Shop Smart AI Recommender - Project Documentation

Version: 1.0
Author: Nazmul Farooquee
Date: August 22, 2025

1. Executive Summary

The **Shop Smart AI Recommender** is a containerized, cloud-native web application that provides an intelligent, conversational interface for product recommendations. Leveraging a Retrieval-Augmented Generation (RAG) architecture, the system answers user queries based on a knowledge base of real product reviews.

This project demonstrates a complete LLMOps lifecycle, from local development and containerization to automated deployment and monitoring on the Google Cloud Platform (GCP) using Kubernetes. The application is designed to be scalable, secure, and observable, utilizing a modern tech stack including LangChain, Flask, Docker, Kubernetes, Prometheus, and Grafana.

2. System Architecture

The application follows a microservices-oriented architecture, containerized with Docker and orchestrated by Kubernetes (Minikube).

Core Components:

1. **Conversational AI Backend**: A Flask application that serves the RAG chain via a REST API.
2. **Vector Database**: Astra DB (cloud-native Cassandra) stores the vectorized product review data for efficient similarity searches.
3. **Frontend**: A responsive single-page web interface built with HTML, CSS, and jQuery.
4. **Containerization**: Docker encapsulates the application and its dependencies into a portable image.
5. **Orchestration**: Kubernetes (Minikube) manages the deployment, scaling, and networking of the application containers.
6. **Monitoring Stack**: Prometheus scrapes key application metrics, and Grafana provides a dashboard for visualization and alerting.

Technology Stack:

- **Language**: Python 3.10
- **AI/ML Frameworks**: LangChain, Groq, Hugging Face `sentence-transformers`
- **Web Framework**: Flask
- **Database**: Astra DB (Vector Store)
- **Containerization**: Docker
- **Orchestration**: Kubernetes (Minikube)

- **Monitoring**: Prometheus, Grafana
- **Cloud Platform**: Google Cloud Platform (GCP)

---

3. Codebase Structure

The project is organized into distinct modules to ensure a clean separation of concerns.

```
/
├── .github/ # (Future) CI/CD workflows
├── chain/ # Core RAG chain logic
│ ├── init.py
│ └── rag_chain.py
├── config/ # Application configuration
│ ├── init.py
│ └── config.py
├── data/ # Raw dataset
│ └── flipkart_product_review.csv
├── grafana/ # Grafana Kubernetes manifests
│ └── grafana-deployment.yaml
├── prometheus/ # Prometheus Kubernetes manifests
│ ├── prometheus-configmap.yaml
│ └── prometheus-deployment.yaml
├── static/ # CSS and other static assets
│ └── style.css
├── templates/ # HTML templates
│ └── index.html
├── utils/ # Reusable helper modules
│ ├── init.py
│ ├── custom_exception.py
│ ├── data_converter.py
│ ├── data_ingestion.py
│ └── logger.py
├── .env # (Local Only) Secret keys and APIs
├── .gitignore # Files to be ignored by Git
├── app.py # Main Flask application entry point
├── Dockerfile # Instructions to build the container image
├── flask-deployment.yaml # Kubernetes manifest for the Flask app
├── requirements.txt # Python dependencies
└── setup.py # Project packaging script
```

---

4. Deployment Pipeline

The deployment process follows these key stages:

1. **Prerequisites**: A GCP VM instance is created and configured with Docker and Minikube.

2. **Code Sync**: The latest version of the code is pulled from the GitHub repository.
3. **Environment Setup**: The `eval $(minikube docker-env)` command is run to direct the local Docker client to the Docker daemon inside the Minikube cluster.
4. **Image Build**: The application is built into a Docker image using the `Dockerfile`. Because of the previous step, this image is built directly within Minikube's environment, making it immediately available to the cluster.
5. **Secrets Management**: A Kubernetes secret is created from a manually created `.env` file on the VM to securely inject API keys into the application pod.
6. **Application Deployment**: The Kubernetes manifests for the application (`flask-deployment.yaml`) and the monitoring stack (`prometheus-*.yaml`, `grafana-*.yaml`) are applied using `kubectl apply`.
7. **Service Exposure**: The application is exposed to the internet via a Kubernetes `Service` of type `LoadBalancer` or accessed for testing via `kubectl port-forward`.

---

5. Monitoring and Observability

The application is instrumented for monitoring using Prometheus.

- **Metrics Exposure**: The Flask application exposes a `/metrics` endpoint, providing key counters (`http_requests_total`, `http_requests_success_total`, `http_requests_error_total`).
- **Service Discovery**: The Prometheus server is configured with `kubernetes_sd_configs` to automatically discover and scrape the application's `/metrics` endpoint.
- **Visualization**: Grafana is deployed and pre-configured with Prometheus as a data source, allowing for the creation of dashboards to visualize application health and performance in real-time.

This setup provides a robust foundation for observing the application's behavior in a production-like environment.