# 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

**MapReduce Implementation for Per-Producer Error Counting**  
  
The MapReduce paradigm was employed using Apache Spark to analyze the performance of an inference server. Specifically, we computed the number of incorrect inferences on a per-producer basis. This extends the classical MapReduce Word Count example, adding complexity by considering per-producer granularity instead of a holistic dataset approach.  
  
Implementation Details  
  
1. Data Input and Preprocessing:  
 - The input data, stored in `MapReduce\_data.csv`, comprised fields such as producer IDs, inferred classes, and ground truth values.  
 - A mapping dictionary was created to associate class labels with numeric class IDs to facilitate computation.  
  
2. Mapping Phase:  
 - The data was filtered to isolate incorrect inferences, where the `inferred\_value\_id` differed from the `ground\_truth`.  
 - Each filtered record was then mapped to its corresponding producer for grouping.  
  
3. Reducing Phase:  
 - Aggregation was performed using Spark’s `groupBy` and `count` functions to compute the number of incorrect inferences for each producer.  
 - The resulting counts were sorted in descending order to identify producers with the highest error rates.  
  
4. Key Observations:  
 - By focusing on per-producer analysis, this approach allowed us to pinpoint problematic data sources or faulty configurations contributing disproportionately to errors.  
 - The use of Apache Spark ensured scalability, enabling the processing of large datasets efficiently.  
  
5. Output Example:

A screenshot of a table

Description automatically generated  
  
This implementation demonstrates the power of distributed data processing for error analysis in an IoT pipeline, leveraging the capabilities of Apache Spark to achieve precise and actionable insights.  
  
Code Highlight  
  
The following code snippet illustrates the core logic of the MapReduce implementation:  
  
```python  
incorrect\_inferences = df.where(col("inferred\_value\_id") != col("ground\_truth")) .groupBy("producer") .agg(count("\*").alias("incorrect\_inferences\_count")) .orderBy("incorrect\_inferences\_count", ascending=False)  
  
for row in incorrect\_inferences.collect():  
 print(f"{row.producer:<40} {row.incorrect\_inferences\_count:>10}")  
```  
  
The modularity and efficiency of this code align with the principles of distributed systems and parallel computation.