DevOps Project Report: Containerized Luhn Validator with Observability

DevOps Engineer

January 16, 2026

1 Project Overview

This project is a resilient, containerized microservice designed to validate credit card numbers using the **Luhn Algorithm**. Beyond simple validation, the application identifies the card issuer (Visa, Mastercard, etc.) and exposes real-time telemetry data.

The system is built using **Node.js**, containerized with **Docker**, and orchestrated via **Kubernetes (Kind)** to ensure high availability. It features a robust observability stack using **Winston** for structured logging and **Prometheus** for metrics collection.

2 System Architecture

The infrastructure mimics a production-grade DevOps environment using a local **Kind** (Kubernetes in Docker) cluster.

- **Application Layer:** A Node.js Express API running in replicated Pods for load balancing.
- **Orchestration Layer:** Kubernetes manages the lifecycle, self-healing (restarts), and scaling of the application.
- **Network Layer:** Custom `NodePort` configurations expose the application and monitoring services directly to the host machine (Localhost) without manual port forwarding.
- **Observability Layer:**
  - *Logging:* JSON-structured logs for machine parsing.
  - *Monitoring:* A dedicated Prometheus instance scrapes metrics from the application pods via Service Discovery.

3 Technical Implementation

3.1 The Microservice (Luhn-API)

The core application validates credit card checksums.

- **Stack:** JavaScript (Node.js/Express)
- **Key Libraries:** `winston` (Logging), `prom-client` (Metrics).
- **Endpoints:**
  - `POST /validate`: Accepts JSON, returns validity and issuer.
  - `GET /health`: Used by Kubernetes **Liveness Probes**.
  - `GET /metrics`: Exposes RED metrics (Rate, Errors, Duration).

3.2 Containerization (Docker)

The application is packaged into a lightweight, immutable Docker image.

- **Image Strategy:** Optimized for local development using `:latest` tags.
- **Build Command:**

```
docker build -t amineguizani33/luhn-validator:latest .
```

3.3 Kubernetes Orchestration

The deployment utilizes standard Kubernetes manifests to ensure stability.

- **Replicas:** 2 copies run simultaneously for high availability.
- **Self-Healing:** A `livenessProbe` checks health every 10s. K8s automatically restarts frozen containers.
- **Networking:** The Kind cluster uses `extraPortMappings`:
  - **App:** Mapped to `localhost:30080`.
  - **Prometheus:** Mapped to `localhost:30090`.

4 Observability & Monitoring

4.1 Structured Logging

The application outputs **JSON logs** to allow log aggregators to filter by trace ID or status.

```
{"level":"info", "message":"Request Completed", "status":200, "durationMs":"12.4"}
```

4.2 Prometheus Metrics

We integrated the **Pull Model** monitoring strategy. The application records duration via `prom-client`, and Prometheus automatically discovers pods via the annotation `prometheus.io/scrape: "true"`.

5 Deployment Guide

Prerequisite: Docker Desktop, Kind, Kubectl.
Step 1: Build Create Cluster

```
docker build -t amineguizani33/luhn-validator:latest .
kind create cluster --config kind-config.yaml
```

Step 2: Load Images (Offline Mode)

```
kind load docker-image amineguizani33/luhn-validator:latest --name devops
kind load docker-image prom/prometheus --name devops
```

Step 3: Deploy Manifests

```
kubectl apply -f deployment.yaml
kubectl apply -f services.yaml
```

6 Challenges & Solutions

- **Challenge:** Accessing app from Windows without `port-forward`.
  Solution: Configured `extraPortMappings` in Kind and matched `nodePort` in Service YAML.
- **Challenge:** `ImagePullBackOff` errors due to slow internet.
  Solution: Used `imagePullPolicy: IfNotPresent` and `kind load` to inject images from cache.
- **Challenge:** App crashing (`CrashLoopBackOff`).
  Solution: Identified missing dependencies (`winston`) and fixed the `livenessProbe` 404 error by implementing the `/health` route.

7 Conclusion

This project demonstrates a modern DevOps workflow. By moving from a simple Node.js script to a fully orchestrated Kubernetes deployment with monitoring, the system is robust, scalable, and observable.