QnA-RAG Agent

Kubernetes Cluster Workflow/Processflow

Introduction

This document outlines the complete workflow and process flow for deploying and managing services in a Kubernetes-based infrastructure. It serves as a comprehensive guide for developers, DevOps engineers, and system administrators working with microservices-based applications in a cloud-native environment.

Purpose of the Document

To describe the technologies, configurations, and operational steps involved in deploying and managing a QnA-RAG microservices architecture using Kubernetes and associated tools.

Tech Stack Overview

- **Kubernetes**: Container orchestration and infrastructure management
- **Python**: Core language for QnA and RAG services (FastAPI)
- **Prometheus**: Monitoring and metrics collection
- **Grafana**: Metrics visualization
- **ArgoCD**: GitOps continuous deployment tool
- **FastAPI**: Python web framework for microservices
- **Terraform**: Infrastructure as code (IaC) for provisioning
- **Ansible**: Automation of environment configuration
- **KEDA**: Kubernetes Event-Driven Autoscaling
- **OPA (Open Policy Agent)**: Policy enforcement and security
- **APISIX**: API gateway and traffic management

Workflow Diagram

As shown in the Diagram.

Diagram Annotations

- Client → NGINX Ingress Controller
- Ingress → QnA API → RAG Agent (via APISIX load balancer)
- RAG Agent ↔ Vector DB (YugabyteDB)
- RAG Agent ↔ LLM via Model API (e.g., OpenAI)
- Prometheus scrapes `/metrics` from QnA and RAG
- Grafana visualizes metrics
- KEDA scales services based on Prometheus and CPU
- NetworkPolicy and CronJobs enforce inter-service rules

Detailed Workflow Steps

Step 1: Infrastructure Setup

Tools Used: Kubernetes (current), Terraform and Ansible (planned)

- Initially, infrastructure was manually managed using Kubernetes YAML files. Terraform will be used in the future to provision Kubernetes clusters.
- Configure network policies (e.g., `qna-rag-NetworkPolicy.yaml`)
- Apply ConfigMap and Secrets (e.g., `agents-configmap.yaml`)
- Define constraints with Gatekeeper templates and policies (`dissallowed-tag-template.yaml`, `dissallowed-tag-constraint.yaml`)
- Ansible is installed and will be used in subsequent phases to automate environment setup (e.g., Prometheus stack, ArgoCD).

Step 2: Application Deployment

Tools Used: Docker, Kubernetes, ArgoCD

- Build Docker images for QnA and RAG services
- Deploy services using manifests (`qna-deployment.yaml`, `rag-deployment.yaml`)
- Configure APISIX route load balancing (`apisixLoadbalancer2.yaml`)
- Ingress is defined via `agents-ingress.yaml`
- Use ArgoCD for GitOps deployment from repository

Step 3: Monitoring and Scaling

Tools Used: Prometheus, Grafana, KEDA

- Prometheus setup using `prometheus-values.yaml` and additional scrape config (`qna-rag-scrape-config.yaml`)
- Services expose `/metrics` via annotations in deployment manifests
- Grafana deployed with PVC support (`grafana.yaml`)
- KEDA configured to autoscale based on CPU and Prometheus metrics (`rag-scaledobject`, `qna-scaledobject` in their respective deployments)

Step 4: Security and Compliance

Tools Used: OPA/Gatekeeper, NetworkPolicy, CronJobs

- Policy enforcement with ConstraintTemplate and Constraint manifests
- CronJobs schedule application/removal of network policies (`cronjob-block.yaml`, `cronjob-unblock.yaml`)
- Secure env variables via Kubernetes Secrets (`agents-secret.yaml`)
- Gatekeeper ensures disallowed tags aren't used in container images

Development & Testing Setup

- Set up Python virtual environment using `venv` or `conda`
- Install dependencies using `requirements.txt` or `requirements-dev.txt`
- Create `.env` file with environment variables: `OPENAI_API_KEY`, `PINECONE_API_KEY`, `RAG_AGENT_URL`

- Run tests using `pytest`, optionally with coverage reporting

Deployment Methods

Method 1: Local Development

- Start services using `uvicorn` on ports 8000 (QnA) and 8001 (RAG)

Method 2: Docker Deployment

- Build and run services with Docker images and mapped ports

Method 3: Kubernetes Deployment

- Apply config maps, secrets, deployments, ingress, and monitoring configs

API Endpoints

QnA Service (Port 8000)

- `GET /` - Health check
- `POST /query` - Accepts user queries

RAG Service (Port 8001)

- `GET /` - Health check
- `POST /query` - Retrieves documents and responds using LLM

CI/CD Pipeline

- GitLab CI handles testing, coverage, image builds
- Uses `.gitlab-ci.yml` with automated test pipelines

Vector Store Configuration

- Vector DB: Pinecone
- Embedding model: OpenAI `text-embedding-3-small`
- Dimension: 1536, Similarity: cosine

Troubleshooting Tips

- **No metrics in Prometheus?** Ensure `prometheus.io/scrape` annotations are enabled
- **Services not scaling?** Double check KEDA triggers and Prometheus accessibility
- **APISIX routing fails?** Validate hostnames and health check rules
- **NetworkPolicy issues?** Ensure cron jobs apply/delete policies correctly

Best Practices

- Follow GitOps principles with ArgoCD
- Use PVCs for persistent state (Grafana dashboards)
- Avoid `latest` image tags via Gatekeeper constraint

- Apply health checks and probes for resilient containers
- Regularly monitor and adjust KEDA threshold settings

Conclusion

This workflow provides a robust, scalable, and secure method to deploy and manage QnA-RAG services using Kubernetes. Through efficient use of tools like Prometheus, KEDA, ArgoCD, and APISIX, the architecture ensures high availability, observability, and compliance across environments.

Repository

The source code for this project is available on GitHub: LangGraph-AI-Agents