknowledge that modern software is assembled, not just written, and third-party risk must be managed.
- **Infrastructure Layer (Container Security)**: Trivy ensures that the runtime environment itself is hardened.
- **Runtime Minimization (Distroless)**: We utilize Google's **Distroless** base images (`gcr.io/distroless/static-debian12`). These images contain *only* the application and its runtime dependencies. They lack a shell (`/bin/sh`), package manager (`apt`), and standard linux utilities. This drastically significantly reduces the attack surface; even if an attacker manages to exploit the application, they cannot easily escalate privileges or run malicious scripts.

6. Results & Observations

The implementation of this DevSecOps pipeline yielded significant measurable improvements in both velocity and stability:

- **Security Posture**: The project achieved a state of **Zero Critical Vulnerabilities** in the production artifact. The automated gates successfully blocked the introduction of insecure dependencies during development testing.
- **Code Quality Assurance**: 100% of the codebase complies with strict linting rules, resulting in a cleaner, more maintainable code structure.
- **Reliability**: The automated testing suite ensures that no regressions are introduced when adding new features. The URL Shortener reliably handles edge cases (e.g., invalid URLs, non-existent codes) gracefully.
- **Operational Excellence**: The deployment pipeline demonstrated seamless **Zero-Downtime Updates**. By leveraging Kubernetes rolling updates, the service remained available to users even while the backend application was being upgraded.

7. Limitations & Improvements

While the current implementation serves as a robust reference architecture for DevSecOps, real-world enterprise constraints would necessitate further enhancements.

Current Limitations

- **Volatility**: The current In-Memory storage implementation means that data is ephemeral. If a pod crashes or is restarted by Kubernetes, all shortened URLs are lost.
- **Access Control**: The API is currently public. There is no rate limiting or authentication mechanism, making it potentially vulnerable to Denial of Service (DoS) attacks or abuse.
- **Scalability Constraints**: The service is currently configured as a single replica, which creates a single point of failure (SPOF) if the underlying node fails.

Future Roadmap

1. **Persistent Storage Layer**: We plan to integrate a high-performance database like **Redis** (for caching) and **PostgreSQL** (for durable storage) to ensure data persistence across restarts.
2. **Authentication & Authorization**: Implementing an API Gateway or OAuth middleware to require API keys for shortening URLs, allowing for usage tracking and rate limiting.
3. **High Availability**: Configuring Horizontal Pod Autoscaling (HPA) to automatically scale the number of pods based on CPU/Memory usage, ensuring the service can handle sudden traffic spikes.
4. **Observability Stack**: Deploying **Prometheus** for metrics collection (RPS, Latency) and **Grafana** for visualization dashboards to provide real-time insights into system health.

End of Report