Project: Elastic GPU Telemetry Pipeline with Message Queue

Design and implement an elastic, scalable and stable telemetry pipeline for an AI Cluster with messaging queue. The AI cluster contains multiple hosts; each host may contain one or more GPUs. You will build a custom **message queue** (do **not** use existing MQs like ZeroMQ, RabbitMQ, Kafka, etc.)

The system will have these main components:

Telemetry Streamer: reads telemetry from CSV and streams it periodically over the custom message queue. The implementation should support the ability to dynamically scale up/down the number of Streamers.

> NOTE: The data to be streamed is provided as a CSV file. Each line in the CSV is an independent telemetry datapoint. This data can be streamed in a loop to simulate continuous stream of telemetry. The time at which a specific telemetry log is processed should be considered as the timestamp of that telemetry.

Telemetry Collector: consumes telemetry from the custom message queue, parses and persists it. The implementation should support the ability to dynamically scale up/down the number of Collectors.

API Gateway: REST API exposing telemetry. The OpenApi spec for the REST API should be auto generated

Messaging Queue: Messaging system will be used to connect the streamers with the collectors, this is something you will need to implement either as a library or as a service. Please spend some design cycles on thinking about scale, performance and availability of this system. For the exercise we will not scale the nodes beyond 10 instances for the streamer/collector.

All components must be deployable to Kubernetes using Helm charts.

You are expected and encouraged to use AI assistance.

🛠 Tech Stack

Programming Language: Golang

Deployment: Docker + Kubernetes
Deployment Tooling: Helm

Deliverables

Submit a **Git repository** containing

Source Code

- Applications that make up the full software stack

Testing

- **Unit tests** are required
- **System tests** are a bonus (not mandatory)

Packaging & Deployment

- **Dockerfiles** for containerizing your applications
- **Helm charts** to install your stack on Kubernetes

API Documentation

- OpenAPI (Swagger) specification for your REST APIs

README

A comprehensive **README** with:

- Clear and detailed writeup of system **architecture** and design considerations
- **Build and packaging** instructions
- **Installation workflow**
- **Sample user workflow**
- **Description of how AI assistance was used**

Document your use of AI

Detailed description of the **development workflow** to illustrate which aspects of the workflow were accelerated / aided by AI vs manually. Aspects like:

- How was the project / repo bootstrapped? If AI was used, document the prompts.
- How was the code bootstrapped? If AI was used, document the prompts.
- How was the unit test developed? If AI was used, document the prompts.
- How was the build env bootstrapped? If AI was used, document the prompts.

API Requirements

Design and implement the following endpoints:

1. List All GPUs

- Return a list of all GPUs for which telemetry data is available.

2. Query Telemetry by GPU

- Return all telemetry entries for a specific GPU, ordered by time.
- Support optional **time window filters**:
  - start_time (inclusive)
  - end_time (inclusive)

Example API Design:

```
GET /api/v1/gpus
GET /api/v1/gpus/{id}/telemetry
GET /api/v1/gpus/{id}/telemetry?start_time=...&end_time=...
```

Success Criteria

- Keep the scope focused but feel free to showcase your strengths in clean design, extensibility, observability, etc.
- Bootstrap an application and drive it to production.
- Write clean, maintainable systems-level code.
- Handle error paths and memory management gracefully
- Ability to use AI tools and AI assistance to develop and deliver. The more the better.
- Ability to measure and display code coverage through unit tests run on the code base using Makefile.
- The OpenApi spec for the REST API should be auto generated. Expose a Makefile command to generate the OpenApi spec.
- Please provide detailed information of all the prompts used and a good description of where a prompt fell short that required manual intervention.
- Bonus points for:
  - Well-documented code and interfaces
  - Clean, idiomatic Go
  - Clear logging and error handling
  - Thoughtful use of AI assistance (code, design, tests, etc.)