# go-k8s-data-pipeline

Building high performance data pipeline with Go and k8s

# Building Scalable Data Pipelines: Using Go, Kafka (KRaft Mode), and Kubernetes

### Introduction

In today's data-driven world, businesses generate massive amounts of data every second. Processing, transforming, and managing this data efficiently requires **scalable data pipelines**. If you're an absolute beginner and wondering how to build such a pipeline using **Go, Kafka (in KRaft mode), MongoDB, and Kubernetes**, this guide is for you!

We'll walk through the **fundamentals of data pipelines**, the **role of Go, Kafka (KRaft), and Kubernetes**, and how to build a **simple, scalable data pipeline** step by step.

---

## Step 1: Understanding Data Pipelines

A **data pipeline** is a set of processes that ingest, process, transform, and store data efficiently. Common components include:

1. **Data Ingestion** – Collecting raw data from different sources.
2. **Processing & Transformation** – Cleaning, filtering, and processing data.
3. **Storage** – Saving processed data in databases or cloud storage.
4. **Serving & Visualization** – Making data available for analytics, dashboards, or other applications.

Modern data pipelines should be **scalable, resilient, and fault-tolerant**—which is where **Go, Kafka (KRaft mode), MongoDB, and Kubernetes** come into play.

---

## Step 2: Setting Up the Local Kubernetes Cluster (Minikube)

For this demo, we'll use **Minikube**, a local Kubernetes cluster.

### 1. Install Minikube

```
# Install Minikube (MacOS/Linux)
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

### 2. Start Minikube Cluster

```
minikube start --memory=8192 --cpus=4
```

Verify installation:

```
kubectl get nodes
```

---

## Step 3: Building a Dummy Data Generator

We need an application to **continuously push large amounts of dummy data to a Kafka topic**.

### 1. Create a Go-based Kafka Producer

```go
package main

import (
    "fmt"
    "log"
    "time"
    "github.com/confluentinc/confluent-kafka-go/kafka"
)

func main() {
    p, err := kafka.NewProducer(&kafka.ConfigMap{"bootstrap.servers":
"kafka.default.svc.cluster.local:9092"})
    if err != nil {
        log.Fatalf("Failed to create producer: %s", err)
    }
    defer p.Close()

    topic := "dummy-data"
    for {
        message := fmt.Sprintf("{\"timestamp\": \"%s\", \"value\": %d}",
time.Now().Format(time.RFC3339), time.Now().UnixNano())
        err := p.Produce(&kafka.Message{TopicPartition:
kafka.TopicPartition{Topic: &topic, Partition: kafka.PartitionAny}, Value:
[]byte(message)}, nil)
        if err != nil {
            log.Printf("Failed to produce message: %s", err)
        }
        time.Sleep(100 * time.Millisecond) // Adjust load rate
    }
}
```

Build and deploy this **Kafka producer** in a Kubernetes **Deployment**.

---

## Step 4: Deploying a Kafka Cluster in KRaft Mode on Kubernetes

We'll use **Bitnami's Kafka Helm chart** to deploy a Zookeeper-free Kafka cluster using **KRaft mode**.

### 1. Add Bitnami Helm Repository

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update
```

### 2. Install Kafka in KRaft Mode

```
helm install kafka bitnami/kafka --set kraft.enabled=true
```

This sets up a fully functional Kafka cluster without Zookeeper.

---

## Step 5: Building an ETL Application in Golang

The **ETL (Extract, Transform, Load) Application** will:

- Extract data from Kafka.
- Transform data into an upsert format.
- Load data into MongoDB.

### 1. Go-based Kafka Consumer & MongoDB Upserter

```
package main

import (
    "context"
    "log"
    "time"
    "github.com/confluentinc/confluent-kafka-go/kafka"
    "go.mongodb.org/mongo-driver/bson"
    "go.mongodb.org/mongo-driver/mongo"
    "go.mongodb.org/mongo-driver/mongo/options"
)

func main() {
    consumer, _ := kafka.NewConsumer(&kafka.ConfigMap{"bootstrap.servers":
"kafka.default.svc.cluster.local:9092", "group.id": "etl-group",
"auto.offset.reset": "earliest"})
    consumer.Subscribe("dummy-data", nil)

    client, _ := mongo.Connect(context.TODO(),
options.Client().ApplyURI("mongodb://mongo-service:27017"))
```

```
    collection := client.Database("etl_db").Collection("data")

    for {
        msg, err := consumer.ReadMessage(-1)
        if err == nil {
            data := bson.M{"timestamp": time.Now(), "value": string(msg.Value)}
            _, err := collection.InsertOne(context.TODO(), data)
            if err != nil {
                log.Println("Insert error:", err)
            }
        }
    }
}
```

---

## Step 6: Deploying MongoDB on Kubernetes

Create `mongodb.yaml` manifest:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongodb
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mongodb
  template:
    metadata:
      labels:
        app: mongodb
    spec:
      containers:
      - name: mongodb
        image: mongo
        ports:
        - containerPort: 27017
```

Deploy MongoDB:

```
kubectl apply -f mongodb.yaml
```

---

## Step 7: Autoscaling the ETL Application

We will configure **Horizontal Pod Autoscaler (HPA)** to scale our ETL application based on Kafka load.

Apply HPA Policy

```
kubectl autoscale deployment etl-app --cpu-percent=50 --min=1 --max=10
```

---

## Step 8: Benchmarking Performance

Use **k6** to simulate load:

```
k6 run load-test.js
```

Monitor Kafka consumer lag and MongoDB performance using **Prometheus & Grafana**.

---

## Conclusion

- ✅ **Fully automated Kafka-based data pipeline** in Go & Kubernetes (Zookeeper-free).
- ✅ **Auto-scaled ETL logic** ensures efficient performance.
- ✅ **Step-by-step deployment & benchmarking** for real-world readiness.

💡 **Next Steps**: Build advanced analytics, use cloud-native services, or optimize for large-scale production!

📌 **Full GitHub repository** coming soon!