DevOps Engineer

Project: Monitoring a Containerized URL Shortener Webservice

Objective: To build, containerize, and monitor a functional URL shortener webservice. The entire stack, including the application and its monitoring tools (Prometheus, Grafana), will run locally using Docker.

Description: This project involves creating a webservice that shortens URLs, stores the mapping, and handles redirects. You will then instrument this service to expose custom performance metrics. Finally, you will use Prometheus to collect these metrics and Grafana to build a comprehensive dashboard for visualizing the service's health and usage patterns.

Technologies to use: Docker, Docker Compose, Prometheus, Grafana, a web framework (e.g., Python's Flask or Node.js's Express), and a simple database like SQLite.

Week 1: Build & Containerize the URL Shortener

Tasks:

- **Develop the Webservice:** Create a simple URL shortener application. It should have two primary API endpoints:
    - A `POST` endpoint (e.g., `/shorten`) that accepts a long URL and returns a short code.
    - A `GET` endpoint (e.g., `/<short_code>`) that reads the code, looks up the long URL, and issues a redirect.
- **Add Data Storage:** Use a simple file-based database like **SQLite** to store the mapping between short codes and long URLs. This keeps the project self-contained.
- **Create a Dockerfile:** Write a Dockerfile to package the webservice into a container image.
- **Initial Docker Compose:** Create a `docker-compose.yml` file to define and run your URL shortener service.

Deliverables:

- A functional URL shortener webservice with source code.
- A Dockerfile that builds a runnable image of the service.
- A `docker-compose.yml` file capable of starting the webservice.
- The service running locally and successfully shortening and redirecting URLs.

Week 2: Instrumenting with Custom Prometheus Metrics

Tasks:

- **Instrument with Custom Metrics:** Modify the webservice code. Using a Prometheus client library, add **custom metrics** to track its specific operations:
    - A **counter** for the number of URLs successfully shortened.
    - A **counter** for the number of successful redirects.
    - A **counter** for failed lookups (404 errors).
    - A **histogram** or **summary** to track the request latency for both creating links and redirecting.
- **Configure Prometheus:** Create a `prometheus.yml` file to scrape the `/metrics` endpoint of your webservice.
- **Integrate into Docker Compose:** Add the Prometheus service to your `docker-compose.yml` file.

Deliverables:

- An updated webservice that exposes custom metrics on a `/metrics` endpoint.
- A `prometheus.yml` configuration targeting the webservice container.
- An updated `docker-compose.yml` that runs both the application and Prometheus.
- Confirmation that the custom metrics (URL creation count, latency, etc.) are visible in the Prometheus UI.

Week 3: Advanced Visualization with Grafana

Tasks:

- **Integrate Grafana:** Add the Grafana service to your `docker-compose.yml` stack.
- **Configure Data Source:** Connect Grafana to your Prometheus service as a data source.
- **Build a Service Dashboard:** Create a new Grafana dashboard specifically for the URL shortener. Visualize the custom metrics you created:
    - Graph the **rate** of URL creations and redirections over time.
    - Display the **total count** of shortened links as a single stat.
    - Create a graph for the **95th percentile request latency**.
    - Display the **rate of 404 errors** to monitor bad requests.

Deliverables:

- A full `docker-compose.yml` file orchestrating the webservice, Prometheus, and Grafana.
- A custom Grafana dashboard providing clear, actionable insights into the webservice's performance and usage.
- A running stack where you can create a short URL and see the metrics change on your dashboard in near real-time.

Week 4: Alerting, Persistence, and Documentation

Tasks:

- **Configure Meaningful Alerts:** In Grafana, set up alerts for important events. For example, create an alert that triggers if the rate of 404 errors exceeds a certain threshold or if request latency is consistently high.
- **Enable Data Persistence:** Modify the `docker-compose.yml` file to use **Docker volumes**. Create one volume for the SQLite database file and separate volumes for Prometheus and Grafana data to ensure nothing is lost on restart.
- **Final Testing:** Shut down and restart the entire stack to verify that the database and all monitoring data persist correctly and that alerts are configured.
- **Document the API:** Create a `README.md` file that not only explains how to run the project but also documents your simple API endpoints.

Deliverables:

- A `docker-compose.yml` file that includes persistent volumes for all stateful services.
- Alerting rules configured in Grafana for key performance indicators.
- A fully tested, stable, and persistent local webservice and monitoring stack.
- Comprehensive project documentation including API specifications.