# Distributed IoT Data Processing System with Kubernetes

## Project Overview

This project implements a distributed system for processing IoT sensor data using Apache Kafka, Docker, and machine learning components, now deployed on a Kubernetes cluster. The system processes CIFAR-100 dataset images through a machine learning pipeline, with components distributed across multiple nodes in a Kubernetes cluster.

## System Architecture

### Infrastructure Components

1. **Kubernetes Cluster**
   * 4-node cluster (1 master + 3 workers)
   * Master node also configured as worker through taint removal
   * Private Docker registry running on master node (VM1)
2. **Virtual Machines**
   * 4 VMs running Ubuntu 22.04 LTS
   * Each VM: m1.medium flavor (2 vCPUs, 3.9GB RAM)
   * Connected via private network
   * Configured with necessary firewall rules for K8s and registry

### Application Components

1. **Private Docker Registry**
   * Hosted on master node (VM1)
   * Stores custom Docker images for:
     + IoT Data Producer
     + ML Inference Service
     + Database Updater
2. **Kubernetes Deployments**
   * Kafka Broker (using official Apache Kafka image)
   * ML Inference Service (custom image)
   * PostgreSQL Database (official image)
   * Producer Jobs (custom image, scalable 1-5 instances)
   * Consumer Deployments (custom image)

## Deployment Architecture

### Kubernetes Configuration

1. **Master Node (VM1)**
   * Runs control plane components
   * Hosts private Docker registry
   * Configured as both master and worker
   * Firewall rules for K8s API server (6443)
   * Registry port (5000) opened
2. **Worker Nodes (VM2-4)**
   * Run application workloads
   * Connected to master node
   * Configured with necessary K8s components
   * Access to private registry

### Network Configuration

1. **Kubernetes Networking**
   * Pod network: 10.244.0.0/16 (Flannel)
   * Service network: 10.96.0.0/12
   * NodePort services for external access
2. **Security**
   * Network policies for pod-to-pod communication
   * Firewall rules for:
     + K8s API server (6443)
     + Kubelet (10250)
     + NodePorts (30000-32767)
     + Private registry (5000)

## Implementation Details

### 1. Infrastructure Setup

#### Ansible Playbook Extensions

- name: Install K8s Packages  
 apt:  
 name:  
 - kubelet  
 - kubeadm  
 - kubectl  
 state: present  
  
- name: Configure Private Registry  
 docker\_container:  
 name: registry  
 image:   
 ports:  
 - "5000:5000"  
 restart\_policy: always  
  
- name: Update Containerd Config  
 template:  
 src: config.toml.j2  
 dest: /etc/containerd/config.toml

### 2. Kubernetes Deployments

#### Kafka Deployment

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: kafka  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: kafka  
 template:  
 spec:  
 containers:  
 - name: kafka  
 image: apache/kafka:latest  
 ports:  
 - containerPort: 9092

#### ML Inference Service

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: ml-inference  
spec:  
 replicas: 1  
 template:  
 spec:  
 containers:  
 - name: inference  
 image: localhost:5000/ml-inference:latest  
 ports:  
 - containerPort: 5000

### 3. Producer Scaling

#### Producer Job Template

apiVersion: batch/v1  
kind: Job  
metadata:  
 name: iot-producer  
spec:  
 parallelism: ${PRODUCER\_COUNT}  
 template:  
 spec:  
 containers:  
 - name: producer  
 image: localhost:5000/iot-producer:latest  
 env:  
 - name: PRODUCER\_ID  
 valueFrom:  
 fieldRef:  
 fieldPath: metadata.name

## Performance Analysis

### Workload Variations

* Tests conducted with 1-5 producer instances
* Each producer sends 1000 samples
* Sample frequency: 1 per second

### Metrics Collection

1. End-to-end latency per message
2. 90th percentile response time
3. 95th percentile response time
4. System resource utilization

### Performance Results

1. **Single Producer**
   * 90th percentile:
   * 95th percentile:
2. **Multiple Producers (2-5)**
   * Scaling impact on latency
   * Resource utilization patterns
   * Network performance metrics

## Security Considerations

1. **Container Security**
   * Private registry with authentication
   * Container image scanning
   * Resource limits enforcement
2. **Network Security**
   * K8s network policies
   * Firewall rules
   * Inter-pod communication restrictions
3. **Access Control**
   * RBAC policies
   * Service account management
   * API server authentication

## Monitoring and Logging

1. **Kubernetes Monitoring**
   * Kubectl get pods/services/deployments
   * Pod logs access
   * Resource utilization tracking
2. **Application Monitoring**
   * Producer/Consumer metrics
   * ML inference performance
   * Database operations

## Future Improvements

1. **High Availability**
   * Multi-master K8s setup
   * Kafka cluster scaling
   * Database replication
2. **Performance Optimization**
   * Resource allocation tuning
   * Network optimization
   * Cache implementation
3. **Monitoring Enhancements**
   * Prometheus integration
   * Grafana dashboards
   * Alert management
4. **Security Hardening**
   * Network policy refinement
   * Secret management
   * Security scanning automation

## Conclusion

This implementation demonstrates a successful migration of the IoT data processing pipeline to a Kubernetes-based infrastructure. The system maintains its core functionality while gaining the benefits of container orchestration, including:

1. Automated scaling of producers
2. Better resource utilization
3. Simplified deployment process
4. Enhanced monitoring capabilities
5. Improved system reliability

The performance analysis shows that the system can effectively handle varying workloads through Kubernetes’ native scaling capabilities, while maintaining acceptable latency metrics across different producer configurations.