Logz.io CSE Technical Assessment: "Hello, (Observ) World!"

This project demonstrates a complete workflow for deploying a containerized application to Kubernetes, shipping its telemetry data to Logz.io, and deriving value from that data through alerts and dashboards.

This document details the setup, my troubleshooting journey, key decisions, and how to reproduce the project.

How to Run This Project

Prerequisites

- **Docker Desktop**: Make sure it is installed and running.
- **Minikube**: For creating a local Kubernetes cluster.
- **kubectl**: For interacting with the Kubernetes cluster.
- **Helm**: The package manager for Kubernetes.

Step-by-Step Instructions

1. **Clone the Repository**:

```
None

git clone <your-repo-url>
cd <your-repo-name>
```

2. **Start Your Kubernetes Cluster**:

```
None

minikube start
```

3. **Build and Push the Docker Image**:
    - (Requires a Docker Hub account)

```
None

# Replace 'your-username' with your Docker Hub username
docker build -t your-username/hello-observ-world .
docker push your-username/hello-observ-world
```

4. **Deploy the Application to Kubernetes**:
    - Update the `image:` field in `deployment.yaml` with your Docker Hub image name.

```
None

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

5. **Deploy the Logz.io Collector**:
    - Update `logzio-monitoring-values.yaml` with your Logz.io region and tokens.

```
None

helm repo add logzio-helm https://logzio.github.io/logzio-helm
helm repo update
helm install logzio-monitoring logzio-helm/logzio-monitoring -n
monitoring --create-namespace -f logzio-monitoring-values.yaml
```

6. **Access the Application**:

```
None

minikube service hello-app-service
```

7. This will open the application in your browser. Refresh the page and add `?name=<your_name>` to the URL to generate logs.

My Troubleshooting Journey

Throughout this assessment, I collaborated with Google's Gemini to simulate a real-world scenario of a engineer using available resources to solve problems. This involved asking targeted questions to validate my assumptions and find solutions to technical roadblocks

The Initial Setup:

Before starting the project I had a Gemini help me with the order of operations of setting up the web application, Installing the kubectl and helm as well as early homebrew installation.

Application Deployment: The CrashLoopBackOff Cycle

After deploying my application to Kubernetes, I couldn't access it. So I had Gemini help me translate the pod logs using the **kubectl get pods** command which revealed that the pods were in a **CrashLoopBackOff** state. This meant the container was starting, crashing, and endlessly restarting.

- **Investigation**: I used `kubectl logs hello-app-deployment-78d9b7d7cd-vb6hp` to inspect the container logs. Which Gemini suggested as the first step in debugging a pod-level issue.
- **Discoveries & Fixes**:
    1. **Syntax Errors**: The logs revealed several Python `SyntaxError`s caused by invisible non-printable characters. I fixed these by manually retyping the problematic lines.
    2. **Dependency Conflict**: After fixing the syntax, a new `Error` appeared. This was due to a version conflict between Flask and its dependency, Werkzeug. I resolved this by pinning both libraries to compatible versions in `requirements.txt`.

With each code fix, I went through the full CI/CD cycle: **rebuild** the Docker image, **push** it to Docker Hub, and trigger a **rolling restart** of the Kubernetes deployment with `kubectl rollout restart deployment <deployment-name>`.

Logz.io Collector: Finding the Right Chart

After getting the application running, one of the most significant challenges was discovering that the Helm chart name suggested by Gemini was actually incorrect and made up.

- **Initial Attempt and Investigation** : Based on this, I searched through **helm search repo logzio** and found `logzio-k8s-telemetry` to be a possible chart. Once I installed this collector I found that the log-collecting components were never deployed, only metrics and APM.

  - After this I made sure the logging was not disabled — Did not work.
  - I noticed there was a Logzio-logs-collector.
    - I asked Gemini "Can we install both Logzio-logs-collector and `logzio-k8s-telemetry` to achieve full observability?"
      - This was not suggested because it would cause duplicated data and wasted resources.

- **The Breakthrough**: After a few dead ends, I reached out to the team and provided my trouble shooting experience. I was pointed to the logzio-k8s-telemetry github and found that `logzio-k8s-telemetry` is a sub-chart focused on metrics. The correct approach was to use the logzio-monitoring **umbrella chart**, which installs and configures all necessary components—including a dedicated logs collector—in one go.
- **Solution**: I switched to the `logzio-monitoring` chart. This immediately resolved the issue, and logs began flowing to the platform as expected.

API Alert Update: Validation Errors

When updating the alert via the API, I encountered several validation errors. I worked with the AI to interpret the API's error messages, leading to the discovery of missing fields in the JSON payload (notificationEmails, severityThresholdTiers, alertNotificationEndpoints) and correcting them one by one.

Key Choices & Trade-offs

- **Kubernetes Environment**: I chose **Minikube** because it provides a lightweight, local, and easily reproducible Kubernetes environment without any cloud costs, making it ideal for this assessment.
- **Application Framework**: I used **Python and Flask** due to its simplicity and my familiarity with it. This allowed me to build the required application with minimal code and focus on the core observability tasks.
- **Collector Selection**: My journey to find the correct Helm chart (`logzio-monitoring`) is a key trade-off. While I could have started with a simpler, logs-only collector, my goal was to meet the assessment's "ideal" requirement of collecting logs, metrics, and traces. The troubleshooting process required more effort but resulted in a more complete and powerful observability setup.

API Call

curl -X PUT "https://api.logz.io/v1/alerts/22296326"

```
-H "X-API-TOKEN: 21ccb947-9f7e-45f1-ba5e-dab423d15c25"
 -H "Content-Type: application/json"
-d '{
"title": "Candidate Name Alert (Threshold 50)"
,"query_string": "message:\"*Andrew*\"",
"operation": "GREATER_THAN",
"value_aggregation_type": "COUNT",
"threshold": 50,
"time_window_minutes": 5,
"notificationEmails": ["andrew.zeiser@logz.io"],
"alertNotificationEndpoints": [],
"severityThresholdTiers": []}'
```

What I'd Improve Next

- **Structured Logging**: The current application uses simple string-based logging. As I scale I would improve this by implementing structured (JSON) logging. This would make the logs much easier to query and parse in Logz.io, as each piece of data would become a dedicated field automatically.
- **CI/CD Automation**: The process of building, pushing, and redeploying was manual. I would set up a simple CI/CD pipeline using GitHub Actions to automate this workflow on every code change.
- **Cost Considerations**: While I didn't enable them for this small project, if it were to scale I would implement cost management features like drop filters to discard noisy, low-value logs (like INFO logs from healthy services) and reduce ingestion volume.